Kurt Meinz

CS61A Course Reader
Summer 2005

Table of Contents:

UNIVERSITY of CALIFORNIA at Berkeley
Department of Electrical Engineering and Computer Sciences
Computer Sciences Division

CS61A
Course Info

Kurt Meinz
Summer 2005

CS61A: The Structure and Interpretation of Computer Programs
General Course Information

1 Introduction

The CS 61 series is an introduction to computer science, with particular emphasis on software and on machines from a programmer's point of view. This first course concentrates mostly on the idea of *abstraction*, allowing the programmer to think in terms appropriate to the problem rather than in low-level operations dictated by the computer hardware. The next course, CS 61B, will deal with the more advanced engineering aspects of software—on constructing and analyzing large programs and on techniques for handling computationally expensive programs. Finally, CS 61C concentrates on machines and how they carry out the programs you write.

In CS 61A, we are interested in teaching you about programming *per se* rather than any programming language in particular. We consider a series of techniques for controlling program complexity, such as functional programming, data abstraction, object-oriented programming, and deductive systems. Of course, to get past generalities you must have programming practice in some particular language, and, in this course, we will use Scheme, a dialect of Lisp. This language is particularly well-suited to the organizing ideas we want to teach. Our hope, however, is that once you have learned the essence of programming, you will find that picking up a new programming language is but a few days' work.

2 Do You Belong Here?

The summer session version of this course is a bit different from the regular semester version. We cover all of the usual material, but we do it in **half** the time. This makes the course *very fast.* If you fall behind, you will find it almost impossible to catch up. At the same time, the summer course has no restrictions on enrollment. Anyone, regardless of prior experience may enroll in the course (until it fills.) We encourage anyone who's curious or interested to take this course, even if they aren't computer science majors!

With that being said, this course will be difficult and time-consuming. Your nominal classroom hours are roughly 12hrs/week – however, you can expect to be spending, at the very least, another 20hrs/week on readings and assignments. If you have other time commitments, such as a summer job or another summer course, you may find yourself stretched too thin. In short, this course will be like a full-time job, so please plan accordingly.

This course expects some logical sophistication, but does not actually require any prior programming experience. During the regular semester, Math 1A is a corequisite for 61A, and there is generally a placement exam to test whether or not you are familiar with *recursion* or *induction.* (For examples, go to `http://www-inst.eecs.berkeley.edu/∼cs61a/miscellaneous/entrance.html`.)

We have found that 80% to 90% of 61A students have had significant prior programming experience, and that students without such experience are at an initial disadvantage. There is no need for you to be familiar with any particular programming language, although if all of your experience has been in BASIC then you probably haven't used recursion. In addition, the computer labs for the course use UNIX machines. You may find it time-consuming and sometimes difficult to do the labs and homework if you have not spent time becoming familiar with UNIX.

Therefore, it is up to you to decide if you are prepared for this course. Check out the course materials yourself, and play around with the labs and homework. My advice is to take the risk and get out as much as you possibly can! If you are still unsure, you can speak to me about it, however, if you ask my opinion, I will probably say that you should take it because the course is wonderful and you will learn a great deal from taking it (regardless of your final grade).

If you don't feel ready for 61A, we recommend that you take CS 3, which is a Scheme-based introductory programming course, or CS 3S, the self-paced version. CS 3 and 3S are directed primarily at students who are not Computer Science majors, but they are also designed to serve as preparation for 61A. You could then take 61A next semester. If you are interested in learning how to program specifically in C or Java, there are engineering courses to teach you these courses, and they will server you better than this course.

If you are not strongly interested in computer *programming* at all, but instead want to learn how to *use* computers as a tool, you should consider IDS 110, a course that presents a variety of personal computer software along with a brief introduction to programming.

If you have substantial prior programming background, you may feel that you can skip 61A. In most cases, we don't recommend that. Although 61A is the first course in the CS sequence, it's quite different from most introductory courses. Unless you have used this same textbook elsewhere, I think I can promise that you won't be bored. If you're not convinced, spend some time looking over the book and then come discuss it with me. Instead, perhaps your prior experience will allow you to skip 61B or 61C, which are more comparable to courses taught elsewhere. See Mike Clancy in the CS department about this.

3 Course Materials

The textbook for this course is *Structure and Interpretation of Computer Programs* by Abelson, Sussman, and Sussman, second edition. It should be available in the textbook section of the ASUC bookstore and other local textbook sellers. **You must get the 1996 second edition! Don't buy a used copy of the first edition.** A paperback version containing all necessary chapters of version 2 may also be available used at the same books stores. If you cannot afford or don't want to buy the book, copies of it are on reserve at the Engineering Library. Also, the **entire** book is readable online. The URL is given later in this document and on the course website.

In addition to the textbook, there is a reader containing necessary materials, including most assignments, information on our computing facilities in general, and about the Scheme language. You can buy the reader at CopyCentral, 2483 Hearst Avenue (at Euclid.) The summer's reader is unlike the normal term's readers, so don't borrow your housemate's old copy. All of the most important material in the reader will also be available on the course website, so, if you really don't want to buy the reader, you don't have to. However, it has been our experience that most students prefer to purchase the reader.

We have also listed an optional text for the course. This book really is optional! Don't just buy it because you saw it on the shelf. The optional text is *Simply Scheme*, by Harvey and Wright. (Brian Harvey is a professor here.) This is usually used as the textbook for CS 3; it gives a slower and gentler introduction to the first five weeks of 61A, for people who feel swamped here.

If you have a home computer, you may want to get a Scheme interpreter for it. The Computer Science Division can provide you with free versions of Scheme for Linux, Windows, or MacOS. The distribution also includes the Scheme library programs that we use in this course. For more information on how to get your home computer to work well with the course materials, check the 'resources' section of the web site.

The course reader includes the notes for the entire semester, and I will make my presentation slides available either immediately before or immediately after lecture. These notes are provided so that you can devote your efforts during lecture to thinking, rather than to frantic scribbling. In addition, any materials used during lecture but not provided in the reader will be made available on-line.

4 Enrollment—Laboratory and Discussion Sections

Summer session is 8 weeks, with every week packing in two standard course weeks. This course is normally structured so that there is one discussion and one lab meeting each week; but we must pack in both into the first two days of the week, and again, both into the last two days of the week. Generally, the lab portion occurs some time after Monday's lecture and again sometime after Wednesday's lecture. The discussion sections meet after lecture on Tuesdays and Thursdays. You will also need to spend additional time working on the computers in the Soda Hall labs. For most weeks, labs will meet in our laboratory room, 271 Soda Hall and the discussions will be in 310 Soda. Occasionally there may be two lab sessions and no discussions. Be sure to check the website and pay attention in lecture.

The discussion and lab sections are run by our Teaching Assistant, Jeff. We anticipate some rearrangements during the first week in response to oversubscribed or undersubscribed sections. **If you are waitlisted**, you should communicate via email with Jeff about your situation. Please be in a definite discussion section by the end of this week, though, because some of the coursework will be done in groups of two students; you are not allowed to form a group with someone in a different section.

You must have a computer account on the 61A course facility. You must set up your account *before Noon on Wednesday, June 22* because that is how we know who is really in the class. Account forms will be distributed in the LAB SECTIONS. The first time you log in, you will be asked to type in your name and other information. Please follow the instructions carefully. **Be sure to remember the "secret code" you use for registration – you will use that secret code to check your grades on-line.** You must get your account *and log into it* no later than **12:01 PM Wednesday** so that we have an accurate class count. Everyone MUST log in by Wednesday Noon (or have made special arrangements with their TA) **OR YOU WILL BE DROPPED** from the course and someone on the waitlist will take your place!

Some of you have personal computers and may want to do the course work at home. This is fine with us, although you'll have to be careful to install the class Scheme library on your home computer to make your computer's version of Scheme behave like the modified one we use in the lab. In any case, though, you must get a class account even if you intend never to use it.

If you get a class account and then decide to drop the course, please let me know *immediately* so that we can admit another student. Thank you.

5 How to get the most from this course

We recognize that everyone's style of learning is unique. Some students are excellent at studying–they work hard, and are extremely diligent. They do all the readings conscientiously, and work all the problems. Some students are incredibly quick, and get by doing little of the reading, even less of the homework, and still ace the tests. Some students learn best by listening to lecture, and discussing it with their friends and TAs. Some students are aiming for the A+, others just to get by with a passing grade. Usually, students are some of each of these types, or are sometimes one, sometimes another. Since everyone's style is their own, we try to have as many opportunities to learn this material as possible. Therefore, use them all, and learn what works best for you.

That said, we do enforce certain types of interaction. In this course, we encourage and REQUIRE that you learn to work together in groups for certain assignments. This means you will need to learn how to work with people whose strengths are not your own. (This is of course the best thing a group can provide!) It also means you will learn how to work with people whose style you find difficult. But overall, you will learn best by learning to collaborate, and helping each other when one is not getting the material.

Different people solve problems differently; there are often many right answers to the problems in this course. And of course, what you find easy, your friend may find hard, and vice versa. Therefore, the best way to learn is to talk with other people, and ask them questions when you are stuck. Even if you think you understand everything, you will learn the material better if you have to try to explain it to someone else. In addition, learning how to think about the problems in many different ways will solidify your understanding of this material.

Finally, it is possible that some of you feel uncomfortable telling others when you don't understand something. Many of us find it hard to ask questions–all the more reason to overcome this fear early! The ability to ask for help is a wonderful strength that will serve you well in life. Throughout this course, I will try to encourage you to ask each other, the staff, and myself for help.

6 Information Resources

Jeff, the readers, and I available to answer questions. You may drop in during office hours, make appointments for other times, or communicate with us by email. Feel free to visit any of the staff. You may find that hearing different people's explanations helps if at first you do not understand some material.

For technical questions about the homework or projects, or administrative questions such as missing homework grades, send electronic mail to your reader. You can also send mail about intellectual questions to me, but if it's about grades I'll just refer you to your TA.

In addition, there is an electronic bulletin board system that you can use to communicate with other 61A students and staff. The ucb newsgroup can be read only from machines in the berkeley.edu domain, so if your net connection is though a commercial ISP then you must log into a lab machine to read the newsgroup or try this:

`http://www-inst.eecs.berkeley.edu/connecting.html`

Please do not send electronic mail to every student individually! That would waste a lot of disk space, even for a small message. Use the newsgroup instead. Electronic mail is for messages to individuals, not to groups.

There is a web-based reader for the cs newsgroup available from the course homepage, located at

`http://www-inst.eecs.berkeley.edu/~cs61a`

The web page for the textbook, with additional study resources, is

`http://www-mitpress.mit.edu/sicp/sicp.html`

There are also web pages for the Scheme programming language:

`http://swissnet.ai.mit.edu/scheme-home.html`
`http://www.schemers.org/`

Additional information to help you in studying, including hints from the course staff and copies of programs demonstrated in lectures, is available at the course website.

7 Computer Resources

The computing laboratory in 271 Soda Hall consists of about 35 SunRay terminals connected to a Sun Solaris server. This is our primary lab room, although the CS 61A accounts can also be used from any EECS Instructional lab in Soda or Cory Hall.

The lab in 271 Soda is normally available for use at all times, but **you need a card key for access to the lab**; to get a card key, stop by the 3rd floor office of Soda Hall and fill out a form for a card key. You will need a $20 deposit to get the card key. The card key will give you access to the 2nd and 3rd floors of Soda Hall so that you may enter at any time, day or night. Do this today! During scheduled lab sessions, only students enrolled in that particular section may be in the lab. Therefore, you might need to use the other Soda Hall labs to work on homework outside of class. In particular, 273 Soda Hall should be at your disposal at all times. When sections are not in session, any 61A student may use any of the 2nd floor labs on a drop-in basis. If there are no free workstations, please feel free to ask anyone who is not doing course work to leave. In particular, *game playing is not permitted.* We are relying on social pressure to discourage abuse (such as stealing the chairs or monopolizing a workstation for six hours during prime time to play chess). Therefore, do not feel embarrassed to apply such pressure.

These machines use the Unix operating system, a timesharing system that is quite different from the microcomputer systems you have probably seen elsewhere. The course reader includes introductory documentation about Unix and about Emacs, the text editing program we are recommending for your use. (It is one of several Unix text editors; you'll find that everyone has his or her own favorite editor and hates all the others.) Although the use of Unix is not extensively taught in 61A lectures, it will be extremely worthwhile for you to spend some time getting to know how the system works.

If you have a home computer and a modem, you may wish to use your class account remotely. If so, you are encouraged to use a commercial Internet Service Provider to connect to the campus; several companies offer student rates. Again, check out

`http://www-inst.eecs.berkeley.edu/connecting.html`

In addition, you should know that, on occasion, our file servers go on the blink. You can detect this situation by noticing that your terminal has suddenly stopped typing characters or you get a message along the lines of "`NFS server not responding...`". If this happens to you (and it will at least once!), don't panic; usually the server is back within minutes or hours with your data intact. Please do not put yourself in a situation where a couple-hour server crash will prevent you from completing your project on-time. "How can I avoid such a horrible situation?" you may ask. By starting (and finishing) your assignments early, of course!

8 Reading, Homework and Programming Assignments

You should try to complete the *reading* assignment for each week **before** the lecture. You will have four subsequent class meetings (two lectures and two discussion/lab sections) to help you understand the readings. Ideally, you would work in lab and afterward on the exercises, and then complete them the next day after section. If you're efficient, you'll then have that night to read the next reading assignment.

Every week there will be problems assigned for you to work on, many of which will involve writing and debugging computer programs. These assignments come in three forms:

- **Laboratory exercises** are short, relatively simple exercises designed to introduce a new topic. Most weeks you'll do these during the scheduled lab meeting following Monday and Wednesday's lecture. Labs are worth a small but nonnegligible number of points, and are typically checked off by Jeff during the lab period.
- **Homework assignments** consist mostly of more difficult problems designed to solidify your understanding of the course material; you'll do these whenever you can schedule time, either in the lab or at home. You may be accustomed to homeworks with huge numbers of boring, repetitive exercises. You won't find that in here! Each assigned exercise teaches an important point.

  There are two homework assignments per week, but both are due on the Sunday after they are assigned. These assignments are included in the course reader and the course homepage. You are encouraged to *discuss* the homework with other students. Specific Homework requirements and grading policies are below.
- **Projects** are larger assignments intended both to teach you the skill of developing a large program and to assess your understanding of the course material. There are four projects during the term, and you'll work on some of them in groups. Specific project requirements and grading policies are listed below.

Everything you turn in for grading must show your name(s) and your computer account login(s)! Please cooperate about this; make sure they're visible on the *top* of the files you turn in, not buried somewhere in a comment or a function.

9 Testing and Grading

The grading policy of the course has these three goals: it should provide a reasonably accurate measure of your understanding of the material; it should minimize competitiveness and grade pressure, so that you can focus instead on the intellectual content of the course; and it should minimize the time I spend arguing with you about your grades. To meet these goals, your course grade is computed using a point system with a total of 300 points:

16	labs	@	1 point each	=	16 pts
15	homeworks	@	4 points each	=	60 pts
2	mini projects	@	7 points each	=	14 pts
2	larger projects	@	10 points each	=	20 pts
3	midterms	@	40 points each	=	120 pts
1	final			=	70 pts
--					---
39	assignments				300 pts

There will be three midterms (set for the end of the third, fifth, and seventh weeks of the term) and a final. The exams will be open book, open notes. (You may not use a computer during the exam.) In the past, some students have worried about time pressure, so we'll hold the midterms on Fridays 'round Noon (Room TBA) instead of during the lecture hour. My goal will be to write one-hour tests, but you'll have at least two hours to work on them. The relatively large number of midterms is meant to help you learn to take tests, and to reduce your anxiety about ruining your grade by having a bad day. In this course, the later topics depend on the early ones, so you must not forget things after each test is over!

Each letter grade corresponds to a range of point scores: 270 points and up is an A, 260–269 is A-, and so on by steps of ten points to 170–179 points for a D−.

		A	270-300	A-	260-269
B+	250-259	B	240-249	B-	230-239
C+	220-229	C	210-219	C-	200-209
D+	190-199	D	180-189	D-	170-179

This grading formula implies that **there is no curve**; your grade will depend only on how well you (and, to a small extent, your partner) do, and not on how well everyone else does. (If everyone does exceptionally badly on some exam, I may decide the exam was at fault rather than the students, in which case I'll adjust the grade cutoffs as I deem appropriate. But I won't adjust in the other direction; if everyone gets an A, that's great.)

Extra credit: Extra credit will be granted at the sole election of the staff and is typically reserved for persons who make a substantical and interesting (in the conceptual sense) addition to an assignment. I will mention opportunities for extra credit in class.

Exam regrading: If you believe we have misgraded an exam, there will be a regrading policy posted on the course website. At the very least, your entire exam will be regraded, so be sure that your score will really improve through this regrading! By University policy, final exams may *not* be regraded; to make up for this, we will grade every final exam twice. Final exams may be viewed at times and places to be announced.

Incomplete grades will be granted only for dire medical or personal emergencies that cause you to miss the final, and only if your work up to that point has been satisfactory.

10 Homework and Project Policies and Grading

In contrast to prior semesters, homework in this course will be done independently. You and your friends are encouraged to discuss the problems among yourselves, but the work that you turn in must be written and tested by you alone. Both of each week's homework assignments are due at 8:00 PM on the following Sunday. Both **homework sets must be submitted electronically** unless otherwise noted.

The purpose of the homework is for you to learn the course, not to prove that you already know it. Therefore, although the weekly homeworks will graded on correctness, you will be afforded an opportunity to recover points by improving your understanding of the material. **If you receive less than 90/100 credit on a particular homework, you can sign up for a face-to-face session with your reader.** During this session, you will have an opportunity to convince your reader that your understanding of the material has improved. If you show sufficient improvement, the reader may adjust your score upwards. Sign-up sheets for the face-to-face sessions will be posted in the laboratory (and perhaps online). **Please bring a paper copy of your homework to the sessions!**

The four programming projects are graded on correctness and style. The first two projects are to be done individually, and the last two in groups of exactly two. The last two projects are larger, and your group will work on a single solution, but the problems within each project are divided into two sets, and each of you will work on one set.

The latter two projects will probably include face-to-face grading with your reader. The reader will ask questions of each member of your group, and you will be graded by ALL of the group's members' ability to answer correctly. Therefore, you must work together to ensure that your partner understands the entire project.

Your group will turn in *one copy* of each project, with both of your names and logins listed on it. **The programming projects must be turned in online as well as in the homework box;** the deadline is usually 11:59 PM on the second Tuesday after it is assigned (i.e. you have two weeks for each project), but there will be some exceptions. You'll get further instructions about this when the time comes.

Online turnin: You must create a directory (you'll learn how to do that in lab) with the official assignment name, which will be something like `hw3` or `proj1`. Put in that directory all the files that you want to turn in. Then, while still in that directory, give the shell command `submit hw5` (or whatever the assignment name is). We'll give more details in the lab.

Paper turnin: There are boxes with slots labelled by course in room 283 Soda Hall. (Don't put them in my mailbox or on my office door!) What you turn in should include transcripts showing that you have tested your solution as appropriate.

11 Collaborative Learning Policies and Cheating

We encourage collaboration. It is the best way to learn and keep up with the wealth of material you are expected to cover. At the same time, cheating is not permitted. Sometimes the line between collaboration and cheating doesn't seem so easy to articulate, so we've tried to come up with very clear and enforceable rules so that you know what is expected and aren't uncomfortable collaborating, and, at the same time, so that those who break the rules can be held accountable.

Unlinke the degree of collaboration allowed and expected on homeworks and labs, the tests in this course must be your own, individual work. I hope that you will work cooperatively with your friends *before* the test to help each other prepare by learning the ideas and skills in the course. But during the test you're on your own. The EECS Department Policy on Academic Dishonesty says, "Copying all or part of another person's work, or using reference materials not specifically allowed, are forms of cheating and will not be tolerated." (61A tests are open-book, so reference materials are okay.) The policy statement goes on to explain the penalties for cheating, which range from a zero grade for the test up to dismissal from the University, for a second offense.

For the programming projects, copying others' work, whether from your friend who took the course last semester or from other current students in other groups is cheating. If you don't know how to do something, it's better to leave it out than to copy someone else's work. If you do learn something from someone else, and understand it now, then cite it as theirs. But be prepared to back up that you understand it without them around. If you do not cite it, it is considered plagiarism, and is again, cheating.

It is highly unlikely that different people would arrive at the exact same solutions on their own. We do have programs to test for code similarity – and these programs are smart enough to know when only the variable names have been changed. Don't cheat–you do a disservice to yourself, to those you copy from, and ultimately, to the whole course as time is taken away from preparing lectures and answering questions to deal with cheaters.

For the homework assignments, before you develop your solutions to the problems you are encouraged to discuss it with other students, in groups as large or small as you like. **When you turn in solutions, you must give credit to any other student(s) who contributed to your work.** This does not mean e.g.

16 of you should turn in precisely the same work. It means that you may talk about it, work it out, try it, and then each person writes it up on their own. Working on the homework in groups is both a good way to learn and a lot more fun! If you take the opportunity to discuss the homework with other students then you'll probably solve every problem correctly.

In my experience, most students who cheat do so because they fall behind gradually, and then panic at the last minute. Some students get into this situation because they are afraid of an unpleasant conversation with an instructor if they admit to not understanding something. I would much rather deal with your misunderstanding *early* than deal with its consequences later. Even if the problem is that you spent the weekend stoned out of your skull instead of doing your homework, please overcome your feelings of guilt and ask for help as soon as you need it.

If you are still unclear on the cheating policy, ask yourself this: in all of your talking with other students, did you UNDERSTAND the solution, or did you merely write down what someone else told you? If you didn't understand, that you aren't doing the work yourself– not honestly. Again, it is better to have the answer wrong, or only partially right than to rely on someone else's answer. (Often because they too could be wrong!)

Working cooperatively in groups is a change from the traditional approach in schools, in which students work either in isolation or in competition. But cooperative learning has become increasingly popular as educational research has demonstrated its effectiveness. One advantage of cooperative learning is that it allows us to give intense assignments, from which you'll learn a great deal, while limiting the workload for each individual student. Another advantage, of course, is that it helps you to understand new ideas when you discuss them with other people. Even if you are the "smartest" person in your group, you'll find that you learn a lot by discussing the course with other students. For example, in the past some of our best students have commented that they didn't *really* understand the course until they worked as lab assistants and had to explain the ideas to later students.

If some medical or personal emergency takes you away from the course for an extended period, or if you decide to drop the course for any reason, please don't just disappear silently! You should inform the other members of your group, and your TA, so that nobody is depending on you to do something you can't finish.

Penalties for cheating: Generally, the penalty for cheating on any assignment will be, at the very least, a zero on the assignment and will result in a notice being sent to the Office of Student Conduct. Further offenses and particularly egregious forms of cheating (like selling answers) will be dealt with more severely.

12 Lateness

A programming project that is not ready by the deadline may be turned in until 24 hours after the due date. These late projects will count for 2/3 of the earned score. No credit will be given for late homeworks, late labs, or for projects turned in after 24 hours. Please do not beg and plead for exceptions. If some personal crisis disrupts your schedule one week, don't waste your time and ours by trying to fake it; just be sure you do the next week's work on time.

By the way, if you wait until the night before to do the homework or a project, you will probably experience some or all of the following: a shortage of available workstations, an unusually slow computer response, or a file server crash.

13 Lost and Found

When people bring me found items from lecture or lab, I take them to the Computer Science office, 387 Soda. Another place to check for lost items is the campus police office in Sproul Hall.

14 Questions and Answers

Q: Is it true that 61A is the weed-out course for wannabe CS majors?

A: No. The lower division sequence as a whole does determine admission to the major, but no one course is crucial. More to the point, the work in all of these courses is *not* designed to be especially hard; the upper division courses are much harder. The grading policy in 61A is not harsh and is *not curved* as it would be if we had weeding out in mind. However, you may take this course as an opportunity to weed *yourself* out; if you find that you don't enjoy the work, perhaps you aren't a computer scientist at heart.

Q: Why don't we learn some practical language like C++?

A: Firstly, Lisp *is* practical. Of the hundreds of languages that have been invented, Lisp is the second-oldest survivor, after Fortran. It hasn't lasted 35 years by being useless. Secondly, and more importantly, the goal of 61A isn't to teach you a language. The language is just the medium for the ideas in the course, and Lisp gets in the way less than most languages because it has very little syntax and because you don't have to worry about what's where in the computer memory. (Next semester you'll learn Java.) Finally, our textbook is **the best computer science book ever written**. It happens to use Lisp; if they'd used COBOL, we'd probably teach COBOL for the sake of this text.

Q: What's your advice on surviving this course?

A: Two things: Don't leave the homework and projects until the last minute, and **ask for help as soon as you don't understand something.**

Q: I am disabled and need special facilities or arrangements to do the course work. What should I do about it?

A: If you need special arrangements about class attendance, taking tests, etc., I'll be glad to accommodate you; please take the initiative about letting me know what you need. For example, if you want to take tests separately, that's fine, as long as you ensure that we've worked out the arrangements before the test. The Disabled Students Program (ext. 2-0518) has voice response terminals from which blind students can connect to our computers. **If English is not your native language,** and you have trouble understanding the course materials or lectures for that reason, please ask for help about that too.

Q: I don't like (or have a conflict with) my pre-assigned discussion section. Can I switch?

A: You must negotiate this with Jeff.

Q: What should we call you?

A: ”Kurt” is just fine.

Q: I'm having trouble understanding the assignments. I've never had a problem like this in school before. Does this mean I'm not as good a programmer as I thought, or should I just wait a week or two and see if things clear up?

A: Neither. THIS COURSE IS CHALLENGING! In some ways, it might be the most challenging CS course you EVER take as an undergraduate. Most Berkeley students found high school pretty easy, and for many of you, this course will be the first real intellectual challenge you've met. You may have come to believe that everything should be easy for you. On the contrary; if you find your courses easy, you're taking the wrong courses! The whole reason you chose an excellent university was to stretch your mind. (If you chose Berkeley for the sake of a prestigious diploma, maybe you should consider majoring in Business Administration.) *There is nothing shameful about asking for help.* You will learn a lot even if you do not get an A+. Every semester a few intelligent students end up in trouble in this course because they're too proud to come to office hours with questions. If you wait two weeks before you ask your question, by then you'll feel hopelessly behind, because the topics for those two weeks depend on the idea that you don't understand now.

Q: I have no prior programming experience, unlike those who have taken CS 3 that you regularly mention. Am I at a disadvantage to those students in terms of workload, grades, etc.?

A: Well, for the first couple weeks, youre definitely at a disadvantage. The cs3 students have already spent an entire semester learning scheme, higher-order procs, lambdas, recursion, and abstraction there is no reason why any of them should get less than perfect scores on any assignment from the first couple weeks. So, you will probably be spending more time and effort than they will for the first couple weeks and your grades over the first few assignments still (probably) wont be as good as theirs.

Fortunately, the class is not curved. It doesn't matter how well the cs3 students do; you need only be concerned with yourself. Many persons who have not taken cs3 get As in 61a. I havent seen the numbers myself, but I have heard that, statistically speaking, there is no difference between the average final grades of cs3 and non-cs3 students.

Q: I'm completely lost; I feel very awkward using scheme (I like my c++ much better) and I'm thinking about dropping the course. What do you think?

A: It's almost ironic that scheme is often harder to learn for people who have prior programming experience in other languages than for those who have never programmed before. Scheme requires a different way of thinking about problems and this can work against people who have had another, different sense of programming *per se* ingrained in them from the use of other languages.

Once you have become accustomed to it, however, you will begin thinking about problems in scheme-terms and feeling awkward coding in anything else. By the end of the course, scheme will be a tool that you use without even thinking about it (like writing with a pen). (Heidegger, anyone?)

How quickly you overcome your initial awkwardness with scheme is up to you: the more you play around with it, the faster you will become proficient. This class is really about thinking logically; if you are rational, reasonably intelligent, and willing to work very hard at absorbing new concepts, you will do very well in the course. If you fail to satisfy any of the three (especially the last), you will have a hard time.

If you do decide to stick it out, please be aware that the TAs and I are happy to help anyone who tries to help himself or herself. Dont be afraid to schedule office hours, etc. were here for you. Also, you may want to look into the recommended text 'Simply Scheme' by Brian Harvey. It is the book used in cs3.

15 First Assignments

Read section 1.1 of Abelson and Sussman as soon as possible. By Wednesday, read 1.3 of Abelson and Sussman. The first homework assignment is due next Sunday (check the reader or web site). You must log into your class account by Wednesday.

CS61A – Lab Assignment 1.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Don’t panic if you don’t finish everything today; you will have two labs sessions to finish this first lab.

0. Login to your user account and change your password – a sample interaction is shown below. Be aware that it may take several minutes for your new password to be recognized by all the machines.

```
nova[1] ~ > ssh po
po[1] ~ > passwd                  [NOTE: this is NOT "password"]
< ... Do some password stuff ... >
po[2] ~ > exit                    [NOTE: make sure you do this!]
nova[2] ~ >                       [We’re back! Move on to the next exercise...]
```

1. Set up the newsgroup. Details are in *A Quick Introduction to Using CS61A Computing Resources.*

2. Read *Basic Emacs Guide for CS 61A Students.* Now, start the Emacs editor, either by typing `emacs` in your main window or by selecting it from the right-mouse-button menu. (Your TA will show you how to do this.)

3. Start Scheme, either by typing `stk` in your main window or by typing meta-S in your Emacs window. Type each of the following expressions into Scheme, ending the line with the Enter (carriage return) key. **Think about the results!** Try to understand how Scheme interprets what you type.

```
3                                  (first ’hello)
(+ 2 3)                            (first hello)
(+ 5 6 7 8)                        (first (bf ’hello))
(+)                                (+ (first 23) (last 45))
(sqrt 16)                          (define pi 3.14159)
(+ (* 3 4) 5)                      pi
+                                  ’pi
’+                                 (+ pi 7)
’hello                             (* pi pi)
’(+ 2 3)                           (define (square x) (* x x))
’(good morning)                    (square 5)
(first 274)                        (square (+ 2 3))
(butfirst 274)
```

4. Use Emacs to create a file called `pigl.scm` in your directory containing the Pig Latin program shown below. Make sure to save this file before proceeding to the next exercise.

```
(define (pigl wd)                            (define (pl-done? wd)
  (if (pl-done? wd)                            (vowel? (first wd)))
      (word wd ’ay)
      (pigl (word (bf wd) (first wd)))))     (define (vowel? letter)
                                               (member? letter ’(a e i o u)))
```

5. Now run Scheme. You are going to create a transcript of a session using the file you just created, like this:

```
(transcript-on "lab1")       ; This starts the transcript file.
(load "pigl.scm")            ; This reads in the file you created earlier.
(pigl ’scheme)               ; Try out your program.
                             ; Feel free to try more test cases here!
(trace pigl)                 ; This is a debugging aid. Watch what happens
(pigl ’scheme)               ; when you run a traced procedure.
(transcript-off)
(exit)
```

Continued on next page.

Lab Assignment 1.1 continued...

6. Use `lpr` to print your transcript file – a sample interaction is shown below.

```
nova[1] ~ > lpr lab1
```

7. Predict what Scheme will print in response to each of these expressions. *Then* try it and make sure your answer was correct, or if not, that you understand why!

```
(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))

((if (< a b) + -) a b)
```

8. In the shell, type the command

```
cp ~cs61a/lib/plural.scm .
```

(Note the period at the end of the line!) This will copy a file from the class library to your own directory. Then, using emacs to edit the file, modify the procedure so that it correctly handles cases like `(plural ’boy)`.

9. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

10. Write a procedure `dupls-removed` that, given a sentence as input, returns the result of removing duplicate words from the sentence. It should work this way:

```
> (dupls-removed ’(a b c a e d e b))
(c a d e b)
> (dupls-removed ’(a b c))
(a b c)
> (dupls-removed ’(a a a a b a a))
(b a)
```

CS61A – Lab Assignment 1.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. For each expression, give a definition of `f` such that evaluating the expression will not cause an error, and say what the expression's value will be, given your definition.

```
a. f              d. ((f))
b. (f)            e. (((f)) 3)
c. (f 3)
```

2. Find the values of the expressions

```
a. ((t 1+) 0)            b. ((t (t 1+)) 0)           c. (((t t) 1+) 0)
```

where `1+` is a primitive procedure that adds 1 to its argument, and `t` is defined as follows:

```
(define (t f)
  (lambda (x) (f (f (f x)))) )
```

Work this out yourself before you try it on the computer!

3. Find the values of the expressions

```
a. ((t s) 0)             b. ((t (t s)) 0)            c.  (((t t) s) 0)
```

where `t` is defined as in question 2 above, and `s` is defined as follows:

```
(define (s x)
  (+ 1 x))
```

4. Write a procedure `substitute` that takes three arguments: a *new* word, an *old* word, and a sentence. It should return a copy of the sentence, but with every occurrence of the old word replaced by the new word.

```
> (substitute ’maybe ’yeah ’(she loves you yeah yeah yeah))
(she loves you maybe maybe maybe)
```

5. First, type the definitions

```
(define a 7)
(define b 6)
```

into Scheme. Then, fill in the blank in the code below with an expression whose value depends on both `a` and `b` to determine a return value of 24. Verify in Scheme that the desired value is obtained.

```
(let
  ((a 3) (b (+ a 2)))
  _______________ )
```

6. Write and test the `make-tester` procedure. Given a word `w` as argument, `make-tester` returns a procedure of one argument `x` that returns true if `x` is equal to `w` and false otherwise. Examples:

```
> ((make-tester ’hal) ’hal)
#t
> ((make-tester ’hal) ’cs61a)
#f
> (define sicp-author-and-astronomer? (make-tester ’gerry))
> (sicp-author-and-astronomer? ’hal)
#f
> (sicp-author-and-astronomer? ’gerry)
#t
```

CS61A – Lab Assignment 2.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

This lab exercise concerns the change counting program on pages 40–41 of Abelson and Sussman.

1. Identify two ways to change the program to *reverse* the order in which coins are tried, that is, to change the program so that pennies are tried first, then nickels, then dimes, and so on.

2. Abelson and Sussman claim that this change would not affect the *correctness* of the computation. However, it does affect the *efficiency* of the computation. Implement one of the ways you devised in exercise 1 for reversing the order in which coins are tried, and determine the extent to which the number of calls to cc is affected by the revision. Verify your answer on the computer, and provide an explanation. Hint: limit yourself to nickels and pennies, and compare the trees resulting from `(cc 5 2)` for each order.

3. Modify the `cc` procedure so that its `kinds-of-coins` parameter, instead of being an integer, is a *sentence* that contains the values of the coins to be used in making change. The coins should be tried in the sequence they appear in the sentence. For the `count-change` procedure to work the same in the revised program as in the original, it should call `cc` as follows:

```
(define (count-change amount)
  (cc amount '(50 25 10 5 1)) )
```

4. Many Scheme procedures require a certain type of argument. For example, the arithmetic procedures only work if given numeric arguments. If given a non-number, an error results.

Suppose we want to write *safe* versions of procedures, that can check if the argument is okay, and either call the underlying procedure or return `#f` for a bad argument instead of giving an error. (We'll restrict our attention to procedures that take a single argument.)

```
> (sqrt 'hello)
ERROR: magnitude: Wrong type in arg1 hello
> (type-check sqrt number? 'hello)
#f
> (type-check sqrt number? 4)
2
```

Write `type-check`. Its arguments are a function, a type-checking predicate that returns `#t` if and only if the datum is a legal argument to the function, and the datum.

5. We really don't want to have to use `type-check` explicitly every time. Instead, we'd like to be able to use a `safe-sqrt` procedure:

```
> (safe-sqrt 'hello)
#f
> (safe-sqrt 4)
2
```

Don't write `safe-sqrt`! Instead, write a procedure `make-safe` that you can use this way:

```
> (define safe-sqrt (make-safe sqrt number?))
```

It should take two arguments, a function and a type-checking predicate, and return a new function that returns `#f` if its argument doesn't satisfy the predicate.

CS61A – Lab Assignment 2.2 **Kurt Meinz**

University of California, Berkeley Summer 2005

1. Try these in Scheme:

```
(define x (cons 4 5))
(car x)
(cdr x)
```

```
(define y (cons ’hello ’goodbye))
(define z (cons x y))
(car (cdr z))
(cdr (cdr z))
```

2. Predict the result of each of these before you try it:

```
(cdr (car z))

(car (cons 8 3))

(car z)

(car 3)
```

3. Enter these definitions into Scheme:

```
(define (make-rational num den)
  (cons num den))

(define (numerator rat)
  (car rat))

(define (denominator rat)
  (cdr rat))

(define (*rat a b)
  (make-rational (* (numerator a) (numerator b))
                 (* (denominator a) (denominator b))))

(define (print-rat rat)
  (word (numerator rat) ’/ (denominator rat)))
```

4. Try this:

```
(print-rat (make-rational 2 3))

(print-rat (*rat (make-rational 2 3) (make-rational 1 4)))
```

5. Define a procedure `+rat` to add two rational numbers, in the same style as `*rat` above.

6. Suppose the constructor for rational numbers was changed to

```
(define (make-rational num den)
  (sentence num den))
```

Rewrite the rest of the functions in exercise 3 such that it preserves the behavior of exercises 4 and 5.

7. *SICP* ex. 2.4

8. *SICP* ex. 2.18; this should take some thought, and you should make sure you get it right, but don’t get stuck on it for the whole hour. **Note:** Your solution should reverse *lists*, not sentences! That is, you should be using `cons`, `list`, and `append`, not `first`, `butfirst`, `sentence`, etc.

CS61A – Lab Assignment 3.1 **Kurt Meinz**

University of California, Berkeley Summer 2005

1. *SICP* ex. 2.25 and 2.53; these should be quick and easy.

2. *SICP* ex. 2.55; **explain your answer to your TA.**

3. *SICP* ex. 2.27. This is the central exciting adventure of today's lab! Think hard about it.

4. Each person individually make up a procedure named `mystery` that, given two lists as arguments, returns the result of applying *exactly two* of `cons`, `append`, or `list` to `mystery`'s arguments, using no quoted values or other procedure calls. Here are some examples of what is and is not fair game:

```
okay                                 not okay

(define (mystery L1 L2)              (define (mystery L1 L2)
  (cons L1 (append L2 L1)))            (cons L1 (cons L2 (cons L1 L2))))

(define (mystery L1 L2)              (define (mystery L1 L2)
  (list L1 (list L1 L1)))              (cons L1 L2))

(define (mystery L1 L2)              (define (mystery L1 L2)
  (append (cons L2 L2) L1))            (append L1 (cons L1 '(A B C))))
```

Type your `mystery` definition into a file, and have one of your partners load it into Scheme and try to guess what it is by trying it out with various arguments.

After everyone has tried someone else's procedure, decide with your partners which procedure was hardest to guess and why, and what test cases were most and least helpful in revealing the definitions.

CS61A – Lab Assignment 3.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

Start by reading *SICP* section 2.3.3 (pages 151–161).

1. *SICP* ex. 2.62.

2. The file `~cs61a/lib/bst.scm` contains the binary search tree procedures from pages 156–157 of *SICP*. Using `adjoin-set`, construct the trees shown on page 156.

3. *SICP* ex. 2.74.

CS61A – Lab Assignment 4.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. Modify the `person` class given in the lecture notes for week 3 (it’s in the file `demo2.scm` in the `~cs61a/lectures/3.0` directory) to add a `repeat` method, which repeats the last thing said. Here’s an example of responses to the `repeat` message.

```
> (define brian (instantiate person ’brian))
brian
> (ask brian ’repeat)
()
> (ask brian ’say ’(hello))
(hello)
> (ask brian ’repeat)
(hello)
> (ask brian ’greet)
(hello my name is brian)
> (ask brian ’repeat)
(hello my name is brian)
> (ask brian ’ask ’(close the door))
(would you please close the door)
> (ask brian ’repeat)
(would you please close the door)
```

2. This exercise introduces you to the `usual` procedure described on page 9 of “Object-oriented Programming – Above-the-line View”. Read about `usual` there to prepare for lab. Suppose that we want to define a class called `double-talker` to represent people that always say things twice, for example as in the following dialog.

```
> (define mike (instantiate double-talker ’mike))
mike
> (ask mike ’say ’(hello))
(hello hello)
> (ask mike ’say ’(the sky is falling))
(the sky is falling the sky is falling)
```

Consider the following three definitions for the `double-talker` class. (They can be found online in the file `~cs61a/lib/double-talker.scm`.)

```
(define-class (double-talker name)
  (parent (person name))
  (method (say stuff) (se (usual ’say stuff) (ask self ’repeat))) )

(define-class (double-talker name)
  (parent (person name))
  (method (say stuff) (se stuff stuff)) )

(define-class (double-talker name)
  (parent (person name))
  (method (say stuff) (usual ’say (se stuff stuff))) )
```

Determine which of these definitions work as intended. Determine also for which messages the three versions would respond differently.

CS61A – Lab Assignment 4.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. Given below is a simplified version of the `make-account` procedure on page 223 of Abelson and Sussman.

```
(define (make-account balance)
  (define (withdraw amount)
    (set! balance (- balance amount)) balance)
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch msg)
    (cond
      ((eq? msg 'withdraw) withdraw)
      ((eq? msg 'deposit) deposit) ) )
  dispatch)
```

Fill in the blank in the following code so that the result works exactly the same as the `make-account` procedure above, that is, responds to the same messages and produces the same return values. The differences between the two procedures are that the inside of `make-account` above is enclosed in the `let` below, and the names of the parameter to `make-account` are different.

```
(define (make-account init-amount)
  (let ( ______________ )
    (define (withdraw amount)
      (set! balance (- balance amount)) balance)
    (define (deposit amount)
      (set! balance (+ balance amount)) balance)
    (define (dispatch msg)
      (cond
        ((eq? msg 'withdraw) withdraw)
        ((eq? msg 'deposit) deposit) ) )
    dispatch) )
```

2. Modify either version of `make-account` so that, given the message `balance`, it returns the current account balance, and given the message `init-balance`, it returns the amount with which the account was initially created. For example:

```
> (define acc (make-account 100)
acc
> (acc 'balance)
100
```

Continued on next page...

Lab Assignment 4.2 continued:

3. Modify `make-account` so that, given the message `transactions`, it returns a list of all transactions made since the account was opened. For example:

```
> (define acc (make-account 100))
acc
> ((acc 'withdraw) 50)
50
> ((acc 'deposit) 10)
60
> (acc 'transactions)
((withdraw 50) (deposit 10))
```

4. Given this definition:

```
(define (plus1 var)
  (set! var (+ var 1))
  var)
```

Show the result of computing

```
(plus1 5)
```

using the substitution model. That is, show the expression that results from substituting `5` for `var` in the body of `plus1`, and then compute the value of the resulting expression. What is the actual result from Scheme?

Continued on next page...

Lab Assignment 4.2 continued:

This lab activity consists of example programs for you to run in Scheme. Predict the result before you try each example. If you don’t understand what Scheme actually does, ask for help! Don’t waste your time by just typing this in without paying attention to the results.

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(make-adder 3)

((make-adder 3) 5)

(define (f x) (make-adder 3))

(f 5)

(define g (make-adder 3))

(g 5)

(define (make-funny-adder n)
  (lambda (x)
    (if (equal? x ’new)
        (set! n (+ n 1))
        (+ x n))))

(define h (make-funny-adder 3))

(define j (make-funny-adder 7))

(h 5)

(h 5)

(h ’new)

(h 5)

(j 5)

(let ((a 3))
  (+ 5 a))

(let ((a 3))
  (lambda (x) (+ x a)))

((let ((a 3))
   (lambda (x) (+ x a)))
 5)
```

```
((lambda (x)
   (let ((a 3))
     (+ x a)))
 5)

(define k
  (let ((a 3))
    (lambda (x) (+ x a))))

(k 5)

(define m
  (lambda (x)
    (let ((a 3))
      (+ x a))))

(m 5)

(define p
  (let ((a 3))
    (lambda (x)
      (if (equal? x ’new)
          (set! a (+ a 1))
          (+ x a)))))

(p 5)

(p 5)

(p ’new)

(p 5)

(define r
  (lambda (x)
    (let ((a 3))
      (if (equal? x ’new)
          (set! a (+ a 1))
          (+ x a)))))

(r 5)

(r 5)

(r ’new)

(r 5)
```

Continued on next page...

Lab Assignment 4.2 continued:

```
(define s
  (let ((a 3))
    (lambda (msg)
      (cond ((equal? msg 'new)
             (lambda ()
               (set! a (+ a 1))))
            ((equal? msg 'add)
             (lambda (x) (+ x a)))
            (else (error "huh?"))))))

(s 'add)

(s 'add 5)

((s 'add) 5)

(s 'new)

((s 'add) 5)

((s 'new))

((s 'add) 5)
```

```
(define (ask obj msg . args)
  (apply (obj msg) args)))

(ask s 'add 5)

(ask s 'new)

(ask s 'add 5)

(define x 5)

(let ((x 10)
      (f (lambda (y) (+ x y))))
  (f 7))

(define x 5)
```

CS61A – Lab Assignment 5.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. Exercise 3.12 of Abelson and Sussman.

2. Suppose that the following definitions have been provided.

`(define x (cons 1 3))` `(define y 2)` A CS 61A student, intending to change the value of `x` to a pair with `car` equal to 1 and `cdr` equal to 2, types the expression `(set! (cdr x) y)` instead of `(set-cdr! x y)` and gets an error. Explain why.

3a. Provide the arguments for the two `set-cdr!` operations in the blanks below to produce the indicated effect on `list1` and `list2`. Do not create any new pairs; just rearrange the pointers to the existing ones.

```
> (define list1 (list (list 'a) 'b))
list1
> (define list2 (list (list 'x) 'y))
list2
> (set-cdr! _______________ _______________ )
okay
> (set-cdr! _______________ _______________ )
okay
> list1
((a x b) b)
> list2
((x b) y)
```

3b. After filling in the blanks in the code above and producing the specified effect on `list1` and `list2`, draw a box-and-pointer diagram that explains the effect of evaluating the expression `(set-car! (cdr list1) (cadr list2))` .

4. Exercises 3.13 and 3.14 in Abelson and Sussman.

CS61A – Lab Assignment 5.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. What is the type of the value of `(delay (+ 1 27))`? What is the type of the value of `(force (delay (+ 1 27)))`?

2. Evaluation of the expression

```
(stream-cdr (stream-cdr (cons-stream 1 '(2 3))))
```

produces an error. Why?

3. Consider the following two procedures.

```
(define (enumerate-interval low high)
  (if (> low high)
    '()
    (cons low (enumerate-interval (+ low 1) high)) ) )

(define (stream-enumerate-interval low high)
  (if (> low high)
    the-empty-stream
    (cons-stream low (stream-enumerate-interval (+ low 1) high)) ) )
```

What's the difference between the following two expressions?

```
(delay (enumerate-interval 1 3))
(stream-enumerate-interval 1 3)
```

4. An unsolved problem in number theory concerns the following algorithm for creating a sequence of positive integers $s_1, s_2, ...$

```
Choose $s_1$ to be some positive integer.
For $n > 1$,
    if $s_n$ is odd, then $s_{n+1}$ is $3 s_n + 1$;
    if $s_n$ is even, then $s_{n+1}$ is $s_n / 2$.
```

No matter what starting value is chosen, the sequence always seems to end with the values 1, 4, 2, 1, 4, 2, 1, ... However, it is not known if this is always the case.

4a. Write a procedure `num-seq` that, given a positive integer `n` as argument, returns the stream of values produced for `n` by the algorithm just given. For example, `(num-seq 7)` should return the stream representing the sequence 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

4b. Write a procedure `seq-length` that, given a stream produced by `num-seq`, returns the number of values that occur in the sequence up to and including the first 1. For example, `(seq-length (num-seq 7))` should return 17. You should assume that there is a 1 somewhere in the sequence.

CS61A – Lab Assignment 6.1 — **Kurt Meinz**
University of California, Berkeley — Summer 2005

1. List all the procedures in the metacircular evaluator that call `mc-eval`.

2. List all the procedures in the metacircular evaluator that call `mc-apply`.

3. Abelson and Sussman, exercise 4.1

4. In this exercise, we will begin to write a postfix version of the metacircular evaluator. For now, do not worry about special forms; we will only provide postfix support for procedure calls, self-evaluating expressions, etc.

a. Make a new copy of `mceval.scm` and call it `postfix-mceval.scm` (we don't want these changes to conflict with future exercises). Make all changes for this exercise in `postfix-mceval.scm`.

b. Our postfix expressions will look the same as our prefix expressions, except that the operator will go on the right. Examples:

```
(2 3 +)
(((3 1 +) 8 *) 9 -)
('(one no trump) car)
```

To make this somewhat efficient (by only looking at each subexpression once), we will maintain a `stack` in `mc-eval`. The purpose of this stack is to keep track of evaluated subexpressions until we get to the operator.

Example: We want to evaluate `(2 (3 4 +) +)`

```
-> mc-eval called with exp = (2 (3 4 +) +)  stack = ()  ;; not at operator yet
   -> mc-eval called with exp = 2  stack = ()           ;; .. so evaluate first arg
-> mc-eval called with exp = ((3 4 +) +)  stack = (2)   ;; not at operator yet -> evaluate 2nd arg
   -> mc-eval called with exp = (3 4 +)  stack = ()     ;; .. second arg is a nested exp
      -> mc-eval called with exp = 3 stack = ()         ;; .... evaluate 1st arg of nested exp
   -> mc-eval called with exp = (4 +)  stack = (3)      ;; .. not at operator yet
      -> mc-eval called with exp = 4 stack = ()         ;; .... evaluate 2nd arg of nested exp
   -> mc-eval called with exp = (+)  stack = (3 4)      ;; .. found the operator
      -> mc-eval called with exp = + stack = ()         ;; .... so evaluate op and perform operation
-> mc-eval called with exp = (+)  stack = (2 7)         ;; found the operator
                                                        ;; .. so evaluate op and perform operation
```

As you can see, everytime we start evaluating a new expression, we start with an empty stack. At the top of `mceval.scm`, give a definition of `the-empty-stack`, which will be used to represent such an empty stack.

c. Modify the code for `mc-eval` so that it now takes in an additional argument: the `stack`

d. Modify the `application?` clause of mc-eval so that it deals with postfix expressions and exhibits the behavior in the example in step b. Do not worry about data abstraction.

e. Modify the code so that all calls to mc-eval have an appropriate stack.

f. Try it out!

g. Given the current modifications, what types of expressions will not work in postfix notation. What additional modifications are necessary?

CS61A – Lab Assignment 6.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. Explain why `make-procedure` does *not* call `eval`.
2. In `setup-environment`, what is the purpose of `import`?
3. Abelson and Sussman, exercise 4.2
4. Abelson and Sussman, exercise 4.4
5. Abelson and Sussman, exercise 4.5

CS61A – Lab Assignment 7.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Part A: Abelson and Sussman, exercises 4.27 and 4.29.

Part B: In this lab exercise you will become familiar with the Logo programming language, for which you'll be writing an interpreter in project 4.

To begin, type `logo` at the Unix shell prompt — **not** from Scheme! You should see something like this:

```
Welcome to Berkeley Logo version 3.4
?
```

The question mark is the Logo prompt, like the `>` in Scheme. (Later, in some of the examples below, you'll see a `>` prompt from Logo, while in the middle of defining a procedure)

1. Type each of the following instruction lines and note the results. (A few of them will give error messages.) If you can't make sense of a result, ask for help.

```
print 2 + 3

print 2+3

print sum 2 3

print (sum 2 3 4 5)

print sum 2 3 4 5

2+3

print "yesterday

print "julia"

print revolution

print [blue jay way]

show [eight days a week]

show first [golden slumbers]

print first bf [she loves you]

pr first first bf [yellow submarine]

to second :stuff
output first bf :stuff
end
```

```
second "something

print second "piggies

pr second [another girl]

pr first second [carry that weight]
pr second second [i dig a pony]

to pr2nd :thing
print first bf :thing
end

pr2nd [the 1 after 909]

print first pr2nd [hey jude]

repeat 5 [print [this boy]]

if 3 = 1+1 [print [the fool on the hill]]

print ifelse 2=1+1  ~
      [second [your mother should know]] ~
      [first "help]

print ifelse 3 = 1 + 2 ~
      [strawberry fields forever] ~
      [penny lane]

print ifelse 4 = 1 + 2 ~
      ["flying] ~
      [[all you need is love]]
```

Continued on next page...

Lab Assignment 7.1 continued...

```
to greet :person
say [how are you,]
end

to say :saying
print sentence :saying :person
end

greet "ringo

show map "first [paperback writer]

show map [word first ? last ?] ~
         [lucy in the sky with diamonds]

to who :sent
foreach [pete roger john keith] "describe
end

to describe :person
print se :person :sent
end

who [sells out]

print :bass

make "bass "paul

print :bass

print bass

to bass
output [johnny cymbal]
end

print bass

print :bass

print "bass
```

```
to countdown :num
if :num=0 [print "blastoff stop]
print :num
countdown :num-1
end

countdown 5

to downup :word
print :word
if emptyp bl :word [stop]
downup bl :word
print :word
end

downup "rain

;;;; The following stuff will work
;;;; only on an X workstation:

cs

repeat 4 [forward 100 rt 90]

cs

repeat 10 [repeat 5 [fd 150 rt 144] rt 36]

cs repeat 36 [repeat 4 [fd 100 rt 90]
              setpc remainder pencolor+1 8
              rt 10]

to tree :size
if :size < 3 [stop]
fd :size/2
lt 30 tree :size*3/4 rt 30
fd :size/3
rt 45 tree :size*2/3 lt 45
fd :size/6
bk :size
end

cs pu bk 100 pd ht tree 100
```

2. Devise an example that demonstrates that Logo uses dynamic scope rather than lexical scope. Your example should involve the use of a variable that would have a different value if Logo used lexical scope. Test your code with Berkeley Logo.

3. Explain the differences and similarities among the Logo operators " (double-quote), `[ ]` (square brackets), and `:` (colon).

CS61A – Lab Assignment 7.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

1. Abelson and Sussman, exercises 4.35 and 4.38.

2. In this exercise we learn what a *continuation* is. Suppose we have the following definition:

```
(define (square x cont)
  (cont (* x x))
```

Here `x` is the number we want to square, and `cont` is the procedure to which we want to pass the result. Now try these experiments:

```
> (square 5 (lambda (x) x))

> (square 5 (lambda (x) (+ x 2)))

> (square 5 (lambda (x) (square x (lambda (x) x))))

> (square 5 display)

> (define foo 3)
> (square 5 (lambda (x) (set! foo x)))
> foo
```

Don't just type them in – make sure you understand why they work! The nondeterministic evaluator works by evalutating every expression with *two* continuations, one used if the computation succeeds, and one used if it fails.

```
(define (reciprocal x yes no)
  (if (= x 0)
      (no x)
      (yes (/ 1 x))))

> (reciprocal 3 (lambda (x) x) (lambda (x) (se x '(cannot reciprocate))))

> (reciprocal 0 (lambda (x) x) (lambda (x) (se x '(cannot reciprocate))))
```

CS61A – Lab Assignment 8.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Abelson and Sussman, exercises 4.55 and 4.62:

4.55: Give simple queries that retrieve the following information from the data base:

All people supervised by Ben Bitdiddle;

The names and jobs of all people in the accounting division;

The names and addresses of all people who live in Slumerville.

4.62: Define rules to implement the `last-pair` operation of exercise 2.17, which returns a list containing the last element of a nonempty list. Check your rules on queries such as

```
(last-pair (3) ?x)
(last-pair (1 2 3) ?x)
(last-pair (2 ?x) (3))
```

Do your rules work correctly on queries such as `(last-pair ?x (3))`?

For the lab exercises and the homework problems that involve writing queries or rules, test your solutions using the query system. To run the query system and load in the sample data:

```
scm
(load "~cs61a/lib/query.scm")
(initialize-data-base microshaft-data-base)
(query-driver-loop)
```

You're now in the query system's interpreter. To add an assertion:

```
(assert! (foo bar))
```

To add a rule:

```
(assert! (rule (foo) (bar)))
```

Anything else is a query.

CS61A – Homework 1.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Functional programming

Lectures: Monday June 23, Tuesday June 24

Reading: Abelson & Sussman, Section 1.1

In this assignment you'll write simple recursive programs to manipulate words and sentences, as well as explore what makes special forms special. You should use the functions `sentence`, `first`, `butfirst`, `last` and `butlast` presented in lecture to operate on words and sentences. These functions are not discussed in the book. If you have taken CS3 and know about higher-order procedures such as `every`, please do not use them; use explicit recursion.

This homework is due at **8 PM on Sunday, June 26**. Please put your answers into a file called `hw1-1.scm` and submit it electronically by typing `submit hw1-1` in the directory where the file is located. You will probably find it convenient to make a new directory (folder) for every week of the course and store the associated labs and homeworks in it; to create a folder called `week1` type `mkdir week1` at the Unix prompt. We understand that many of you have never used Unix before and will be struggling to find your way around. If you run into problems submitting the homework electronically don't freak out. We'll be quite lenient the first time around. Get your TA to help you submit on Monday. In subsequent weeks, we expect you to have mastered the online submission process.

Be sure to test each function you write; the sample calls given here do not guarantee your code is bug-free. Include your test cases in your submission, but make sure to comment them out (the semicolon character begins a one-line comment in Scheme) so the file loads smoothly. Unless explicitly disallowed, you may always write helper procedures.

One final note: Please ensure that your submitted `.scm` file loads into STk via the `(load "hw1-1.scm")` command smoothly. **Submissions that cause errors on loading may lose points.**

Question 1. Write a procedure `increment` that takes two arguments: a number n and a sentence of numbers. It should increment each number in the sentence by n and return a sentence of the results:

```
STk> (increment 5 (se 1 2 -5 10))
(6 7 0 15)
```

Question 2. Write a procedure `ends-vow` that takes a sentence as its argument and returns a sentence containing only those words of the argument whose last letter is a vowel (a, e, i, o, u):

```
STk> (ends-vow '(please put the salami above the blue elephant))
(please the salami above the blue)
STk> (ends-e '(absolutely nothing))
()
```

Question 3. Write a procedure `reverse` which reverses a sentence:

```
STk> (reverse '(the matrix cannot tell you who you are))
(are you who you tell cannot matrix the)
STk> (reverse '(kurt alex greg carolen))
(carolen greg alex kurt)
```

The adventure continues on the next page.

Question 4. Write a predicate `non-decreasing?` that takes a **non-empty** sentence of numbers as its argument. It should return a true value if the numbers are in non-decreasing order and a false value otherwise:

```
STk> (non-decreasing? '(1 4 8 17))
#t
STk> (non-decreasing? '(2 5 4))
#f
STk> (non-decreasing? '(17))
#t
STk> (non-decreasing? '(1 1))
#t
```

Question 5. This question concerns special forms.

A. Most versions of Lisp provide `and` and `or` procedures like the ones described on Page 19 of the book. In principle there is no reason why these can't be ordinary procedures, but some versions of Lisp make them special forms. Suppose we evaluate:

```
STk> (or (= x 0) (= y 0) (= z 0))
```

If `or` is an ordinary procedure, all three argument expressions will be evaluated when `or` is invoked. But if the variable `x` has the value `0`, we know that `or` should return true regardless of the values of `y` and `z`. There is no reason to evaluate the other two expressions! A Lisp interpreter in which `or` is a special form can evaluate the arguments one by one until either a true one is found or it runs out of arguments. (This is called *short-circuit* evaluation.)

Devise a test that will determine whether Scheme's `and` and `or` are a short-circuiting special forms or ordinary functions. That is, do `and` and `or` evaluate all their arguments all the time or do they stop as soon as they know the correct value to return?

B. Scheme has two special forms for making choices, `cond` and `if`. Is it possible to define one in terms of the other? Specifically, say we attempt to define our own `if` procedure:

```
STk> (define (my-if predicate consequent alternative)
        (cond (predicate consequent)
              (else alternative)))
```

Let's take it out for a spin:

```
STk> (my-if (= 5 6) 'yes 'no)
no
```

It seems to work, so try something more interesting:

```
STk> (define (my-factorial n)
        (my-if (= n 0)
               1
               (* n (my-factorial (- n 1)))))
```

What happens when you attempt to use `my-factorial`? Why?

CS61A – Homework 1.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Higher-order procedures

Lectures: Wednesday June 25, Thursday June 26

Reading: Abelson & Sussman, Section 1.3

In this assignment you'll gain experience with Scheme's first class procedures and the `lambda` special form for creating anonymous functions.

This homework is due at **8 PM on Sunday, June 26**. Please put your answers into a file called `hw1-2.scm` and submit electronically by typing `submit hw1-2` in the directory where the file is located.

The book's treatment of this subject is highly mathematical because it doesn't introduce symbolic data (such as words and sentences) until later. Don't panic if you have trouble with the half-interval example on Page 67; you can just skip it. Try to read and understand everything else.

Question 1. Use higher-order functions such as `every` and `keep` presented in lecture to write the function `permute`; don't use explicit recursion! `Permute` takes two arguments,a sentence and a word. The first sentence, called the *template*, contains only numbers. A number n in the template corresponds to the nth letter in the second argument to `permute` (counting from 1). `Permute` should rearrange its second argument to conform to the template:

```
STk> (permute '(1 1 2 1) 'hello)
hheh
STk> (permute '(3 2 1) 'chicken)
ihc
STk> (permute '() 'lalala)
()
```

Don't check for out-of-bounds numbers in the template. You may find `item` useful.

Question 2. This question builds on the `sum` procedure defined on Page 58.

A. The `sum` function allows one to add up the elements of a pattern defined by the parameters `term` and `next` over some range $[a, b]$. Use `sum` to define a function `sum-evens` that takes two numbers and returns the sum of all even numbers between them, inclusive: 88

```
STk> (sum-evens 1 10)
30                       ;; 2 + 4 + 6 + 8 + 10
STk> (sum-evens 4 9)
18                       ;; 4 + 6 + 8
STk> (sum-evens 8 8)
7                        ;; 8
```

Your definition of `sum-odds` must have the following form:

```
(define (sum-odds a b)
   (sum ?? ?? ?? ??))
```

You may assume the first argument to `sum-odds` will be less than or equal to the second.

The excitement continues on the next page.

B. What if we want to multiply numbers over a range? Define a function `product` that takes the same arguments as `sum` but does multiplication rather than addition:

```
STk> (sum (lambda (x) 10) 1 (lambda (x) (+ x 1)) 3)
30
STk> (product (lambda (x) 10) 1 (lambda (x) (+ x 1)) 3)
1000
```

C. The factorial of a number n is $1 \cdot 2 \cdot 3 \cdot ... \cdot n$. Use `product` to define a `factorial` function.

D. Now use `product` to approximate π using the formula:

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdot ...}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdot ...}$$

Do this by writing a function `pi` that takes one numeric argument i. This parameter should in some way control the number of terms computed; hence a larger value of i should yield a closer approximation to π. Exactly what is meant by "number of terms" is up to you. All we care about is that a larger value of i produces a better approximation. For example, our solution takes i to be the largest number in the numerator:

```
STk> (pi 1000)
3.1431607055322752
```

Depending on the meaning you give to i and the algorithm you employ, you might not get as close an approximation (or you might get an even closer one!). It is likely that making i too big will overload the machine, so don't be overeager.

One way to do this problem is to compute the numerator and denominator independently, then divide them. While this can be done, it's trickier than it looks because you have to ensure the same number of terms in both, and, as you can see, the numerator and denominator don't line up nicely. If you're stuck, try treating $\frac{2 \cdot 4}{3 \cdot 3}$ as one unit.

E. Writing `product` after `sum` should have seemed redundant. They differ in only two ways: the combiner function and the value returned in the base case (often called the "null value"). We'd like to generalize the pattern exhibited by both functions to create a still more powerful procedure called `accumulate`. This function should take all the arguments that `sum` and `product` do plus the two additional parameters: the combiner and the null value. Once you have written `accumulate` both `sum` and `product` may be defined in terms of it like this:

```
STk> (define (sum term a next b) (accumulate + 0 term a next b))
sum
STk> (define (product term a next b) (accumulate * 1 term a next b))
product
```

Use `accumulate` to define the function `enumerate-interval`, which takes two numeric arguments a and b, where $a \leq b$. It returns a sentence of all the numbers between a and b, inclusive:

```
STk> (enumerate-interval 10 3)
(10 9 8 7 6 5 4 3)
STk> (enumerate-interval 3 -3)
(3 2 1 0 -1 -2 -3)
```

The excitement continues on the next page.

Question 2. This question explores procedures as return values.

A. Define a procedure `triple` that takes a one-argument function f and **returns a procedure** that applies f thrice:

```
STk> (define 1+ (lambda (x) (+ x 1)))
1+
STk> (define 3+ (triple 1+))
3+
STk> ((triple 3+) 10)
16
```

What value is returned by the following? Try to figure it out in your head first!

```
STk> (((triple (triple triple)) 1+) 5)
```

B. Now generalize `triple` by writing a procedure `repeated` that takes two arguments: a unary function f and and a nonnegative integer n which is the number of times f should be applied. It should **return a procedure** which applies f that many times:

```
STk> (repeaed square 2)
#[closure arglist=(x) cd7fdc]       ;; returns a procedure!
STk> ((repeated square 2) 5)
625
STk> ((repeated bf 3) '(the matrix has you))
(you)
STk> ((repeated first 0) '(luke i am your father))    ;; identity function
(luke i am your father)
```

A particularly elegant solution exists that uses `compose` from Exercise 1.42 in the book.

CS61A – Homework 2.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Recursion and iteration

Lectures: Monday June 30, Tuesday July 1

Reading: Abelson & Sussman, Section 1.2 through 1.2.4 (Pages 31–47)

In this assignment you'll practice writing procedures that evolve iterative processes. The homework is due at **8 PM on Sunday, July 3**. Please put your solutions into a file called `hw2-1.scm` and submit electronically by typing `submit hw2-1` in the appropriate directory. Include test cases and make sure that your `.scm` files loads without errors.

Question 1. You've seen the `keep` higher-order function in lecture. It takes two arguments: a predicate and a sentence. It returns a new sentence of only those elements that satisfy the predicate (i.e. those for which the predicate returns a true value):

```
STk> (keep odd? '(1 2 3 4 5 6 7))
(1 3 5 7)
STk> (keep (lambda (x) (equal? x 'foo)) '(follow the white rabbit))
()
```

Write `keep` so it generates an iterative process.

Question 2. The `fast-expt` procedure presented on Page 45 performs exponentiation in a logarithmic number of steps using successive squaring. Its order of growth is approximately $\Theta(log_2(n))$, which is pretty damn good. However, the book's version evolves a recursive process: each time n is even a call to `square` is left to be done before the function returns. Re-write `fast-expt` so it evolves an iterative process (and still uses a logarithmic number of steps, of course). The idea behind successive squaring is:

$$b^n = (b^{\frac{n}{2}})^2 = (b^2)^{\frac{n}{2}}$$

To adapt this to an iterative algorithm, you'll need to maintain an extra iteration variable, call it a for "answer," that is taken to be 1 initially; the final value of a will be the result of `fast-expt`. The value of ab^n should not change from one iteration to the next. In other words, ab^n should remain *invariant* throughout the computation. The individual values of a, b and n may change from iteration to iteration.

```
STk> (fast-expt 3 6)
729
STk> (fast-expt 2 32)
4294967296
```

The adventure continues on the next page.

Question 3. Read and complete Exercise 1.37 from SICP. Don't get intimidated by the math. This question has *nothing* to do with ϕ, the special number 1.6180, except that its inverse can be approximated with the continued fraction:

$$\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}$$

You don't need to understand the mathematical significance of ϕ. However, your `cont-frac` function should give a good approximation to $\frac{1}{\phi}$:

```
STk> (cont-frac (lambda (i) 1.0) (lambda (x) 1.0) 100)
0.618033988749895
```

But before you start approximating $\frac{1}{\phi}$, test your function with a small k-term finite continued fraction like:

$$\cfrac{1}{1+\cfrac{2}{2+\cfrac{3}{3}}}$$

There are just three terms in this fraction, making it easy to compute by hand:

```
STk> (/ 1 (+ 1 (/ 2 (+ 2 (/ 3 3)))))
0.6
```

Using `cont-frac` should give matching results:

```
STk> (cont-frac (lambda (x) x) (lambda (x) x) 3)
0.6
```

Hint: You will find it easier to count up from one to k in the recursive version, and to count down from k to zero in the iterative version.

Question 4. A *perfect number* is defined as a number equal to the sum of all its factors less than itself. For example, the first perfect number is 6, because $1 + 2 + 3 = 6$. The second perfect number is 28, because $1 + 2 + 4 + 7 + 14 = 28$. What is the third perfect number? Write a procedure `next-perfect` that takes a single number n and tests numbers starting with n until a perfect number is found:

```
STk> (next-perfect 4)
6
STk> (next-perfect 6)
6
STk> (next-perfect 7)
28
```

To find the third perfect number evaluate `(next-perf 29)`. To do this problem, you'll need a `sum-of-factors` subprocedure. If you run this program when the system is heavily loaded, it may take a while to compute the answer! Make sure your program can find 6 and 28 first.

Does `next-perfect` evolve an iterative or recursive process?

CS61A – Homework 2.2	**Kurt Meinz**
University of California, Berkeley	Summer 2005

Topic: Data abstraction

Lectures: Wednesday July 2, Thursday July 3

Reading: Abelson & Sussman, Sections 2.1 and 2.2.1 (Pages 79–106)

In this assignment you'll practice working with Scheme lists. Although lists look like sentences, you should treat them as a completely separate type. Do not use sentence operators on lists! Below is a table of sentence functions and their list counterparts:

Sentences	Lists
`first butfirst`	`car cdr`
`last butlast`	`none`
`sentence`	`list append cons`
`every`	`map`
`keep`	`filter`
`member?`	`member`
`item`	`list-ref`
`empty?`	`null?`
`count`	`length`

The homework is due at **8 PM Sunday, July 3**. Please put your solutions into a file called `hw2-2.scm` and submit them online. Include test cases, but comment them out so the file loads cleanly.

Question 1. In the sentence world, `keep` can be written in terms of `every`, like this:

```
STk> (define (keep pred sent)
        (every (lambda (e) (if (pred e) e '())) sent))
STk> (keep number? '(in 1 day it will be 1999))
(1 1999)
```

Can you use a similar trick to write `filter` using `map`, which is defined on Page 105? Why or why not?

Question 2. Read and complete Exercise 2.12 in SICP. Here is the interval ADT:

```
(define (make-interval a b) (cons a b))
(define (upper-bound interval) (car interval))
(define (lower-bound interval) (cdr interval))
```

We'll represent percentages as decimal values in the range $[0, 1]$. Here is the desired behavior:

```
STk> (define my-interval (make-center-percent 10 .1))   ;; 10% tolerance
STk> (center my-interval)
10
STk> (percent my-interval)
.1
STk> (lower-bound my-interval)
9
STk> (upper-bound my-interval)
11
```

The fun continues on the next page.

Question 3. The great thing about lists is that they can hold *any* Scheme value: numbers, booleans, even procedures! Watch:

```
STk> (define procs (list + * =))    ;; why doesn't '(+ * =) work?
procs
STk> ((car procs) 10 10 10)
30
STk> ((caddr procs) 9 11)
#f
```

A. We'd like to exploit this feature by writing a function `call-all` that takes two arguments: a list of unary procedures and an arbitrary Scheme value x. It should invoke all the procedures in the list on x in the intuitive order (the rightmost procedure is the last one invoked):

```
STk> (call-all (list (lambda (x) (- x 6)) abs sqrt) -10)
4
STk> (call-all (list cdr cdr cdr null?) '(free your mind))
#t
STk> (call-all nil 'foo)   ;; identity function
foo
STk> (call-all (list list list list) 'foo)
(((foo)))
```

Write `call-all`; there is a very simple recursive solution.

B. It'd be a lot nicer if instead of taking two arguments `call-all` could take *any number* of arguments, the last of which is x, like this:

```
STk> (call-all cdr cdr cdr null? '(free your mind))
#t
STk> (call-all 'foo)   ;; identity function
foo
```

Scheme provides a special *dotted-tail notation* for definitions that allows procedures to take an arbitrary number of arguments. For example:

```
STk> (define (f x y . z) (list x y z))
```

The procedure `f` can be called with two or more arguments. The first will be in `x`; the second in `y` and any remaining arguments will be put into a list `z`:

```
STk> (f 1 2 3 4)
(1 2 (3 4))
STk> (f 1 2)
(1 2 ())
STk> (f 1)
*** Error: wrong number of arguments to procedure: (f 1)
```

As another example, the primitive operator `list` can be defined like this:

```
STk> (define (list . args) args)
list
STk> (list 1 2 'three)
(1 2 three)
```

Use dotted-tail notation to create a new version of `call-all` that behaves as above. It should accept one or more arguments.

The fun continues on the next page.

Question 4. Since lists can contain lists, it becomes possible to create *nested* lists. Next week we'll explore nested lists and other hierarchical data structures. As a prelude, write a function `group-2` that takes a single list as argument. The number of things in the list will always be a multiple of two. The function should group every two consecutive elements into a list, returning a list of lists:

```
STk> (group-2 '(a b c d e f g h))
((a b) (c d) (e f) (g h))
STk> (group-2 '(hello (mr)))
((hello (mr)))
```

First write `group-2` so it generates a recursive process. Next write a version that generates an iterative process. The iterative solution is a bit more difficult; you may find `append` useful.

CS61A – Homework 3.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Hierarchical data

Lectures: Monday July 7, Tuesday July 8

Reading: Abelson & Sussman, Section 2.2.2–2.2.3, 2.3.1, 2.3.3

In this assignment you'll gain experience working with structures that have variable depth such as lists of lists and the Tree and Mobile abstract data types. However, the book does not have a tree ADT. In fact, when the book refers to "trees"—as in `scale-tree` on Page 112—it's really talking about deep lists.

Our tree ADT—which consists of the functions `make-tree`, `children` and `datum`—is itself usually implemented using lists:

```
(define make-tree cons)
(define datum car)
(define children cdr)
```

But remember, the underlying representation of any ADT is irrelevant! We can define `make-tree` and friends in a thousand different ways. As you do this homework, fight the desire to think of a Mobile or a Tree in terms of their underlying representations as lists.

This assignment is due at **8 PM on Sunday, July 10**. Put your answers into a file called `hw3-1.scm` and turn it in online with `submit hw3-1` as usual.

Question 1. Write a function `deep-map` that takes a unary function and a (possibly) nested list. It should apply the function to each atomic element of the list and return a new list with the same nested structure:

```
STk> (deep-map not '(#f ((#f) (#t))))
(#t ((#t) (#f)))
STk> (deep-map (lambda (a) 'foo) '())
()
STk> (deep-map (lambda (x) 'foo) '(((((3) 4) (5)) 6)))
((((foo) foo) (foo)) foo))
STk> (deep-map square '(1 2 (3) 4))
(1 4 (9) 16)
STk> (deep-map list '(1 2 (3) 4))
((1) (2) ((3)) (4))
```

The hard work continues on the next page.

Question 2. In this question we'll make use of the Tree ADT presented in lecture. A Tree can have any number of children. The constructor is `make-tree` and takes two arguments, the second of which is a *list of Trees* which are the children. The selectors are `datum` and `children`. The following code builds up the tree at right (from the bottom up):

```
(define eight (make-tree 8 '()))
(define twelve (make-tree 12 '()))
(define ten (make-tree 10 '()))
(define six (make-tree 6 '()))
(define seven (make-tree 7 '()))
(define two (make-tree 2 (list eight)))
(define nine (make-tree 9 (list ten)))
(define three (make-tree 3 (list six nine seven)))
(define one (make-tree 1 (list two twelve three)))
(define thirteen (make-tree 13 '()))
(define fourteen (make-tree 14 '()))
(define fifteen (make-tree 15 (list thirteen fourteen)))
(define zero (make-tree 0 (list one fifteen)))
```

```
      1
    / | \
   /  |  \
  2  12   3
  |      / | \
  8     6  9  7
           |
           10
```

This is a large Tree specifically so that you can play with it. In testing your code, you may want to work with one of the subtrees, such as `three` or `nine`.

Write a function `fringe` that takes a Tree and returns a list of the datums of the leaf nodes, in any order:

```
STk> (fringe zero)
(8 12 6 10 7 13 14)
STk> (fringe three)
(6 10 7)
STk> (fringe six)
(6)
```

Question 3. This question explores a Mobile ADT. A Mobile is like a tree with only two branches at every node: a right branch and a left branch. From each branch hangs either a weight, which is just a number, or another Mobile. Here is a constructor:

```
(define (make-mobile left-branch right-branch)
    (list 'mobile left-branch right-branch))
```

A branch also consists of two parts: a length and a structure. The length of a branch is numeric; the structure at the end, however, can be *either* another Mobile or a weight (a number).

```
(define (make-branch branch-length branch-structure)
    (list 'branch branch-length branch-structure))
```

The following code builds up the Mobile at right (from the bottom up):

```
STk> (define mobile-1 (make-mobile (make-branch 4 5)
                                   (make-branch 2 10)))
STk> (define mobile-2 (make-mobile (make-branch 3 10)
                                   (make-branch 2 mobile-1)))
STk> (define mobile-3 (make-mobile (make-branch 8 mobile-2)
                                   (make-branch 5 10)))
```

```
       0
    8 / \ 5
     /   \
    0    10
 3 / \ 2
  /   \
 10    0
    4 / \ 2
     /   \
    5    10
```

The learning continues on the next page.

A. There are four selectors that need to be written. Two are for Mobiles: `right-branch` and `left-branch`, and two are for branches: `branch-structure` and `branch-length`. We'll build a simple error check into the selectors to ensure they're applied to the right type:

```
(define (left-branch mobile)
   (if (and (list? mobile) (equal? (car mobile) 'mobile))
       (cadr mobile)
       (error "Not a mobile -- LEFT-BRANCH: " mobile)))
```

Write the three remaining selectors analogously. Try them out on `mobile-3` above:

```
STk> (branch-structure (right-branch mobile-3))
10
STk> (branch-length (left-branch (branch-structure (left-branch mobile-3))))
3
STk> (branch-structure
         (left-branch
            (branch-structure
               (right-branch (branch-structure (left-branch mobile-3))))))
5
```

B. Write a function `total-weight` which returns the weight of a Mobile. Assume branches are weightless; hence, only the weights increase the total weight of a Mobile:

```
STk> (total-weight mobile-1)
15
STk> (total-weight mobile-2)
25
STk> (total-weight mobile-3)
35
```

Students tend to solve this problem by performing an unnecessarily exhaustive case analysis: is the left branch a weight? is the right branch a weight? are both of them weights? This approach indicates that you don't trust the recursion. You only need *one* base case and one recursive case! Ask yourself, what are the "leaves" of a Mobile?

C. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch, and if all the other mobiles hanging beneath it are themselves balanced. The "torque applied by a branch" means the product of the `branch-length` and the `total-weight` of the `branch-structure`. For example, the torque applied by the top-right branch of the mobile (whose length is 5 and whose structure is the weight 10) is 50. Write a `balanced?` predicate that takes a Mobile and returns a true value if it is balanced, #f otherwise:

```
STk> (balanced? mobile-1)
#t
STk> (balanced? mobile-2)
#t
STk> (balanced? mobile-3)
#f
```

Aim for the simplest possible base case. You may assume that a weight by itself is *always* balanced.

Question 4. We can represent a set as a list of distinct, unordered elements. We'd like to find the subsets of such a set. The subsets of a set S are all the sets that can be formed by selecting any number of the elements of S. For example:

```
STk> (subsets '(1 2 3))
(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

Notice that the empty set is a subset of *every* set, and every set is a subset of itself. Complete the following definition of subsets.

```
(define (subsets s)
   (if (null? s)
       (list '())
       (let ((rest (subsets (cdr s))))
         (append rest (map ?? rest)))))
```

Trust the recursion!

CS61A – Homework 3.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Representing abstract data

Lectures: Wednesday July 10, Thursday July 11

Reading: Abelson & Sussman, Sections 2.4 through 2.5.2 (Pages 169–200)

This assignment gives you practice with data-directed programming and message-passing. Part of the assignment involves understanding and modifying the generic arithmetic system described in the book.

The file `~cs61a/lib/packages.scm` contains the code from the book that implements generic arithmetic. It has the definitions of `install-rectangular-package`, `install-polar-package`, `install-scheme-number-package`, `install-rational-package` and `install-complex-package`. But remember, these are just procedure definitions. **You have to invoke them to populate the table!** For convenience, we've provided the function `install-all-packages` which is defined as:

```
(define (install-all-packages)
  (install-polar-package)
  (install-rectangular-package)
  (install-complex-package)
  (install-scheme-number-package)
  (install-rational-package)
  'engage-warp-9)
```

The file also contains `apply-generic` from Page 184, the generic procedures from Pages 184 and 189, the relevant constructors and other supporting code.

This assignment is due at **8 PM on Sunday, July 10**. Put your answers into a file called `hw3-2.scm` and turn it in electronically. Comment out your test cases so the file loads smoothly.

Question 1. Write a function `add-up-complex` that takes a list of complex numbers (in polar or rectangular form) and returns a complex number representing their sum. The result should be in polar form.

```
STk> (define x (make-complex-from-real-imag 3 4))
x
STk> (define y (make-complex-from-real-imag 10 0))
y
STk> (define z (make-complex-from-mag-ang 5 1.2))
z
STk> (add-up-complex (list x y z))
(complex rectangular 14.8117887723834 . 8.66019542983613)
STk> (add-up-complex '())
(complex rectangular 0 . 0)
```

The adventure continues on the next page.

Question 2. Read and complete Exercise 2.77 on Page 192. You might want to `trace` the `apply-generic` procedure.

In case this is not clear, when Louis types

```
(put 'magnitude '(complex) magnitude)
```

the `magnitude` procedure actually inserted into the table is the one defined on Page 184. The definitions of `real-part`, `imag-part` and `angle` are there, too.

A good place to start is by reproducing Louis' error:

```
STk> (load "~cs61a/lib/packages.scm")
okay
STk> (install-all-packages)
engage-warp-9
STk> (define z (make-complex-from-real-imag 3 4))
z
STk> (magnitude z)
*** Error:
    No method for these types -- APPLY-GENERIC (magnitude (complex))
```

Question 3. We'd like to create a generic procedure `zero?` that tests if its argument is equal to zero. We're going to use `apply-generic` to define it:

```
(define (zero? x) (apply-generic 'zero? x))
```

Your job is to add something to the `complex`, `rational` and `scheme-number` packages to make this generic definition work. Here is the desired behavior:

```
STk> (zero? (make-rational 1 2))
#f
STk> (zero? (make-complex-from-real-imag 0 0))
#t
STk> (zero? (make-complex-from-mag-ang 0 1.4))   ;; zero magnitude
#t
STk> (zero? (make-scheme-number 43))
#f
STk> (zero? (make-rational 0 234))
#t
STk> (zero? (make-scheme-number 0))
#t
```

Show just the parts you added.

The learning continues on the next page.

Question 4. Berkeley is a great place to buy coffee. So many vendors to choose from: Starbucks, Tullys, Peets, Strada, etc. Some of these places offer a bulk discount: the more coffee you buy the less it costs. We'll model the pricing scheme of a given coffee vendor with a function that takes the quantity of coffee you'd like to purchase and returns the total price. Suppose we've set up a table keyed by vendor name and coffee type like this:

```
STk> (put 'starbucks 'frap (lambda (n) (* n 3.50)))
STk> (put 'starbucks 'mocha (lambda (n) (* n 1.50)))
STk> (put 'coffee-source 'frap (lambda (n)
                                 (cond ((< n 5) (* n 4.00))
                                       ((< n 10) (* n 3.00))
                                       (else (* n 2.50)))))
STk> (put 'tullys 'frap (lambda (n)
                          (cond ((< n 10) (* n 3.50))
                                ((< n 30) (* n 3.00))
                                (else (* n 2.50)))))
STk> (put 'peets 'frap (lambda (n)
                          (if (< n 50)
                              (* n 4.00)
                              (* n 2.00))))
```

To find out how much ten Starbucks fraps cost you'd type:

```
STk> ((get 'starbucks 'frap) 10)
35.0
```

Write a function `best-deal` that takes three arguments: the type of coffee, the quantity you want to purchase and a list of vendors **at least one of which sells the desired item**. It should return the *name* of the vendor with the best price for that quantity of goods. If multiple vendors exists with the same low price `best-deal` should return the first one in the list.

```
STk> (best-deal 'frap 1 '(starbucks tullys office-depot))
starbucks
STk> (best-deal 'frap 10000 '(starbucks walmart peets tullys strada))
peets
STk> (best-deal 'frap 13 '(coffee-source tullys))
coffee-source
STk> (best-deal 'mocha 87 '(peets starbucks coffee-source))
starbucks
```

As you can see, not all the vendors will sell the product desired. Some of the vendors might not even be in the table! At least one will. Recall that `get` returns #f if it does not find anything in the table matching *both* keys.

You may want to use the following helper function, which returns the first vendor in a list of vendors that sells a specific good.

```
(define (vendor-that-sells good vendors)
  (if (get (car vendors) good)
      (car vendors)
      (vendor-that-sells good (cdr vendors))))

STk> (vendor-that-sells 'mocha '(copy-central peets starbucks))
starbucks
```

The excitement continues on the next page.

Question 5. In the last homework, you implemented a Mobile ADT and wrote functions `total-weight` and `balanced?` that worked on Mobiles. Here is the Mobile constructor:

```
(define (make-mobile left-branch right-branch)
    (list 'mobile left-branch right-branch))
```

We'd now like to implement Mobiles as *message-passing objects*, similar to `make-from-real-imag` on Page 186. Here is the new Mobile constructor:

```
(define (make-mobile left-branch right-branch)
   (define (dispatch op)
      (cond ((eq? op 'left-branch) left-branch)
            ((eq? op 'right-branch) right-branch)
            (else (error "I don't understand -- MAKE-MOBILE: " op))))
   dispatch)
```

Implement `make-branch`, the constructor for branches, in message-passing style. Then write the four selectors `left-branch`, `right-branch`, `branch-structure` and `branch-length` to work with this implementation of Mobiles and branches. Test them on `mobile-3`, defined in the last homework. Your `total-weight` and `balanced?` functions should work **without modification** with this new representation of Mobiles.

CS61A – Homework 4.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Object-oriented programming

Lectures: Monday July 14, Tuesday July 15

Reading: "Object-Oriented Programming—Above-the-line view" (in course reader)

This homework gives your practice with our OOP system for Scheme. To use it, you must load `~cs61a/lib/obj.scm`. The assignment is due at **8 PM Sunday, July 17**. Put your solutions into a file `hw4-1.scm`, yadda, yadda, yadda ... you know the drill.

Question 1. Create a class called `random-generator` that takes one instantiation argument, a number *n*. An instance of this class should respond to the message `new` by returning a random number that is less than *n*. (Recall that `(random 10)` returns a random number between 0 and 9.) Any other message should cause the instance to spit out the number it returned the last time:

```
STk> (define rand1 (instantiate random-generator 10))
rand1
STk> (ask rand1 'new)
4
STk> (ask rand1 'new)
9
STk> (ask rand1 'foo)
9
STk> (ask rand1 'bar)
9
STk> (ask rand1 'baz)
9
```

If a newly instantiated `random-generator` is given a message that is not `new`, it may return anything.

Question 2. Create a `coke-machine` class. Instances of this class have one instantiation variable, the price (in cents) of a coke, and respond to five messages:

- `num-cokes` — Returns the number of cokes currently in the machine. Initially, zero.
- `fill` *n* — Fills the machine with *n* cokes. Machines start out empty. Returns anything.
- `price` — Returns the price of a coke.
- `deposit` *n* — Deposits *n* cents into the machine toward the purchase of a coke. You can deposit several coins and the machine should remember the total. Return value is up to you.
- `coke` — Returns the string `"Machine empty"`, the string `"Not enough money")` or your change, which signifies the successful purchase of a beverage. Decreases the number of cokes in the machine by one and clears the money in the machine.

Here's an example:

```
STk> (define my-machine (instantiate coke-machine 70))
STk> (ask my-machine 'num-cokes)
0
```

The question continues on the next page.

```
STk> (ask my-machine 'coke)
"Machine empty"
STk> (ask my-machine 'fill 60)             ;; return value up to you
STk> (ask my-machine 'deposit 25)          ;; return value up to you
STk> (ask my-machine 'coke)
"Not enough money"
STk> (ask my-machine 'deposit 25)          ;; now there's 50 cents in there
STk> (ask my-machine 'deposit 25)          ;; now there's 75 cents
STk> (ask my-machine 'coke)
5                                          ;; 5 cents change
STk> (ask my-machine 'num-cokes)
59
```

You may assume that the machine has an infinite supply of change and infinite space to store cokes.

Question 3. The OOP construct `usual` forwards a message to the parent class, up exactly one level in the inheritance hierarchy. Extend this capability by writing a *method* called `n-usual` that sends a message to the *n*th ancestor in the inheritance hierarchy. This feature need only work with single inheritance. The method will take two arguments: *n* and a message. If *n* is zero, the message should be given to `self`. In order for this to work, each class in the hierarchy must have the same `n-usual` method. Here is the desired behavior (with return values omitted for clarity):

```
STk> (define-class (a)
        (method (foo) (display "Foo in A") (newline))
        (method (n-usual n message) ... ))
STk> (define-class (b)
        (parent (a))
        (method (foo) (display "Foo in B") (newline))
        (method (n-usual n message) ... ))
STk> (define (c)
        (parent (b))
        (method (foo) (display "Foo in C") (newline))
        (method (n-usual n message) ... ))
STk> (define a1 (instantiate a))
STk> (define b1 (instantiate b))
STk> (define c1 (instantiate c))
STk> (ask c1 'n-usual 0 'foo)
Foo in C
STk> (ask c1 'n-usual 1 'foo)
Foo in B
STk> (ask c2 'n-usual 2 'foo)
Foo in A
```

Assume the *n*th ancestor can handle the message, and that the message takes no arguments. This problem is trickier than it looks. You'll need more than one base case.

The assignment continues on the next page.

Question 4. This exercise is mindblowingly cool. We can use OOP to represent `cons` pairs, and out of these OOP pairs we can make lists! For simplicity, assume throughout this exercise that our OOP lists will contain only *atomic* data, such as words and numbers. We'll need two classes, `oop-pair` and `the-null-list`:

```
(define-class (oop-pair the-car the-cdr)
       (method (length)
          (+ 1 (ask the-cdr 'length)))
       (method (list-ref n)
          (if (= n 0)
              the-car
              (ask the-cdr 'list-ref (- n 1)))))

(define-class (the-null-list)
       (method (length) 0)
       (method (list-ref n)
          (error "Can't LIST-REF into null list")))
```

Just like a proper list made of primitive `cons` pairs must end in `nil`, a proper OOP list must end in an instance of `the-null-list` class. Here is how you can use these definitions to construct the list `(a b c)`:

```
STk> (define my-oop-list (instantiate oop-pair 'a
                             (instantiate oop-pair 'b
                                (instantiate oop-pair 'c
                                    (instantiate the-null-list)))))
my-oop-list
STk> my-oop-list
#[closure arglist=(message) 32ddec]   ;; it's an object!
STk> (ask my-oop-list 'length)
3
STk> (ask my-oop-list 'list-ref 2)
c
```

Pause here to make sure you understand how this works.

A. It's not very convenient to construct these OOP lists as above. Define a procedure `regular->oop-list` that takes a regular Scheme list and returns the equivalent OOP list:

```
STk> (define oop-list-1 (regular->oop-list '(holy cow)))
oop-list-1
STk> oop-list-1
#[closure arglist=(message) d2d88c]       ;; it's an object!
STk> (ask oop-list-1 'length)
2
STk> (ask oop-list-1 'list-ref 0)
holy
```

The assignment continues on the next page.

B. It’s also not very convenient to view the contents of an OOP list. Add a `print` method to the `oop-pair` and `the-null-list` classes that has this behavior:

```
STk> (define oop-list-2 (regular->oop-list ’(2 soon 2 tell)))
oop-list-2
STk> (ask oop-list-2 ’print)
[2 soon 2 tell ]
okay                                              ;; return value up to you
STk> (ask (instantiate the-null-list) ’print)
[]
okay
```

Use the `display` procedure to print the elements of the list. The return value of the `print` method is up to you. We only care about its side-effect. Don’t worry about extra spaces in the output.

C. Lastly, add a `member?` method to the two class definitions:

```
STk> (define oop-list-3 (regular->oop-list ’(a prison for your mind)))
oop-list-3
STk> (ask oop-list-3 ’member? ’prison)
#t
STk> (ask oop-list-3 ’member? ’jail)
#f
```

CS61A – Homework 4.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Assignment, state, environments

Lectures: Wednesday July 16, Thursday July 17

Reading: Abelson & Sussman Sections 3.1, 3.2 (Pages 217-251) and
"Object-Oriented Programming—Below-the-line view" (in course reader)

This assignment gives you practice with procedures that have local state and with the environment model of evaluation. Do not use OOP; use regular Scheme. This assignment is due at **8 PM on Sunday, July 17**. Please put your answers to Questions 1-3 into a file `hw4-2.scm` and submit electronically. Question 4 asks you to draw an environment diagram. Please do this on a blank sheet of paper and turn it into the box labeled with your TA's name in 283 Soda before lecture on Monday. Don't forget to **write your name and login** on the paper.

Question 1. To *instrument* a procedure means to make it do something in addition to what it already does. For example, a procedure can be instrumented to keep statistics about itself, such as how many times it has been called.

A. Write a procedure `instrument` that takes a one-argument procedure f. It should return an *instrumented* version of f that keeps track of how many times it was called using a *local* counter. If the instrumented procedure is called with the special symbol `times-called`, it should return the number of times it has been invoked. If it's called with `reset`, the internal counter should be set to zero. Any other argument should be passed directly to f:

```
STk> (define i-square (instrument (lambda (x) (* x x))))
i-square
STk> (i-square 5)
25
STk> ((repeated i-square 3) 2)
256
STk> (i-square 'times-called)
4
STk> (i-square 'reset)
ok                                ;; return value up to you
STk> (i-square 'times-called)
0
```

B. We'd like to keep track of how many times `factorial` is called (including recursive calls). So we try:

```
STk> (define (factorial n)
       (if (= n 0)
           1
           (* n (factorial (- n 1)))))
factorial
STk> (define i-factorial (instrument factorial))
i-factorial
STk> (i-factorial 5)
120
STk> (i-factorial 'times-called)
1
```

Explain why `i-factorial` thinks it's only been called once, not five times. You may find it useful to draw an environment diagram, though you don't have to.

Question 2. Modify the `make-account` procedure on Page 223 to create password-protected accounts. You choose the password when you create an account; you must then supply that same password when you wish to withdraw or deposit:

```
STk> (define a1 (make-account 100 'this-is-my-password))
a1
STk> ((a1 'withdraw 'this-is-my-password) 40)
60
STk> ((a1 'withdraw 'this-is-not-my-password) 10)
Incorrect Password                              ;; print this and
ok                                              ;; return something
STk> ((a1 'deposit 'this-is-my-password) 10)
70
```

If an account is accessed three consecutive times with wrong password, display the following warning: ”Do it again and I'll call the police”. If an account is accessed more than three consecutive times with the wrong password, invoke this procedure (or your own variant):

```
(define (police)
   (display "Bad boys, bad boys\n")
   (display "Watcha gonna do whatcha gonna do when they come for you\n")
   (display "Bad boys, bad boys\n")
   (display "Watcha gonna do whatcha gonna do when they come for you\n")
   (display "...\n")
   (display "Nobody naw give you no break\n")
   (display "Police naw give you no break\n"))
```

(Lyrics from www.geocities.com/tvshowthemelyrics/CopsSong.html)

Question 3. Under the substitution model of evaluation, the *order* in which arguments were evaluated (e.g., left to right or right to left) didn't matter. With the introduction of assignment—or other side-effects, such as printing—the order in which expressions are evaluated matters a great deal; different results are possible if arguments are evaluated from left to right instead of right to left. Devise a way to test the order in which arguments are evaluated. Determine which way does `+`, `(lambda (x y z) (list x y z))` and `cons` evaluate their arguments in STk.

The fun continues on the next page.

Question 4. Draw an environment diagram for the following expressions. Also fill in the return value in each blank:

```
STk> (define make-counter
        (let ((total 0))
          (lambda ()
             (let ((count 0))
               (lambda ()
                 (set! count (+ 1 count))
                 (set! total (+ 1 total))
                 (list count total))))))
STk> (define c1 (make-counter))
STk> (c1)

________________

STk> (c1)

________________

STk> (define c2 (make-counter))
STk> (c2)

________________

STk> (c1)

________________
```

There is a program called EnvDraw that draws environment diagrams. There is no reason you shouldn't check your work using it. To run it, type `envdraw` at your shell (`%` is the shell prompt):

```
% envdraw
```

This will launch STk. From STk call the `envdraw` procedure:

```
STk> (envdraw)
okay
```

Now, any expression you evaluate at the STk prompt will be shown in the environment. To keep EnvDraw from drawing the entire diagram at once, turn on Stepping mode; when you want it to draw the next piece of the diagram, choose Step from the menu.

But wait! We did say *check* your work using EnvDraw, not let EnvDraw do the whole thing and copy. Environment diagrams are fundamental to what we do from now until the end of the semester. Putting in the time to understand them this week will make subsequent topics easier.

Lastly, it is useful to think of local state variables as either *class* variables or *instance* variables. If we treat `c1` and `c2` in the above expression as instances, which are the class and instance variables?

CS61A – Homework 5.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Mutation

Lectures: Monday July 21, Tuesday July 22

Reading: Abelson & Sussman, Section 3.3.1–3

This assignment gives you practice with mutation of pairs and circular structures made of pairs. Additionally, several of problems require an understanding of last week's material. Question 4, `count-pairs`, is a classic problem in computer science. Make sure to spend enough time on it. This homework is due at **8 PM on Sunday, July 24**. Put your answers into a file `hw5-1.scm` and submit it electronically.

Question 1. This question isn't so much about how tables work, but about using them. *Memoization* is a technique for increasing the efficiency of a program by recording previously computed results in a local table. The keys are the arguments to the memoized procedure. When the memoized procedure is asked to compute a value, it first checks the table. If the value has already been computed, just pull it out of the table. Otherwise, compute the value and store it in the table for future use. (Note that memoization only benefits procedures that are strictly *functional*; it would not make sense to memoize `random` or other procedures with side-effects.)

A. Here is the familiar procedure for computing Fibonacci numbers:

```
(define (fib n)
   (cond ((= n 0) 1)
         ((= n 1) 1)
         (else (+ (fib (- n 1)) (fib (- n 2))))))
```

Its order of growth is $\Theta(2^n)$. Here is a memoized version:

```
(define memo-fib
  (let ((history (make-table)))
    (lambda (n)
       (let ((previously-computed (lookup n history)))
          (or previously-computed
              (cond ((= n 0) 1)
                    ((= n 1) 1)
                    (else
                      (let ((result (+ (memo-fib (- n 1)) (memo-fib (- n 2)))))
                        (insert! n result history)
                        result))))))))
```

Code for one-dimentional tables is in `~cs61a/lib/tables.scm`. To get a rough idea of how much work is saved, `trace` both versions and compute the 11th Fibonacci number. Explain why `memo-fib` computes the nth Fibonacci number in a number of steps proportional to n. That is, show that `memo-fib` has roughly a linear order of growth. Treat `lookup` and `insert!` as constant-time operations.

B. Memoize the `count-change` procedure defined on Page 40 of SICP. Actually, memoize its helper `cc`. Model your `memo-cc` procedure on `memo-fib`. Notice that `cc` has a structure that is very similar to `fib`: two base cases, one recursive case but with two recursive calls. The only difference is that `cc` takes two arguments, `amount` and `kinds-of-coins`. While you can use a two-dimentional table to deal with this, it is probably easier to use a one-dimentional table and `list` both arguments for the key. Test your `memo-cc` against the original `cc` procedure to make sure it returns the same answer—but faster! You'll find the original `count-change` procedure in `~cs61a/lib/change.scm`.

The adventure continues on the next page.

Question 2. In this question we look at destructive removal of elements from a proper list.

A. Write the procedure `remove-nth!` that takes a list and a number *n*. It should *destructively* remove the *n*th element of the list (counting from zero). The return value of `remove-nth!` is up to you; it's the side-effect we're after. You may assume that *n* will be within the length of the list. Additionally, you may assume that *n* will never be zero; that is, we'll never ask `remove-nth!` to get rid of the very first list element. The desired behavior is this:

```
STk> (define red-pill (list 'how 'deep 'the 'rabbit 'hole 'is))
red-pill
STk> (remove-nth! red-pill 1)
ok                                   ;; return value is garbage
STk> red-pill
(how the rabbit hole is)
STk> (remove-nth! red-pill 3)
ok
STk> red-pill
(how the rabbit is)
```

B. Now the interesting part: why can't *n* be zero? Specifically, why is it **impossible** to write a `remove-nth!` function that can remove *all* the elements of a given list? For example:

```
STk> (define a (list 'hello))
a
STk> (remove-nth! a 0)
ok
STk> a
()
```

Assuming you have a working `remove-nth!` from Part A, why does the following definition of `remove-nth-with-zero!`, which attempts to handle the case when *n* is zero, fail?

```
(define (remove-nth-with-zero! lst n)
   (if (= n 0)
       (set! lst (cdr lst))
       (remove-nth! lst n)))     ;; call remove-nth! if n is nonzero
```

You may find it useful to draw an environment diagram (or have EnvDraw draw it for you).

Question 3. Write a function `interleave!` that takes two lists, the first of which is non-empty, and interleaves their elements using mutation. That is, `interleave!` should insert an element of the second list between every two elements of the first list. The return value of `interleave!` is up to you. Here is a sample call (with some return values omitted for clarity):

```
STk> (define numbers (list 1 2 3 4 5))
STk> (define letters (list 'a 'b))
STk> (interleave! numbers letters)
STk> numbers
(1 a 2 b 3 4 5)
STk> letters
(a 2 b 3 4 5)
```

Test `interleave!` thoroughly and include your test cases in your submission (but comment them out). **Do not allocate any new pairs!** The point of this problem is to reuse existing pairs, not make new ones. Hence, `cons` and friends are illegal.

The homework continues on the next page.

Question 4. We'd like to write a procedure `count-pairs` that returns the number of pairs in an arbitrary structure. The following is a version that would work for any structure of pairs that can be constructed *without* mutation:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

Let's take it out for a spin:

```
STk> (count-pairs (list 'a 'b 'c))
3
STk> (count-pairs (cons 'a (cons 'b 'c)))
2
STk> (count-pairs (list (list (list (list 'a)) 'b) 'c))
6
```

Mutation, however, allows us to fool `count-pairs` into thinking a structure has more pairs than it really does:

```
STk> (define test (list 'a 'b 'c))   ;; 3 pairs
test
STk> (set-car! test (cdr test))      ;; still 3 pairs
okay
STk> (count-pairs test)
5
```

Worse still, `count-pairs` will go into an infinite loop on circular structures:

```
STk> (define test (list 'a 'b 'c))
test
STk> (set-car! test test)
okay
STk> (count-pairs test)
```
doesn't return

Fix `count-pairs` so it correctly returns the number of pairs in *any* structure, circular or not. Do this by having `count-pairs` keep track of pairs it has already visited in a local list. (Yes, this means you'll need to maintain local state somewhere.) When facing a new pair, check if it is already in the list with `memq`, which is is like `member` but uses `eq?` to perform comparisons. You will need a helper.

Test the new `count-pairs` on the nastiest circular structures you can come up with.

CS61A – Homework 5.2 | **Kurt Meinz**
University of California, Berkeley | Summer 2005

Topic: Streams

Lectures: Wednesday July 23, Thursday July 24

Reading: Abelson & Sussman, Section 3.5.1–3, 3.5.5

This assignment explores infinite streams. Use `show-stream` (abbreviated as `ss`) to print a stream; it takes an optional second argument specifying the number of elements to print. This homework is due at **8PM on Sunday, July 24**. Please put your solutions into a file called `hw6-1.scm` and submit it electronically with `submit hw6-1`. As always, include test cases in your file but be sure to comment them out so the file loads smoothly.

Question 1. Write a procedure `list->stream` that takes a list as its argument and returns an infinite stream of the elements of the list, re-starting at the begging once the end of the list is reached:

```
STk> (ss (list->stream '(there is no spoon)))
(there is no spoon there is no spoon there is ...)
```

Question 2. In this question, you'll write a more general `stream-map` procedure and use it to define a stream *implicitly.*

A. We'd like to generalize the two-argument `stream-map` function defined on Page 320 so that it behaves as follows (Assume `ones` and `integers` are both infinite streams.):

```
STk> (ss (stream-map list ones integers) 5)
((1 1) (1 2) (1 3) (1 4) (1 5) ...)
```

As you can see, the new `stream-map` takes n streams and a procedure that can take n arguments. The procedure is applied to the corresponding elements of each stream. You may assume that the streams given to `stream-map` will be infinite. Hence, a base case is not needed. Complete this definition of `stream-map`:

```
(define (stream-map proc . streams)
   ( ??
     (apply proc (map  ?? streams))
     (apply stream-map (map  ?? streams))))
```

B. We'd now like to create an infinite stream of factorials:

```
STk> (ss factorials)
(1 2 6 24 120 720 5040 40320 362880 3628800 ...)
```

The nth element of this stream is $n+1$ factorial. Complete the following implicit definition of this stream:

```
(define factorials (cons-stream 1 (stream-map *  ??  ??)))
```

Notice that unlike `list->stream` from Question 1, you're not writing a function that returns a stream; instead, you're defining the variable `factorials` to be the *stream itself.* Yet, because of the delayed evaluation afforded by streams, you may refer to the stream you're defining as you're defining it! See Page 328 for a more complete discussion of *implicit* stream definitions. Do not define any helper functions for this problem. You may, however, use the `integers` stream.

The adventure continues on the next page.

Question 3. Create an infinite stream called `runs` that looks like this:

```
STk> (ss runs 15)
(1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 ... )
```

You'll probably want to use the generator approach to creating streams by defining an auxiliary function, say, `runs-generator` and calling it with some initial values. Then use it to define `runs`:

```
STk> (define runs (runs-generator parameters))
```

Question 4. Write a procedure `chocolate` that takes the name of someone who likes chocolate a lot and creates an infinite stream that says so:

```
STk> (ss (chocolate 'greg) 25) (greg likes chocolate greg really
likes chocolate greg really really likes chocolate greg really
really really likes chocolate greg really really really really
likes chocolate ... )
```

If you have trouble with this problem, try to first define a version of `chocolate` for lists that takes an additional argument: the maximum number of "really"s. Then gradually change list operations like `cons` and `append` to stream operations like `cons-stream` and `stream-append`. You'll need a helper function.

Question 5. The `pairs` procedure defined on Page 341 seems more complicated than needed. In the book's version, the first pair, represented by (S_0, T_0) on the diagram on Page 339, is formed explicitly. The `stream-map` handles the subsequent pairings of S_0. Why is the first pair a special case? Why can't `stream-map` take care of the entire row? Here is a simpler version of `pairs`:

```
(define (pairs s t)
   (interleave (stream-map (lambda (x) (list (stream-car s) x)) t)
               (pairs (stream-cdr s) (stream-cdr t))))
```

Does this work? Explain what happens when we attempt to evaluate the following with the new definition:

```
STk> (pairs integers ones)
```

CS61A – Homework 6.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Metacircular evaluator

Lectures: Monday July 28, Tuesday July 29

Reading: Abelson & Sussman, Section 4.1.1–6 (Pages 359–393)

This is the first of two homeworks on the metacircular evaluator. This assignment focuses on adding simple special forms as derived expressions and modifying the behavior of existing special forms. A version of the metacircular evaluator is available in `~cs61a/lib/mceval.scm`. Please copy it to your homework directory and rename is `hw6-1.scm`. Answer all questions by adding to or modifying the code in this file. Clearly mark the parts you changed. You may include test cases in this file (just be sure to comment them out) or in a separate file called `tests`. When you are done, you will have a Scheme interpreter that supports `let`, `let*` and an extended version of `define`, as well as have a built-in `map` higher-order procedure. This assignment is due at **8 PM on Sunday, July 31**.

To keep your sanity **test any new code in isolation** before testing it through the interpreter. Get in the habit of testing incrementally: test the smallest nontrivial piece of code first, and work your way up. This way, any errors you encounter will be closer to the code that produced them.

Lastly, remember to use `mce` to start the interpreter for the first time, since `mce` initializes the global environment. When you wish to get back to the REPL and preserve the state of the environment, use `driver-loop`.

Question 1. This question concerns adding derived expressions to the metacircular evaluator.

A. Add `let` as a special form to the metacircular evaluator by implementing a syntactic translation `let->lambda` that transforms a `let` expression into the equivalent procedure call:

```
STk> (let->lambda '(let ((a 1) (b (+ 2 3))) (* a b))))
((lambda (a b) (* a b)) 1 (+ 2 3))                     ;; returns a list!
```

Remember, `let->lambda` takes a *list* that represents a `let` expression and returns another *list* that represents the equivalent procedure call. Do not be intimidated by this problem simply because it appears in the context of the MCE. This is a simple list-manipulation problem; the only thing that is new is that the list happens to look like Scheme code. Make sure your `let->lambda` function works correctly before proceeding; test it in isolation, at the STk (not MCE!) prompt. After you have written `let->lambda`, install `let` into the interpreter by adding the following clause to `mc-eval`:

```
((let? exp) (mc-eval (let->lambda exp) env))
```

Don't forget to define the predicate `let?` in the obvious way. You should now be able to use the `let` form in your metacircular interpreter, like this:

```
;;; M-Eval input:
(let ((cadr (lambda (x) (car (cdr x)))))
  (cadr '(one two three)))

;;; M-Eval value:
two
```

The question continues on the next page.

B. The `let*` special form is similar to `let` except that the bindings are preformed sequentially (from left to right), allowing you to refer to previous `let` variables in defining later ones:

```
STk> (let* ((a 10) (b (* a a)) (c (+ a b)))
       (list a b c))
(10 100 110)
```

One way to implement `let*` is by transforming it into nested `let` expressions. That is, the expression

```
(let* ((a 10) (b (* a a)) (c (+ a b)))
  (list a b c))
```

is just syntactic sugar for

```
(let ((a 10))
  (let ((b (* a a)))
    (let ((c (+ a b)))
      (list a b c))))
```

Add `let*` to the MCE by implementing this syntactic transformation. Write the function `let*->lets` which takes a *list* that looks like a `let*` expression and returns nested lets. Before going further, test your function in isolation:

```
STk> (let*->lets '(let* ((a 10) (b (* a a)) (c (+ a b)))
       (list a b c)))
(let ((a 10)) (let ((b (* a a))) (let ((c (+ a b))) (list a b c))))
```

Then do everything else necessary to allow `let*` to be used in metacircular Scheme.

Question 2. In lab (Exercise 4.4) you added `and` and `or` to the MCE. An important detail of these two special forms is that `and` returns #f or the *last* true value. For example:

```
STk> (and 1 2 3 4)
4
```

Similarly, `or` returns #f or the *first* true value:

```
STk> (or 1 2 3 4)
1
```

Here is a naïve implementation of `or` that is intended to behave as above:

```
(define (eval-or exp env)
    (if (null? exp)
        #f
        (if (true? (mc-eval (car exp) env))
            (mc-eval (car exp) env)
            (eval-or (cdr exp) env))))
```

Please define `or?` in the standard way and add the following clause to `mc-eval`:

```
((or? exp) (eval-or (cdr exp) env))    ;; cdr to strip off the "and" tag
```

Show a sample interaction with the MCE that reveals a bug in this `eval-or`. You can use STk to see what the "right answer" is for any given `or` expression. How would you fix this bug? (You don't actually need to fix it if you don't want to.)

The action continues on the next page.

Question 3. Sometimes it's convenient to initialize a whole slew of variables with a single `define`. Modify the `eval-definition` function to cope with the definition of any number of variables. For example:

```
;;; M-Eval input:
(define a (+ 2 3)
        b (* 2 5)
        c (+ a b))
;;; M-Eval value:
ok
;;; M-Eval input:
(list a b c)
;;; M-Eval value:
(5 10 15)
```

Like `let*` in the previous problem, the bindings should be performed sequentially in a left-to-right order, allowing later bindings to refer to earlier ones. Do not implement this feature as a derived expression by, say, turning

```
(define a (+ 2 3) b (* 2 5) c (+ a b))
```

into

```
(begin (define a (+ 2 3)) (define b (* 2 5)) (define c (+ a b)))
```

Change `eval-definition` instead. Remember to always test in isolation first:

```
STk> (eval-definition '(define a (+ 2 3) b (* 2 5) c (+ a b))
                      the-global-environment)
ok
STk> (lookup-variable-value 'c the-global-environment)
15
```

Hint: You may find it convenient to change the `definition?` clause in `mc-eval` to strip off the "define" tag, like this:

```
((definition? exp) (eval-definition (cdr exp) env))
```

The learning continues on the next page.

Question 4. The MCE is missing quite a few primitive procedures. Evaluate `primitive-procedures` in STk to see which ones are available. The goal of this question is to make the higher-order function `map` available on startup in the metacircular evaluator:

```
STk> (mce)   ;; initializes interpreter

;;; M-Eval input:
(map (lambda (x) (* x x)) '(1 2 3))

;;; M-Eval value:
(1 4 9)
```

Depending on how you do this, `map` may end up a primitive procedure:

```
;;; M-Eval input:
map

;;; M-Eval value:
(primitive #[closure arglist=(func lst) 9d3c10])
```

or a compound procedure that is pre-defined in the MCE:

```
;;; M-Eval input:
map

;;; M-Eval value:
(compound-procedure (func lst) (...) <procedure-env>)
```

A. Why can't we just import STk's `map` into the MCE by adding it to the list of known primitives:

```
(define primitive-procedures
   (list (list 'car car)
         (list 'cdr cdr)
         (list 'map map)     ;; new!
               ...
```

Explore what happens when you attempt to use `map` in metacircular Scheme. **Hint:** STk's `map` is designed to be used with STk procedures, which look like `#[closure arglist=(x) d3afbc]`. What do MCE procedures look like?

B. Find a way to add `map` to the MCE. You may add it as a primitive or compound procedure, **but not as a special form**. There is no reason to make `map` a special form because `map` obeys the normal rules of evaluation.

You know you've done this right when you can use `map` immediately after initializing the interpreter (as in the example above). You may modify *any* functions or definitions you need to.

CS61A – Homework 6.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Metacircular evaluator, Lazy evaluator

Lectures: Wednesday July 30, Thursday July 31

Reading: Abelson & Sussman, Sections 4.2.2–3 (Pages 401–411)

This assignment gives you practice making substantial modifications to the metacircular evaluator, as well as introduces you to the lazy evaluator. **It is long!** As before, make a copy of `~cs61a/lib/mceval.scm` and rename it `hw6-2.scm`. Alternatively, you may use your modified metacircular evaluator from the last homework. Include test cases in this file or a separate file called `tests`. Please do Question 4 in a file called `question4.scm`. Submit all files electronically. The homework is due at **8 PM on Sunday, July 31**.

Question 1. We can create new bindings with `define` in Scheme, but there is no way to get rid of old ones. Add the `undefine` special form to the metacircular evaluator which should remove the *most local* binding of a given symbol (the same binding that would be retrieved if the symbol was to be looked up):

```
;;; M-Eval input:
(define color 'yellow)
;;; M-Eval value:
ok
;;; M-Eval input:
((lambda (color) (undefine color) color) 'green)    ;; removes local "color"
;;; M-Eval value:
yellow
;;; M-Eval input:
(undefine color)            ;; now global "color" is gone too
;;; M-Eval value:
ok                          ;; return value up to you
;;; M-Eval input:
color
*** Error: Unbound variable color
```

This problem requires you to understand the representation of environments in the interpreter. Since environments are made of pairs (surprise, surprise) you'll need `set-car!` and `set-cdr!` to change them. Make sure to test your code outside the interpreter first. This will help you isolate bugs. Assuming the underlying Scheme procedure that implements `undefine` is called `eval-undefine`, here is how you might test it:

```
STk> (define my-environment                     ;; make simple environment
            (extend-environment                 ;; with one frame that
                '(a b) '(1 2)                   ;; contains two bindings
                the-empty-environment))         ;; a=1 and b=2
STk> my-environment
(((a b) 1 2))                                   ;; peek at its representation as a list
STk> (eval-undefine 'b my-environment)
ok
STk> my-environment
(((a) 1))                                       ;; no more b
```

Lastly, briefly explain why `undefine` *must* be a special form.

The homework continues on the next page.

Question 2. We're going to borrow a neat looping construct from Emacs Lisp called `do-list`. It has this syntax:

```
(do-list (<variable> <list> <return value>)
    <body>)
```

The evaluation rules are: for every element of <*list*>, bind <*variable*> to the element and evaluate <*body*>. It's important that <*variable*> be made local to the evaluation of <*body*>. It should not exist after `do-list` is done. When the list is empty, evaluate and return <*return value*>. Below are some examples (you will need to add `display` and `newline` to the list of primitives):

```
;;; M-Eval input:
(do-list (num (list 1 2 3) 'foo)
   (display num)
   (newline))
1
2
3

;;; M-Eval value:
foo

;;; M-Eval input:
(define (reverse seq)
  (let ((result '()))
    (do-list (e seq result)
       (set! result (cons e result)))))

;;; M-Eval input:
(reverse (list 'a 'b 'c))

;;; M-Eval value:
(c b a)
```

Add `do-list` to the MCE, but **not as a derived expression!** That's less interesting. Write an evaluation function for it instead.

As you work on this problem, you may notice a slight ambiguity in the specs. Should <*return value*> be evaluated in the original environment, or in the environment where <*variable*> is bound? It's up to you.

The fun continues on the next page.

Question 3. Add `trace` and `untrace` to the metacircular evaluator. Both primitive and compound procedures should be traceable, just like in STk. Don't worry about proper indentation of trace output; we just want the basic printing of arguments and return value of a traced procedure, like this:

```
;;; M-Eval input:
(trace *)                               ;; return value omitted

;;; M-Eval input:
(* (* 2 3) (+ 4 1))

* with args (2 3)
returns 6
* with args (6 5)
returns 30

;;; M-Eval value:
30

;;; M-Eval input:
(untrace *)                              ;; * is no longer traced

;;; M-Eval input:
(define (factorial n)
   (if (= n 0)
       1
       (* n (factorial (- n 1)))))

;;; M-Eval input:
(trace factorial)

;;; M-Eval input:
(factorial 5)
factorial with args (5)
factorial with args (4)
factorial with args (3)
factorial with args (2)
factorial with args (1)
factorial with args (0)
returns 1
returns 1
returns 2
returns 6
returns 24
returns 120

;;; M-Eval value:
120
```

The return values of `trace` and `untrace` are up to you. There are several ways to do this problem, but the easiest is to stick a boolean inside the list that represents a procedure that will be true if the procedure is being traced and false otherwise. All that's left then is to figure out where procedures are called and do some printing. Depending on your implementation, `trace` and `untrace` may or may not need to be special forms. Did you make them special forms? Briefly explain.

The excitement continues on the next page.

Question 4. This question explores the difference between normal-order evaluation (as implemented by the lazy interpreter) and applicative-order evaluation (as done by STk or our own metacircular evaluator). Please put your answers to this question into a file `question4.scm`.

A. The "Hanoi stream" is an infinite stream of the form:

```
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6 1 2 1 ...
```

As you can see, every other element in the stream is a one:

```
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6 1 2 1 ...
```

If we take out all the ones, we get a stream where every other element is a two:

```
2   3   2   4   2   3   2   5   2   3   2   4   2   3   2   6   2   ...
```

And so on. This leads to the following rather intuitive generator procedure:

```
(define (make-hanoi-stream n)
   (interleave (stream-of n)
               (make-hanoi-stream (+ n 1))))

(define (stream-of x) (cons-stream x (stream-of x)))
```

The Hanoi stream can then be made by evaluating:

```
(define hanoi (make-hanoi-stream 1))
```

Try this in STk and explain the results. **Hint:** You have seen this question before.

B. Now let's try a similar approach in the lazy evaluator. Although the lazy evaluator does not have streams, it has something better: non-strict compound procedures, which allow us to implement *lazy lists*. To quote SICP (Page 409), "With lazy evaluation, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations." As shown on Page 409, implement pairs as procedures in the lazy evaluator, thereby eliminating the need to have `cons`, `car` and `cdr` as primitives. Adapt `make-hanoi-stream` and `stream-of` to create lazy lists instead of streams. You may use this definition of `interleave`:

```
(define (interleave list1 list2)
   (cons (car list1) (interleave list2 (cdr list1))))
```

Use the following procedure to print the first n elements of a lazy list (you will need to add `display` and `newline` as primitives):

```
(define (show-lazy lst n)
   (if (= n 0)
       (begin (display "...") (newline))
       (begin (display (car lst))
              (display " ")
              (show-lazy (cdr lst) (- n 1))))
   'ok)
```

Include a short session with the lazy evaluator demonstrating that the generator function works.

CS61A – Homework 7.1 | **Kurt Meinz**
University of California, Berkeley | Summer 2005

Topic: Lazy evaluator, Analyzing evaluator, Nondeterministic evaluator

Lectures: Monday August 4, Tuesday August 5

Reading: Abelson & Sussman, Section 4.1.7–4.3.2 (Pages 393–426) skim the parsing stuff

This assignment is an evaluator potpourri, giving you practice with the lazy and nondeterministic evaluators mostly "above the line."

- `~cs61a/lib/lazy.scm` – Lazy evaluator
- `~cs61a/lib/vambeval.scm` – Nondeterministic evaluator

Please put your solutions into a file called `hw7-1.scm` and submit it online as usual. Include only the code you wrote and test cases. The assignment is due at **8 PM on Sunday, August 7**.

Question 1. In the lazy evaluator `actual-value` is called in four places: to evaluate the arguments to a primitive procedure, to evaluate the operator in a procedure application, to print the results in the REPL and to evaluate the predicate in a conditional. This question investigates what happens when we replace `actual-value` with `mc-eval` in two of these. For each of the following two scenarios, describe what goes wrong and include a brief session with the lazy evaluator that demonstrates the problem.

A. Suppose we change the application clause to use `mc-eval`, like this:

```
((application? exp)
 (mc-apply (mc-eval (operator exp) env)    ;; was actual-value
           (operands exp)
           env))
```

B. Suppose we change `eval-if` to use `mc-eval`, like this:

```
(define (eval-if exp env)
   (if (true? (mc-eval (if-predicate exp) env))    ;; was actual-value
       (mc-eval (if-consequent exp) env)
       (mc-eval (if-alternative exp) env)))
```

The adventure continues on the next page.

Question 2. We'd like to write a nondeterministic program to crack a combination lock. Since there is only a finite number of combinations, all it takes is time! We will represent locks as message-passing objects created with the following procedure:

```
(define (make-lock combination)
   (lambda (message combo)
      (cond ((eq? message 'try) (if (equal? combo combination) 'open 'nice-try))
            (else (error "I don't understand " message)))))
```

As you can see, it's not a very sophisticated lock; it only knows the message `try`, which comes with one argument taken to be a test combination. If the test combination matches the real combination, the lock says `open`; otherwise it says `nice-try`.

A. Your task is to write a nondeterministic program `code-breaker` that takes a lock and returns the combination that opens it. Assume that a combination is a list of three elements

`((left` *n*`) (right` *n*`) (left` *n*`))`

where *n* is between 0 and 20, inclusive, and the directions are exactly as shown: left, right, left. Here is the desired behavior:

```
;;; Amb-Eval input:
(define lock1 (make-lock '((left 10) (right 14) (left 3))))

;;; Starting a new problem
;;; Amb-Eval value:
ok

;;; Amb-Eval input:
(code-breaker lock1)

;;; Starting a new problem
;;; Amb-Eval value:
((left 10) (right 14) (left 3))
```

B. Now let's remove the left-right-left requirement. Combinations are still three-element lists, but the directions can be in any order. Each of the following are valid combinations:

```
((left 3) (left 4) (left 5))
((right 17) (left 4) (left 15))
((right 20) (right 20) (right 20))
```

Modify your program from Part A to crack these locks.

CS61A – Homework 7.2 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Nondeterministic evaluator

Lectures: Wednesday August 6, Thursday August 7

Reading: Abelson & Sussman, Section 4.3 (Pages 412–437)

In this homework you will gain experience modifying the nondeterministic evaluator. Most of this assignment is very much "below the line." Two versions of the amb evaluator are available:

- `~cs61a/lib/ambeval.scm` — This is the nondeterministic interpreter from the book, based on the analyzing evaluator (which we have not covered).
- `~cs61a/lib/vambeval.scm` — This is a version of the nondeterministic interpreter based on the metacircular evaluator. This is also the version described in lecture. Most students find this one easier to understand. (The "v" is for vanilla.)

Copy whichever version you wish to use to do the homework into a file `hw7-2.scm` and make all modifications in this file. Clearly indicate what you changed. When you are done, you will have a nondeterministic interpreter that supports `quit`, `permanent-set!`, `or` and `if-fail`. You should include test cases either in this file (commented out), or a separate file called `tests`. Please put your answer to Question 1 into a file `question1.scm`. Submit all files electronically. The assignment is due at **8 PM on Sunday, August 7**.

All problems that ask you to add something to the nondeterministic evaluator have very short solutions. You should not be writing a lot of code at all! Wrapping your brain around continuations is the tricky part.

Question 1. Read and complete Exercise 4.42 in SICP. This is the only "above the line" problem on the homework.

Question 2. We'd like to be able to quit the amb evaluator at *any point* in the execution of a program. Add a `quit` feature to the nondeterministic evaluator that immediately returns control to STk. **It must be a clean exit—don't cause an error!** The return value of `quit` is up to you; ours returns the string "Have a nice day." The following are some examples of how `quit` should behave; `quit` must exit the amb evaluator not just from the toplevel, but from any depth in the evaluation (the bars separate different sessions with the evaluator):

```
;;; Amb-Eval input:
(quit)                                  ;; exit from toplevel

;;; Starting a new problem
"Have a nice day"

STk>
```

```
;;; Amb-Eval input:
(list 1 2 (quit) 3)                      ;; exit from subexpression evaluation

;;; Starting a new problem
"Have a nice day"

STk>
```

The question continues on the next page.

```
;;; Amb-Eval input:
(define (factorial n)
  (if (= n 0)
      (begin (newline) (quit))          ;; exit from arbitrarily deep recursion
      (begin (display n)
             (display " ")
             (* n (factorial (- n 1))))))
;;; Amb-Eval input:
(factorial 14)
;;; Starting a new problem 14 13 12 11 10 9 8 7 6 5 4 3 2 1
"Have a nice day"
STk>
```

Hint: Remember that control flow is done via continuations in the nondeterministic evaluator. To continue the computation you must invoke the success continuation; to backtrack you invoke the fail continuation. What if you call neither?

Question 3. One of the really neat things about the nondeterministic evaluator is that variable assignments are “undone” when backtracking occurs. Backtracking occurs automatically when `(amb)` is encountered; it also can be forced when the user types `try-again`. Therefore, assignments can be undone by saying `try-again`. Watch:

```
;;; Amb-Eval input:
(define neo 2)                                ;; return value omitted
;;; Amb-Eval input:
(define trinity 4)
;;; Amb-Eval input:
(define cypher 6)
;;; Amb-Eval input:
(begin (set! neo (* neo neo))
       (set! trinity (* trinity trinity))
       (set! cypher 'bloody-rat)
       (list neo trinity cypher))
;;; Starting a new problem
;;; Amb-Eval value:
(4 16 bloody-rat)                              ;; clearly the assignment takes effect
;;; Amb-Eval input:
try-again                                      ;; but it is not permanent
;;; There are no more values of ...
;;; Amb-Eval input:
(list neo trinity cypher)
;;; Starting a new problem
;;; Amb-Eval value:
(2 4 6)                                         ;; back to their old values
```

Sometimes, however, we want assignments to be permanent. Add a special form `permanent-set!` that is just like `set!` but does not get rolled back when backtracking occurs.

The question continues on the next page.

You can use `permanent-set!` to count the number of times the nondeterministic evaluator backtracks:

```
;;; Amb-Eval input:
(define count 0)                ;; return value omitted
;;; Amb-Eval input:
(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b a))))
  (permanent-set! count (+ 1 count))
  (require (not (eq? x y)))
  (list x y count))
;;; Starting a new problem
;;; Amb-Eval value:
(a b 2)
;;; Amb-Eval input
try-again
;;; Amb-Eval value:
(b a 4)
```

Hint: This question does not ask you to add new functionality, but to subtract from what's already there. Find the line(s) in `eval-assignment` that implement this undo effect and get rid of them. The failure continuation is a good place to look.

Question 4. Add the `or` special form to the nondeterministic evaluator by writing an evaluation procedure `eval-or` that handles it. **Do not add `or` as a derived expression.** As in regular Scheme, `or` should take any number of arguments and return the value of the first one that is true, or #f if none are.

You should model `eval-or` very heavily on `get-args` (code from `vambeval.scm`):

```
(define (get-args exps env succeed fail)
  (if (null? exps)
      (succeed '() fail)
      (ambeval (car exps)
               env
               (lambda (arg fail2)                    ;; first success continuation
                 (get-args (cdr exps)
                           env
                           (lambda (args fail3)           ;; second success continuation
                             (succeed (cons arg args) fail3))
                           fail2))
               fail)))
```

Like `list-of-values` in the MCE, the job of `get-args` is to evaluate a sequence of Scheme expressions, `exps`, and return a list of their values:

```
STk> (get-args '((+ 2 3) (first 'neo) (bf 'trinity))
               the-global-environment
               (lambda (result fail-cont) result)
               (lambda () 'failed))
(5 n rinity)
```

There are two success continuations. The first one is invoked if evaluating the very first expression in the sequence *does not* cause a failure; in this case, `arg` refers to the value of that first expression. The second one is invoked if the remaining expressions in the sequence were evaluated without failure; in this case, `args` is a list of their values. Notice how the list of values is built up in this second success continuation by consing `arg` into `args`.

The question continues on the next page.

A good place to start is by adding this clause to ambeval

```
((or? exp) (eval-or (cdr exp) env succeed fail))   ;; cdr to strip off "or" tag
```

and defining eval-or to do exactly what get-args does. Of course this means that or will evaluate all of its arguments and return a list of their results, which is not quite what we want, but it's a start! Try it out. Then tinker with this eval-or to make it behave as specified above. Here are some sample calls:

```
STk> (eval-or '((= 2 3) (list 1 2) this-should-not-be-evaluated)
              the-global-environment
              (lambda (result fail-cont) result)
              (lambda () 'failed))
(1 2)
STk> (eval-or '((= 2 3) (amb) this-should-not-be-evaluated)
              the-global-environment
              (lambda (result fail-cont) result)
              (lambda () 'failed))
failed
STk> (eval-or '()
              the-global-environment
              (lambda (result fail-cont) result)
              (lambda () 'failed))
#f
```

And here is how or can be used in the interpreter:

```
;;; Amb-Eval input:
(or (amb 1 2 #f) 'hello)
;;; Starting a new problem
;;; Amb-Eval value:
1
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
2
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
hello
;;; Amb-Eval input:
try-again
;;; There are no more values of
(or (amb 1 2 #f) 'hello)
```

The assignment continues on the next page.

Question 5. Read and complete Exercise 4.52 in the book. This question is more difficult than the others since you'll need to come up with the `if-fail` special form from scratch. Assuming your function for handling `if-fail` is called `eval-if-fail` and takes the entire expression as argument, here is how you might test it in isolation:

```
STk> (eval-if-fail '(if-fail (amb) 'hello)
                   the-global-environment
                   (lambda (result new-fail) result)
                   (lambda () 'failed))
hello
STk> (eval-if-fail '(if-fail (amb) (amb))
                   the-global-environment
                   (lambda (result new-fail) result)
                   (lambda () 'failed))
failed
```

Hint: To make something happen on failure, you must put it into the fail continuation.

CS61A – Homework 8.1 **Kurt Meinz**
University of California, Berkeley Summer 2005

Topic: Logic programming

Lectures: Monday August 11, Tuesday August 12

Reading: Abelson & Sussman, Section 4.4.1–3

This assignment gives you practice writing logic programs. It's very much "above the line" since we don't expect you to know how the query system works. This homework is due at **midnight on Wednesday, August 10**. Please put your solutions into a file `hw8-1.scm` and submit electronically. Make sure to include your test cases, too.

To add an assertion: `(assert!` *<conclusion>*`)`

To add a rule: `(assert! (rule` *<conclusion> <body (optional)>*`))`

Anything else is a query.

The query interpreter is in the file `~cs61a/lib/query.scm`. To initialize the interpreter type `(query)`; to re-enter the main loop without reinitializing, type `(query-driver-loop)`. Nothing—not even the rules for the `same` and `append-to-form` relations—is there when the interpreter is initialized.

Question 1. Do Exercise 4.56 in SICP. To load the database, type the following after loading `query.scm`:

```
STk> (initialize-data-base microshaft-data-base)
STk> (query-driver-loop)
```

The pattern `(?a . ?b)` matches any pair, so you can use it to print everything that is in the database.

Question 2. This question explores the unary arithmetic system described in lecture where numbers are represented as lists.

A. Note that summing two of these unary numbers merely involves joining the lists that represent them. We can define a rule for adding query numbers using `append-to-form` (Page 451):

```
;;; Query input:
(assert! (rule (?a + ?b = ?c) (append-to-form ?a ?b ?c)))
Assertion added to data base.
;;; Query input:
((a a a a) + (a a a) = ?what)
;;; Query results:
((a a a a) + (a a a) = (a a a a a a a))  ;; 4 + 3 = 7
```

Devise rules to allow multiplication of query numbers:

```
;;; Query input:
((a a a a) * (a a a) = ?what)
;;; Query results:
((a a a a) * (a a a) = (a a a a a a a a a a a a))  ;; 4 * 3 = 12
```

The question continues on the next page.

B. Using your multiplication rule from above, implement a factorial relation for query numbers:

```
;;; Query input:
((a a a a) ! = ?what)

;;; Query output:
((a a a a) ! = (a a a a a a a a a a a a a a a a a a a a a a a a))  ;; 4! = 24
```

Question 3. We can interpret the query interpreter's failure to return any results as saying, "Your query was not consistent with any assertions I know or any rules I can apply on the basis of those assertions." This can be used to implement true/false queries where the interpreter echoes the query if it is true and displays no results otherwise. For example:

```
;;; Query input:
(deep-member a ((a) b))

;;; Query output:
(deep-member a ((a) b))           ;; true

;;; Query input:
(deep-member c ((b ((a)))))

;;; Query output:
                                    ;; false

;;; Query input:
(deep-member c ((b ((a c)))))

;;; Query output:
(deep-member c ((b ((a c)))))
;;; Query input:
(deep-member ?anything ())

;;; Query output:
                                 ;; false
```

Write rules for the deep-member relation that behaves as above.

Question 4. Write query rules for the assoc relation. It should work like this:

```
;;; Query input:
(assoc carolen ((greg 10) (kurt 12) (carolen 10) (alex 13) (carolen 15)) ?what)

;;; Query results:
(assoc carolen ((greg 10) (kurt 12) (carolen 10) (alex 13) (carolen 15)) (carolen 10))

;;; Query input:
(assoc todd ((greg 10) (kurt 12) (carolen 10) (alex 13) (carolen 15)) ?what)

;;; Query results:
                                              ;; no results!
```

Notice that the *first* sublist beginning with carolen is brought forth. The query should run backward, too:

```
;;; Query input:
(assoc ?who ((greg 10) (kurt 12) (carolen 10) (alex 13) (carolen 15)) (?who 10))

;;; Query results:
(assoc greg ((greg 10) (kurt 12) (carolen 10) (alex 13) (carolen 15)) (greg 10))
(assoc carolen ((greg 10) (kurt 12) (carolen 10) (alex 13) (carolen 15)) (carolen 10))
```

CS 61A Lecture Notes First Half of Week 1

Topic: Functional programming

Reading: Abelson & Sussman, Section 1.1 (pages 1–31)

Course overview:

Computer science isn't about computers (that's electrical engineering) and it isn't primarily a science (we invent things more than we discover them). CS is partly a form of engineering (concerned with building reliable, efficient mechanisms, but in software instead of metal) and partly an art form (using programming as a medium for creative expression). Most of all, however, CS is applied logic. At its best, CS is like getting logic and math to do interesting and useful things for you.

Programming is really easy, as long as you're solving small problems. Any kid in junior high school can write programs in BASIC, and not just exercises, either; kids do quite interesting and useful things with computers. But BASIC doesn't scale up; once the problem is so complicated that you can't keep it all in your head at once, you need help, in the form of more powerful ways of thinking about programming. (But in this course we mostly use small examples, because we'd never get finished otherwise, so you have to imagine how you think each technique would work out in a larger case.)

We deal with four big programming styles/approaches/paradigms:

- Functional programming (1 month)
- Object-oriented programming (1 weeks)
- Non-deterministic programming (1 week)
- Logic programming (1 week)

The big idea of the course is *abstraction*: inventing languages that let us talk more nearly in a problem's own terms and less in terms of the computer's mechanisms or capabilities. There is a hierarchy of abstraction in computing:

```
Application programs
High-level language (Scheme)
Low-level language (C)
Machine language Architecture (registers, memory, etc)
Circuit elements (gates)
Transistors
Solid-state physics quantum mechanics
```

In 61A, we'll be dealing with only the very top levels of the pyramid; in 61C we look at lower levels. We want to start at the highest level to get you thinking right and help you avoid getting lost in the details.

Style of work: Cooperative learning. No grading curve, so no need to compete. Homework is to learn from; only tests are to test you. Don't cheat; ask for help instead. (This is the *first* CS course; if you're tempted to cheat now, how are you planning to get through the harder ones?)

Introducing ... Scheme

In 61A we program in Scheme, which is an *interactive* language. That means that instead of writing a great big program and then cranking it through all at once, you can type in a single expression and find out its value. For example:

```
3                          self-evaluating
(+ 2 3)                    function notation
(sqrt 16)                  names don't have to be punctuation
(+ (* 3 4) 5)              composition of functions


+                          functions are things in themselves
'+                         quoting
'hello                     can quote any word
'(+ 2 3)                   can quote any expression
'(good morning)            even non-expression sentences


(first 274)                functions don't have to be arithmetic
(butfirst 274)             (abbreviation bf)
(first 'hello)             works for non-numbers
(first hello)              reminder about quoting
(first (bf 'hello))        composition of non-numeric functions
(+ (first 23) (last 45))   combining numeric and non-numeric


(define pi 3.14159)        special form
pi                         value of a symbol
'pi                        contrast with quoted symbol
(+ pi 7)                   symbols work in larger expressions
(* pi pi)


(define (square x)
  (* x x))                 defining a function
(square 5)                 invoking the function
(square (+ 2 3))           composition with defined functions
```

Terminology: the *formal parameter* is the name of the argument (`x`); the *actual argument expression* is the expression used in the invocation (`(+ 2 3)`); the *actual argument value* is the value of the argument in the invocation (5). The argument's name comes from the function's definition; the argument's value comes from the invocation.

Examples:

```
(define (plural wd)
  (word wd 's))
```

This simple `plural` works for lots of words (book, computer, elephant) but not for words that end in `y` (fly, spy). So we improve it:

```
;;;;;                          In file cs61a/lectures/1.1/plural.scm
(define (plural wd)
  (if (equal? (last wd) 'y)
      (word (bl wd) 'ies)
      (word wd 's)))
```

`If` is a special form that only evaluates one of the alternatives.

Pig Latin: Move initial consonants to the end of the word and append "ay"; SCHEME becomes EMESCHAY.

```
;;;;;                          In file cs61a/lectures/1.1/pigl.scm
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))

(define (pl-done? wd)
  (vowel? (first wd)))

(define (vowel? letter)
  (member? letter '(a e i o u)))
```

`Pigl` introduces *recursion*—a function that invokes itself. More about how this works later in the week.

Another example: Remember how to play Buzz? You go around the circle counting, but if your number is divisible by 7 or has a digit 7 you have to say "buzz" instead:

```
;;;;;                          In file cs61a/lectures/1.1/buzz.scm
(define (buzz n)
  (cond ((equal? (remainder n 7) 0) 'buzz)
        ((member? 7 n) 'buzz)
        (else n)))
```

This introduces the `cond` special form for multi-way choices.

`Cond` is the big exception to the rule about the meaning of parentheses; the clauses aren't invocations.

Functions.

• A function can have any number of arguments, including zero, but must have exactly one return value. (Suppose you want two? You combine them into one, e.g., in a sentence.) It's not a function unless you always get the same answer for the same arguments.

• Why does that matter? If each little computation is independent of the past history of the overall computation, then we can *reorder* the little computations. In particular, this helps cope with parallel processors.

• The function definition provides a formal parameter (a name), and the function invocation provides an actual argument (a value). These fit together like pieces of a jigsaw puzzle. *Don't write a "function" that only works for one particular argument value!*

• Instead of a sequence of events, we have composition of functions, like $f(g(x))$ in high school algebra. We can represent this visually with function machines and plumbing diagrams.

Recursion:

```
;;;;;                        In file cs61a/lectures/1.1/argue.scm
> (argue '(i like spinach))
(i hate spinach)
> (argue '(broccoli is awful))
(broccoli is great)

(define (argue s)
  (if (empty? s)
      '()
      (se (opposite (first s))
          (argue (bf s)))))

(define (opposite w)
  (cond ((equal? w 'like) 'hate)
        ((equal? w 'hate) 'like)
        ((equal? w 'wonderful) 'terrible)
        ((equal? w 'terrible) 'wonderful)
        ((equal? w 'great) 'awful)
        ((equal? w 'awful) 'great)
        ((equal? w 'terrific) 'yucky)
        ((equal? w 'yucky) 'terrific)
        (else w) ))
```

This computes a function (the `opposite` function) of each word in a sentence. It works by dividing the problem for the whole sentence into two subproblems: an easy subproblem for the first word of the sentence, and another subproblem for the rest of the sentence. This second subproblem is just like the original problem, but for a smaller sentence.

We can take `pigl` from last lecture and use it to translate a whole sentence into Pig Latin:

```
(define (pigl-sent s)
  (if (empty? s)
      '()
      (se (pigl (first s))
          (pigl-sent (bf s)))))
```

The structure of `pigl-sent` is a lot like that of `argue`. This common pattern is called *mapping* a function over a sentence.

Not all recursion follows this pattern. Each element of Pascal's triangle is the sum of the two numbers above it:

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

Normal vs. applicative order.

To illustrate this point we use a modified Scheme evaluator that lets us show the process of applicative or normal order evaluation. We define functions using `def` instead of `define`. Then, we can evaluate expressions using `(applic (...))` for applicative order or `(normal (...))` for normal order. (Never mind how this modified evaluator itself works! Just take it on faith and concentrate on the results that it shows you.)

In the printed results, something like

```
(* 2 3) ==> 6
```

indicates the ultimate invocation of a primitive function. But

```
(f 5 9) ---->
(+ (g 5) 9)
```

indicates the substitution of actual arguments into the body of a function defined with `def`. (Of course, whether actual argument values or actual argument expressions are substituted depends on whether you used `applic` or `normal`, respectively.)

```
> (load "lectures/1.1/order.scm")
> (def (f a b) (+ (g a) b))     ; define a function
f
> (def (g x) (* 3 x))           ; another one
g
> (applic (f (+ 2 3) (- 15 6))) ; show applicative-order evaluation

(f (+ 2 3) (- 15 6))
   (+ 2 3) ==> 5
   (- 15 6) ==> 9
(f 5 9) ---->
(+ (g 5) 9)
   (g 5) ---->
   (* 3 5) ==> 15
(+ 15 9) ==> 24
24
> (normal (f (+ 2 3) (- 15 6))) ; show normal-order evaluation

(f (+ 2 3) (- 15 6)) ---->
(+ (g (+ 2 3)) (- 15 6))
   (g (+ 2 3)) ---->
   (* 3 (+ 2 3))
      (+ 2 3) ==> 5
   (* 3 5) ==> 15
   (- 15 6) ==> 9
(+ 15 9) ==> 24                 ; Same result, different process.
24
```

(continued on next page)

```
> (def (zero x) (- x x))        ; This function should always return 0.
zero
> (applic (zero (random 10)))

(zero (random 10))
   (random 10) ==> 5
(zero 5) ---->
(- 5 5) ==> 0
0                               ; Applicative order does return 0.

> (normal (zero (random 10)))

(zero (random 10)) ---->
(- (random 10) (random 10))
   (random 10) ==> 4
   (random 10) ==> 8
(- 4 8) ==> -4
-4                              ; Normal order doesn't.
```

The rule is that if you're doing functional programming, you get the same answer regardless of order of evaluation. Why doesn't this hold for `(zero (random 10))`? Because it's not a function! Why not?

Efficiency: Try computing

```
(square (square (+ 2 3)))
```

in normal and applicative order. Applicative order is more efficient because it only adds 2 to 3 once, not four times. (But later in the semester we'll see that sometimes normal order is more efficient.)

Note that the reading for the second half of the week is section 1.3, skipping 1.2 for the time being.

CS 61A Lecture Notes Second Half of Week 1

Topic: Higher-order procedures

Reading: Abelson & Sussman, Section 1.3

Note that we are skipping 1.2; we'll get to it later. Because of this, never mind for now the stuff about iterative versus recursive processes in 1.3 and in the exercises from that section.

We're all done teaching you the syntax of Scheme; from now on it's all big ideas!

This lecture's big idea is *function as object* (that is, being able to manipulate functions as data) as opposed to the more familiar view of function as process, in which there is a sharp distinction between program and data.

The usual metaphor for function as process is a recipe. In that metaphor, the recipe tells you what to do, but you can't eat the recipe; the food ingredients are the "real things" on which the recipe operates. But this week we take the position that a function is just as much a "real thing" as a number or text string is.

Compare the *derivative* in calculus: It's a function whose domain and range are functions, not numbers. The derivative function treats ordinary functions as things, not as processes. If an ordinary function is a meat grinder (put numbers in the top and turn the handle) then the derivative is a "metal grinder" (put meat-grinders in the top...).

• Using functions as arguments.

Arguments are used to generalize a pattern. For example, here is a pattern:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define pi 3.141592654)

(define (square-area r) (* r r))

(define (circle-area r) (* pi r r))

(define (sphere-area r) (* 4 pi r r))

(define (hexagon-area r) (* (sqrt 3) 1.5 r r))
```

In each of these procedures, we are taking the area of some geometric figure by multiplying some constant times the square of a linear dimension (radius or side). Each is a function of one argument, the linear dimension. We can generalize these four functions into a single function by adding an argument for the shape:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define (area shape r) (* shape r r))

(define square 1)
(define circle pi)
(define sphere (* 4 pi))
(define hexagon (* (sqrt 3) 1.5))
```

We define names for shapes; each name represents a constant number that is multiplied by the square of the radius.

In the example about areas, we are generalizing a pattern by using a variable *number* instead of a constant number. But we can also generalize a pattern in which it's a *function* that we want to be able to vary:

```
;;;;;                        In file cs61a/lectures/1.3/general.scm
(define (sumsquare a b)
  (if (> a b)
      0
      (+ (* a a) (sumsquare (+ a 1) b)) ))

(define (sumcube a b)
  (if (> a b)
      0
      (+ (* a a a) (sumcube (+ a 1) b)) ))
```

Each of these functions computes the sum of a series. For example, `(sumsquare 5 8)` computes $5^2 + 6^2 + 7^2 + 8^2$. The process of computing each individual term, and of adding the terms together, and of knowing where to stop, are the same whether we are adding squares of numbers or cubes of numbers. The only difference is in deciding which function of `a` to compute for each term. We can generalize this pattern by making *the function* be an additional argument, just as the shape number was an additional argument to the area function:

```
(define (sum fn a b)
  (if (> a b)
      0
      (+ (fn a) (sum fn (+ a 1) b)) ))
```

Here is one more example of generalizing a pattern involving functions:

```
;;;;;                        In file cs61a/lectures/1.3/filter.scm
(define (evens nums)
  (cond ((empty? nums) '())
        ((= (remainder (first nums) 2) 0)
         (se (first nums) (evens (bf nums))) )
        (else (evens (bf nums))) ))

(define (ewords sent)
  (cond ((empty? sent) '())
        ((member? 'e (first sent))
         (se (first sent) (ewords (bf sent))) )
        (else (ewords (bf sent))) ))

(define (pronouns sent)
  (cond ((empty? sent) '())
        ((member? (first sent) '(I me you he she it him her we us they them))
         (se (first sent) (pronouns (bf sent))) )
        (else (pronouns (bf sent))) ))
```

Each of these functions takes a sentence as its argument and *filters* the sentence to return a smaller sentence containing only some of the words in the original, according to a certain criterion: even numbers, words that contain the letter `e`, or pronouns. We can generalize by writing a `filter` function that takes a predicate function as an additional argument.

```
(define (filter pred sent)
  (cond ((empty? sent) '())
        ((pred (first sent)) (se (first sent) (filter pred (bf sent))) )
        (else (filter pred (bf sent))) ))
```

- Unnamed functions.

Suppose we want to compute

$$\sin^2 5 + \sin^2 6 + \sin^2 7 + \sin^2 8$$

We can use the generalized `sum` function this way:

```
> (define (sinsq x) (* (sin x) (sin x)))
> (sum sinsq 5 8)
2.408069916229755
```

But it seems a shame to have to define a named function `sinsq` that (let's say) we're only going to use this once. We'd like to be able to represent the function *itself* as the argument to `sum`, rather than the function's name. We can do this using `lambda`:

```
> (sum (lambda (x) (* (sin x) (sin x))) 5 8)
2.408069916229755
```

`Lambda` is a special form; the formal parameter list obviously isn't evaluated, but the body isn't evaluated *when we see the* `lambda`, either—only when we invoke the function can we evaluate its body.

- First-class data types.

A data type is considered *first-class* in a language if it can be

- the value of a variable (i.e., named)
- an argument to a function
- the return value from a function
- a member of an aggregate

In most languages, numbers are first-class; perhaps text strings (or individual text characters) are first-class; but usually functions are not first-class. In Scheme they are. So far we've seen the first two properties; we're about to look at the third. (We haven't really talked about aggregates yet, except for the special case of sentences, but we'll see in chapter 2 that functions can be elements of aggregates.) It's one of the design principles of Scheme that everything in the language should be first-class. Later, when we write a Scheme interpreter in Scheme, we'll see how convenient it is to be able to treat Scheme programs as data.

- Functions as return values.

```
(define (compose f g) (lambda (x) (f (g x))))
(define (twice f) (compose f f))
(define (make-adder n) (lambda (x) (+ x n)))
```

The derivative is a function whose domain and range are functions.

People who've programmed in Pascal might note that Pascal allows functions as arguments, but *not* functions as return values. That's because it makes the language harder to implement; you'll learn more about this in 164.

• Let.

We write a function that returns a sentence containing the two roots of the quadratic equation $ax^2+bx+c=0$ using the formula

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

(We assume, to simplify this presentation, that the equation has two real roots; a more serious program would check this.)

```
;;;;;                         In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (se (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
      (/ (- (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a)) ))
```

This works fine, but it's inefficient that we have to compute the square root twice. We'd like to avoid that by computing it once, giving it a name, and using that name twice in figuring out the two solutions. We know how to give something a name by using it as an argument to a function:

```
;;;;;                         In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (define (roots1 d)
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) ))
  (roots1 (sqrt (- (* b b) (* 4 a c)))) )
```

`Roots1` is an internal helper function that takes the value of the square root in the formula as its argument `d`. `Roots` calls `roots1`, which constructs the sentence of two numbers.

This does the job, but it's awkward having to make up a name `roots1` for this function that we'll only use once. As in the `sum` example earlier, we can use `lambda` to make an unnamed function:

```
;;;;;                         In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  ((lambda (d)
     (se (/ (+ (- b) d) (* 2 a))
         (/ (- (- b) d) (* 2 a)) ))
   (sqrt (- (* b b) (* 4 a c))) ))
```

This does exactly what we want. The trouble is, although it works fine for the computer, it's a little hard for human beings to read. The connection between the name `d` and the `sqrt` expression that provides its value isn't obvious from their positions here, and the order in which things are computed isn't the top-to-bottom order of the expression. Since this is something we often want to do, Scheme provides a more convenient notation for it:

```
;;;;;                         In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) )))
```

Now we have the name next to the value, and we have the value of `d` being computed above the place where it's used. But you should remember that `let` does not provide any new capabilities; it's merely an abbreviation for a `lambda` and an invocation of the unnamed function.

The unnamed function implied by the `let` can have more than one argument:

```
;;;;;                           In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c))))
        (-b (- b))
        (2a (* 2 a)))
    (se (/ (+ -b d) 2a)
        (/ (- -b d) 2a) )))
```

Two cautions: (1) These are not long-term “assignment statements” such as you may remember from other languages. The association between names and values only holds while we compute the body of the `let`. (2) If you have more than one name-value pair, as in this last example, they are not computed in sequence! Later ones can’t depend on earlier ones. They are all arguments to the same function; if you translate back to the underlying `lambda`-and-application form you’ll understand this.

Another point of interest: Please note how, by using a language with first-class functions, we can construct local variables. We say, in this case, that the **expressive power** of first-class functions includes the ability to construct local variables. Indeed, the notion that first-class unnamed procedures give us other types of functionality ”for free” will be a recurring theme of this course.

CS 61A   Lecture Notes   First Half of Week 2

Topic: Recursion and iteration

Reading: Abelson & Sussman, Section 1.2 through 1.2.4 (pages 31–72)

The next two lectures are about efficiency. Mostly in 61A we don't care about that; it becomes a focus of attention in 61B. In 61A we're happy if you can get a program working at all, except for the next 2 lectures, when we introduce ideas that will be more important to you later.

We want to know about the efficiency of algorithms, not of computer hardware. So instead of measuring runtime in microseconds or whatever, we ask about the number of times some primitive (fixed-time) operation is performed. Example:

```
;;;;;                         In file cs61a/lectures/1.2/growth.scm
(define (square x) (* x x))

(define (squares sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
          (squares (bf sent)) )))
```

To estimate the efficiency of this algorithm, we can ask, "if the sentence has N numbers in it, how many multiplications do we perform?" The answer is that we do one multiplication for each number in the argument, so we do N altogether. The amount of time needed should roughly double if the number of numbers doubles.

Another example:

```
;;;;;                         In file cs61a/lectures/1.2/growth.scm
(define (sort sent)
  (if (empty? sent)
      '()
      (insert (first sent)
              (sort (bf sent)) )))

(define (insert num sent)
  (cond ((empty? sent) (se num sent))
        ((< num (first sent)) (se num sent))
        (else (se (first sent) (insert num (bf sent)))) ))
```

Here we are sorting a bunch of numbers by comparing them against each other. If there are N numbers, how many comparisons do we do?

Well, if there are K numbers in the argument to `insert`, how many comparisons does it do? K of them. How many times do we call `insert`? N times. But it's a little tricky because each call to `insert` has a different length sentence. The range is from 0 to $N-1$. So the total number of comparisons is actually

$$0 + 1 + 2 + \cdots + (N-2) + (N-1)$$

which turns out to be $\frac{1}{2}N(N-1)$. For large N, this is roughly equal to $\frac{1}{2}N^2$. If the number of numbers doubles, the time required should quadruple.

That constant factor of $\frac{1}{2}$ isn't really very important, since we don't really know what we're halving—that is, we don't know exactly how long it takes to do one comparison. If we want a very precise measure of how many microseconds something will take, then we have to worry about the constant factors, but for an overall sense of the nature of the algorithm, what counts is the N^2 part. If we double the size of the input to a program, how does that affect the running time?

We use "Big O" notation to express this sort of approximation. We say that the running time of the `sort` function is $O(N^2)$ while the running time of the `squares` function is $O(N)$. The formal definition is

$$f(x) = O(g(x)) \Leftrightarrow \exists k, N \mid \forall x > N, |f(x)| \leq k \cdot |g(x)|$$

What does all this mean? Basically that one function is always less than another function (e.g., the time for your program to run is less than x^2) except that we don't care about constant factors (that's what the k means) and we don't care about small values of x (that's what the N means).

Why don't we care about small values of x? Because for small inputs, your program will be fast enough anyway. Let's say one program is 1000 times faster than another, but one takes a millisecond and the other takes a second. Big deal.

Why don't we care about constant factors? Because for large inputs, the constant factor will be drowned out by the order of growth—the exponent in the $O(x^i)$ notation. Here is an example taken from the book *Programming Pearls* by Jon Bentley (Addison-Wesley, 1986). He ran two different programs to solve the same problem. One was a fine-tuned program running on a Cray supercomputer, but using an $O(N^3)$ algorithm. The other algorithm was run on a Radio Shack microcomputer, so its constant factor was several million times bigger, but the algorithm was $O(N)$. For small N the Cray was much faster, but for small N both computers solved the problem in less than a minute. When N was large enough for the problem to take a few minutes or longer, the Radio Shack computer's algorithm was faster.

```
;;;;;                        In file cs61a/lectures/1.2/bentley

               t1(N) = 3.0 N^3          t2(N) = 19,500,000 N

       N       CRAY-1 Fortran           TRS-80 Basic

      10       3.0 microsec             200 millisec
     100       3.0 millisec             2.0 sec
    1000       3.0 sec                  20 sec
   10000       49 min                   3.2 min
  100000       35 days                  32 min
 1000000       95 yrs                   5.4 hrs
```

Typically, the algorithms you run across can be grouped into four categories according to their order of growth in time required. The first category is *searching* for a particular value out of a collection of values, e.g., finding someone's telephone number. The most obvious algorithm (just look through all the values until you find the one you want) is $O(N)$ time, but there are smarter algorithms that can work in $O(\log N)$ time or even in $O(1)$ (that is, constant) time. The second category is *sorting* a bunch of values into some standard order. (Many other problems that are not explicitly about sorting turn out to require similar approaches.) The obvious sorting algorithms are $O(N^2)$ and the clever ones are $O(N \log N)$. A third category includes relatively obscure problems such as matrix multiplication, requiring $O(N^3)$ time. Then there is an enormous jump to the really hard problems that require $O(2^N)$ or even $O(N!)$ time; these problems are effectively not solvable for values of N greater than one or two dozen. (Inventing faster computers won't help; if the speed of your computer doubles, that just adds 1 to the largest problem size you can handle!) Trying to find faster algorithms for these *intractable* problems is a current hot research topic in computer science.

- Iterative processes

So far we've been talking about time efficiency, but there is also memory (space) efficiency. This has gotten less important as memory has gotten cheaper, but it's still somewhat relevant because using a lot of memory increases swapping (not everything fits at once) and so indirectly takes time.

The immediate issue for today is the difference between a *linear recursive process* and an *iterative process.*

```
;;;;;                         In file cs61a/lectures/1.2/count.scm
(define (count sent)
  (if (empty? sent)
      0
      (+ 1 (count (bf sent))) ))
```

This function counts the number of words in a sentence. It takes $O(N)$ time. It also requires $O(N)$ space, not counting the space for the sentence itself, because Scheme has to keep track of N pending computations during the processing:

```
(count '(i want to hold your hand))
(+ 1 (count '(want to hold your hand)))
(+ 1 (+ 1 (count '(to hold your hand))))
(+ 1 (+ 1 (+ 1 (count '(hold your hand)))))
(+ 1 (+ 1 (+ 1 (+ 1 (count '(your hand))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (count '(hand)))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (count '())))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 1)))))
(+ 1 (+ 1 (+ 1 (+ 1 2))))
(+ 1 (+ 1 (+ 1 3)))
(+ 1 (+ 1 4))
(+ 1 5)
6
```

When we get halfway through this chart and compute `(count '())`, we aren't finished with the entire problem. We have to remember to add 1 to the result six times. Each of those remembered tasks requires some space in memory until it's finished.

Here is a more complicated program that does the same thing differently:

```
;;;;;                         In file cs61a/lectures/1.2/count.scm
(define (count sent)
  (define (iter wds result)
    (if (empty? wds)
        result
        (iter (bf wds) (+ result 1)) ))
  (iter sent 0) )
```

This time, we don't have to remember uncompleted tasks; when we reach the base case of the recursion, we have the answer to the entire problem:

```
(count '(i want to hold your hand))
(iter '(i want to hold your hand) 0)
(iter '(want to hold your hand) 1)
(iter '(to hold your hand) 2)
(iter '(hold your hand) 3)
(iter '(your hand) 4)
(iter '(hand) 5)
(iter '() 6)
6
```

When a process has this structure, Scheme does not need extra memory to remember all the unfinished tasks during the computation.

This is really not a big deal. For the purposes of this course, you should generally use the simpler linear-recursive structure and not try for the more complicated iterative structure; the efficiency savings is not worth the increased complexity. The reason Abelson and Sussman make a fuss about it is that in other programming languages any program that is recursive in *form* (i.e., in which a function invokes itself) will take (at least) linear space even if it could theoretically be done iteratively. These other languages have special iterative syntax (`for`, `while`, and so on) to avoid recursion. In Scheme you can use the function-calling mechanism and still achieve an iterative process.

- More is less: non-obvious efficiency improvements.

The nth row of Pascal's triangle contains the constant coefficients of the terms of $(a + b)^n$. Each number in Pascal's triangle is the sum of the two numbers above it. So we can write a function to compute these numbers:

```
;;;;;                           In file cs61a/lectures/1.2/pascal.scm
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

This program is very simple, but it takes $O(2^n)$ time! [Try some examples. Row 18 is already getting slow.]

Instead we can write a more complicated program that, on the surface, does a lot more work because it computes an *entire row* at a time instead of just the number we need:

```
;;;;;                           In file cs61a/lectures/1.2/pascal.scm
(define (new-pascal row col)
  (nth col (pascal-row row)) )

(define (pascal-row row-num)
  (define (iter in out)
    (if (empty? (bf in))
        out
        (iter (bf in) (se (+ (first in) (first (bf in))) out)) ))
  (define (next-row old-row num)
    (if (= num 0)
        old-row
        (next-row (se 1 (iter old-row '(1))) (- num 1)) ))
  (next-row '(1) row-num) )
```

This was harder to write, and seems to work harder, but it's incredibly faster because it's $O(N^2)$.

The reason is that the original version computed lots of entries repeatedly. The new version computes a few unnecessary ones, but it only computes each entry once.

Moral: When it really matters, think hard about your algorithm instead of trying to fine-tune a few microseconds off the obvious algorithm.

CS 61A   Lecture Notes   Second Half of Week 2

Topic: Data abstraction

Reading: Abelson & Sussman, Sections 2.1 and 2.2.1 (pages 79–106)

• Big ideas: data abstraction, abstraction barrier.

If we are dealing with some particular type of data, we want to talk about it in terms of its *meaning*, not in terms of how it happens to be represented in the computer.

Example: Here is a function that computes the total point score of a hand of playing cards. (This simplified function ignores the problem of cards whose rank-name isn't a number.)

```
;;;;;                         In file cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (empty? hand)
      0
      (+ (butlast (last hand))
         (total (butlast hand)) )))


> (total '(3h 10c 4d))
17
```

This function calls `butlast` in two places. What do those two invocations mean? Compare it with a modified version:

```
;;;;;                         In file cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (empty? hand)
      0
      (+ (rank (one-card hand))
         (total (remaining-cards hand)) )))

(define rank butlast)
(define suit last)

(define one-card last)
(define remaining-cards butlast)
```

This is more work to type in, but the result is much more readable. If for some reason we wanted to modify the program to add up the cards left to right instead of right to left, we'd have trouble editing the original version because we wouldn't know which `butlast` to change. In the new version it's easy to keep track of which function does what.

The auxiliary functions like `rank` are called *selectors* because they select one component of a multi-part datum.

Actually we're *violating* the data abstraction when we type in a hand of cards as `'(3h 10c 4d)` because that assumes we know how the cards are represented—namely, as words combining the rank number with a one-letter suit. If we want to be thorough about hiding the representation, we need *constructor* functions as well as the selectors:

```
;;;;;                          In file cs61a/lectures/2.1/total.scm
(define (make-card rank suit)
  (word rank (first suit)) )

(define make-hand se)



> (total (make-hand (make-card 3 'heart)
                    (make-card 10 'club)
                    (make-card 4 'diamond) ))
17
```

Once we're using data abstraction we can change the implementation of the data type without affecting the programs that *use* that data type. This means we can change how we represent a card, for example, without rewriting `total`:

```
;;;;;                          In file cs61a/lectures/2.1/total.scm
(define (make-card rank suit)
  (cond ((equal? suit 'heart) rank)
        ((equal? suit 'spade) (+ rank 13))
        ((equal? suit 'diamond) (+ rank 26))
        ((equal? suit 'club) (+ rank 39))
        (else (error "say what?")) ))

(define (rank card)
  (remainder card 13))

(define (suit card)
  (nth (quotient card 13) '(heart spade diamond club)))
```

We have changed the internal *representation* so that a card is now just a number between 1 and 52 (why? maybe we're programming in FORTRAN) but we haven't changed the *behavior* of the program at all. We still call `total` the same way.

Data abstraction is a really good idea because it helps keep you from getting confused when you're dealing with lots of data types, but don't get religious about it. For example, we have invented the *sentence* data type for this course. We have provided symmetric selectors `first` and `last`, and symmetric selectors `butfirst` and `butlast`. You can write programs using sentences without knowing how they're implemented. But it turns out that because of the way they *are* implemented, `first` and `butfirst` take $O(1)$ time, while `last` and `butlast` take $O(N)$ time. If you know that, your programs will be faster.

• Pairs.

To represent data types that have component parts (like the rank and suit of a card), you have to have some way to *aggregate* information. Many languages have the idea of an *array* that groups some number of elements. In Lisp the most basic aggregation unit is the *pair*—two things combined to form a bigger thing. If you want more than two parts you can hook a bunch of pairs together; we'll discuss this more next week.

The constructor for pairs is CONS; the selectors are CAR and CDR.

The book uses pairs to represent many different abstract data types: rational numbers (numerator and denominator), complex numbers (real and imaginary parts), points (x and y coordinates), intervals (low and high bounds), and line segments (two endpoints). Notice that in the case of line segments we think of the representation as *one pair* containing two points, not as three pairs containing four numbers. (That's what it means to respect a data abstraction.)

Note: What's the difference between these two:

```
(define (make-rat num den) (cons num den))
(define make-rat cons)
```

They are both equally good ways to implement a constructor for an abstract data type. The second way has a slight speed advantage (one fewer function call) but the first way has a debugging advantage because you can trace `make-rat` without tracing all invocations of `cons`.

• Data aggregation doesn't have to be primitive.

In most languages the data aggregation mechanism (the array or whatever) seems to be a necessary part of the core language, not something you could implement as a user of the language. But if we have first-class functions we can use a function to represent a pair:

```
;;;;;                          In file cs61a/lectures/2.1/cons.scm
(define (cons x y)
  (lambda (which)
    (cond ((equal? which 'car) x)
          ((equal? which 'cdr) y)
          (else (error "Bad message to CONS" message)) )))


(define (car pair)
  (pair 'car))

(define (cdr pair)
  (pair 'cdr))
```

This is like the version in the book except that they use 0 and 1 as the *messages* because they haven't introduced quoted words yet. This version makes it a little clearer what the argument named `which` means.

The point is that we can satisfy ourselves that this version of `cons`, `car`, and `cdr` works in the sense that if we construct a pair with this `cons` we can extract its two components with this `car` and `cdr`. If that's true, we don't need to have pairs built into the language! All we need is `lambda` and we can implement the rest ourselves. (It isn't really done this way, in real life, for efficiency reasons, but it's neat that it could be.)

• Big idea: abstract data type *sequence* (or *list*).

We want to represent an ordered sequence of things. (They can be any kind of things.) We *implement* sequences using pairs, with each `car` pointing to an element and each `cdr` pointing to the next pair.

What should the constructors and selectors be? The most obvious thing is to have a constructor `list` that takes any number of arguments and returns a list of those arguments, and a selector `nth` that takes a number and a list as arguments, returning the *n*th element of the list.

Scheme does provide those, but it often turns out to be more useful to select from a list differently, with a selector for the first element and a selector for all the rest of the elements (i.e., a smaller list). This helps us write recursive functions such as the mapping and filtering ones we saw for sentences earlier.

Since we are implementing lists using pairs, we ought to have specially-named constructors and selectors for lists, just like for rational numbers:

```
(define adjoin cons)
(define first car)
(define rest cdr)
```

Many Lisp systems do in fact provide `first` and `rest` as synonyms for `car` and `cdr`, but the fact is that this particular data abstraction is commonly violated; we just use the names `car`, `cdr`, and `cons` to talk about lists.

This abstract data type has a special status in the Scheme interpreter itself, because lists are read and printed using a special notation. If Scheme knew only about pairs, and not about lists, then when we construct the list `(1 2 3)` it would print as `(1 . (2 .(3 . ())))` instead.

• Lists vs. sentences.

We started out the semester using an abstract data type called *sentence* that looks a lot like a list. What's the difference, and why did we do it that way?

Our goal was to allow you to create aggregates of words without having to think about the structure of their internal representation (i.e., about pairs). We do this by deciding that the elements of a sentence must be words (not sublists), and enforcing that by giving you the constructor `sentence` that creates only sentences.

Example: One of the homework problems for this problem set asks you to reverse a list. You'll see that this is a little tricky using `cons`, `car`, and `cdr` as the problem asks, but it's easy for sentences:

```
(define (reverse sent)
  (if (empty? sent)
      '()
      (se (reverse (bf sent)) (first sent)) ))
```

To give you a better idea about what a sentence is, here's a version of the constructor function:

```
;;;;;                          In file cs61a/lectures/2.2/sentence.scm
(define (se a b)
  (cond ((word? a) (se (list a) b))
        ((word? b) (se a (list b)))
        (else (append a b)) ))

(define (word? x)
  (or (symbol? x) (number? x)) )
```

`Se` is a lot like `append`, except that the latter behaves oddly if given words as arguments. `Se` can accept words or sentences as arguments.

• Box and pointer diagrams.

Here are a few details that people sometimes get wrong about them:

1. An arrow can't point to half of a pair. If an arrowhead touches a pair, it's pointing to the entire pair, and it doesn't matter exactly where the arrowhead touches the rectangle. If you see something like

```
(define x (car y))
```

where `y` is a pair, the arrow for `x` should point to *the thing that the* `car` *of* `y` *points to*, not to the left half of the `y` rectangle.

2. The direction of arrows (up, down, left, right) is irrelevant. You can draw them however you want to make the arrangement of pairs neat. That's why it's crucial not to forget the arrowheads!

3. There must be a top-level arrow to show where the structure you're representing begins.

How do you draw a diagram for a complicated list? Take this example:

```
((a b) c (d (e f)))
```

You begin by asking yourself how many elements the list has. In this case it has three elements: first `(a b)`, then `c`, then the rest. Therefore you should draw a three-pair *backbone*: three pairs with the `cdr` of one pointing to the next one. (The final `cdr` is null.)

Only after you've drawn the backbone should you worry about making the `car`s of your three pairs point to the three elements of the top-level list.

CS 61A Lecture Notes First Half of Week 3

Topic: Hierarchical data

Reading: Abelson & Sussman, Section 2.2.2–2.2.3, 2.3.1, 2.3.3

- Trees.

Big idea: representing a hierarchy of information.

Definitions: *node*, *datum*, *root*, *branch*, *leaf*, *parent*, *child.*

The name "tree" comes from the branching structure of the pictures, like real trees in nature except that they're drawn with the root at the top and the leaves at the bottom.

A *node* is a point in the tree. In these pictures, each node includes a *datum* (the value shown at the node, such as `France` or `26`) but also includes the entire structure under that datum and connected to it, so the `France` node includes all the French cities, such as `Paris`. Therefore, **each node is itself a tree**—the terms "tree" and "node" mean the same thing! The reason we have two names for it is that we generally use "tree" when we mean the entire structure that our program is manipulating, and "node" when we mean just one piece of the overall structure. Therefore, another synonym for "node" is "subtree."

The *root node* (or just the *root*) of a tree is the node at the top. Every tree has one root node. (A more general structure in which nodes can be arranged more flexibly is called a *graph*; you'll study graphs in 61B and later courses.)

The *children* of a node are the nodes directly beneath it. For example, the children of the `26` node in the picture are the `15` node and the `33` node. The parent of particular node is the node above it. Note that exactly one node has no parent (namely, the root node).

A *branch* node is a node that has at least one child. A *leaf* node is a node that has no children. (The root node is usually a branch node, except in the trivial case of a one-node tree.)

What are trees good for?

- Hierarchy: world, countries, states, cities.
- Ordering: binary search trees.
- Composition: arithmetic operations at branches, numbers at leaves.

- Below-the-line representation of trees.

Lisp has one built-in way to represent sequences, but there is no official way to represent trees. Why not?

- Branch nodes may or may not have data.
- Binary vs. n-way trees.
- Order of siblings may or may not matter.
- Can tree be empty?

We can think about a tree ADT in terms of a selector and constructors:

```
(make-tree datum children)
(datum node)
(children node)
```

The selector `children` should return a list (sequence) of the children of the node. These children are themselves trees. A leaf node is one with no children:

```
(define (leaf? node)
  (null? (children node)) )
```

This definition of leaf? should work no matter how we represent the ADT.

If every node in your tree has a datum, then the straightforward implementation is

```
;;;;;                         Compare file cs61a/lectures/2.2/tree1.scm
(define make-tree cons)
(define datum car)
(define children cdr)
```

On the other hand, it's also common to think of any list structure as a tree in which the leaves are words and the branch nodes don't have data. For example, a list like

```
(a (b c d) (e (f g) h))
```

can be thought of as a tree whose root node has three children: the leaf a and two branch nodes. For this sort of tree it's common not to use formal ADT selectors and constructors at all, but rather just to write procedures that handle the car and the cdr as subtrees. To make this concrete, let's look at mapping a function over all the data in a tree.

First we review mapping over a sequence:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES seq)
  (if (null? seq)
      '()
      (cons (SQUARE (car seq))
            (SQUARES (cdr seq)) )))
```

The pattern here is that we apply some operation (square in this example) to the data, the elements of the sequence, which are in the cars of the pairs, and we recur on the sublists, the cdrs.

Now let's look at mapping over the kind of tree that has data at every node:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES tree)
  (make-tree (SQUARE (datum tree))
             (map SQUARES (children tree)) ))
```

Again we apply the operation to every datum, but instead of a simple recursion for the rest of the list, we have to recur for *each child* of the current node. We use map (mapping over a sequence) to provide several recursive calls instead of just one.

If the data are only at the leaves, we just treat each pair in the structure as containing two subtrees:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES tree)
  (cond ((null? tree) '())
        ((atom? tree) (SQUARE tree))
        (else (cons (SQUARES (car tree))
                    (SQUARES (cdr tree)) )) ))
```

The hallmark of tree recursion is to recurse for both the car and the cdr.

- **Mapping over trees**

One thing we might want to do with a tree is create another tree, with the same shape as the original, but with each datum replaced by some function of the original. This is the tree equivalent of map for lists.

```
;;;;;                         In file cs61a/lectures/2.2/tree1.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (map (lambda (t) (treemap fn t))
                  (children tree) )))
```

This is a remarkably simple and elegant procedure, especially considering the versatility of the data structures it can handle (trees of many different sizes and shapes). It's one of the more beautiful things you'll see in the course, so spend some time appreciating it.

Every tree node consists of a datum and some children. In the new tree, the datum corresponding to this node should be the result of applying fn to the datum of this node in the original tree. What about the children of the new node? There should be the same number of children as there are in the original node, and each new child should be the result of calling treemap on an original child. Since a forest is just a list, we can use map (not treemap!) to generate the new children.

• Mutual recursion

Pay attention to the strange sort of recursion in this procedure. Treemap does not actually call itself! Treemap calls map, giving it a function that in turn calls treemap. The result is that each call to treemap may give rise to any number of recursive calls, via map: one call for every child of this node.

This pattern (procedure A invokes procedure B, which invokes procedure A) is called *mutual recursion.* We can rewrite treemap without using map, to make the mutual recursion more visible:

```
;;;;;                         In file cs61a/lectures/2.2/tree11.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (forest-map fn (children tree))))

(define (forest-map fn forest)
  (if (null? forest)
      '()
      (cons (treemap fn (car forest))
            (forest-map fn (cdr forest)))))
```

Forest-map is a helper function that takes a forest, not a tree, as argument. Treemap calls forest-map, which calls treemap.

Mutual recursion is what makes it possible to explore the two-dimensional tree data structure fully. In particular, note that reaching the base case in forest-map does not mean that the entire tree has been visited! It means merely that one group of sibling nodes has been visited (a "horizontal" base case), or that a node has no children (a "vertical" base case). The entire tree has been seen when every child of the root node has been completed.

Note that we use cons, car, and cdr when manipulating a forest, but we use make-tree, datum, and children when manipulating a tree. Some students make the mistake of thinking that data abstraction means "always say datum instead of car"! But that defeats the purpose of using different selectors and constructors for different data types.

• Deep lists

Trees are our first two-dimensional data structure. But there's a sense in which any list that has lists as elements is also two-dimensional, and can be viewed as a kind of tree. We'll use the name *deep lists* for lists that contain lists. For example, the list

```
[[john lennon] [paul mccartney] [george harrison] [ringo starr]]
```

is probably best understood as a sequence of sentences, but instead we can draw a picture of it as a sort of tree:

Don't be confused; this is *not* an example of the Tree abstract data type we've just developed. In this picture, for example, only the "leaf nodes" contain data, namely words. We didn't make this list with `make-tree`, and it wouldn't make sense to examine it with `datum` or `children`.

But we can still use the *ideas* of tree manipulation if we'd like to do something for every word in the list. Compare the following procedure with the first version of `treemap` above:

```
;;;;;                        In file cs61a/lectures/2.2/tree22.scm
(define (deep-map fn lol)
  (if (list? lol)
      (map (lambda (element) (deep-map fn element))
       lol)
      (fn lol)))
```

The formal parameter `lol` stands for "list of lists." This procedure includes the two main tasks of `treemap`: applying the function `fn` to one datum, and using `map` to make a recursive call for each child.

But `treemap` applies to the Tree abstract data type, in which every node has both a datum and children, so `treemap` carries out both tasks for each node. In a deep list, by contrast, the "branch nodes" have children but no datum, whereas the "leaf nodes" have a datum but no children. That's why `deep-map` chooses only one of the two tasks, using `if` to distinguish branches from leaves.

Note: SICP does not define a Tree abstract data type; they use the term "tree" to describe what I'm calling a deep list. So they use the name `tree-map` in Exercise 2.31, page 113, which asks you to write what I've called `deep-map`. (Since I've done it here, you should do the exercise without using `map`.) SICP does define an abstract data type for *binary* trees, in which each node can have a `left-branch` and/or a `right-branch`, rather than having any number of children.

• **Car/cdr recursion**

Consider the deep list `((a b) (c d))`. Ordinarily we would draw its box and pointer diagram with a horizontal spine at the top and the sublists beneath the spine:

But imagine that we grab the first pair of this structure and "shake" it so that the pairs fall down as far as they can. We'd end up with this diagram:

Note that these two diagrams represent the same list! They have the same pairs, with the same links from one pair to another. It's just the position of the pairs on the page that's different. But in this new picture, the structure looks a lot like a binary tree, in which the branch nodes are pairs and the leaf nodes are atoms (non-pairs). The "left branch" of each pair is its `car`, and the "right branch" is its `cdr`. With this metaphor, we can rewrite `deep-map` to look more like a binary tree program:

```
;;;;;                          In file cs61a/lectures/2.2/tree3.scm
(define (deep-map fn xmas)
  (cond ((null? xmas) '())
        ((pair? xmas)
         (cons (deep-map fn (car xmas))
               (deep-map fn (cdr xmas))))
        (else (fn xmas))))
```

(The formal parameter `xmas` reflects the fact that the picture looks kind of like a Christmas tree.)

This procedure strongly violates data abstraction! Ordinarily when dealing with lists, we write programs that treat the car and the cdr differently, reflecting the fact that the car of a pair is a list element, whereas the cdr is a sublist. But here we treat the car and the cdr identically. One advantage of this approach is that it works even for improper lists:

```
> (deep-map square '((3 . 4) (5 6))
((9 . 16) (25 36))
```

- **Tree recursion**

Compare the car/cdr version of `deep-map` with ordinary `map`:

```
(define (map fn seq)
  (if (null? seq)
      '()
      (cons (fn (car seq))
        (map fn (cdr seq)))))
```

Each non-base-case invocation of `map` gives rise to one recursive call, to handle the cdr of the sequence. The car, an element of the list, is not handled recursively.

By contrast, in `deep-map` there are *two* recursive calls, one for the car and one for the cdr. This is what makes the difference between a sequential, one-dimensional process and the two-dimensional process used for deep lists and for the Tree abstraction.

A procedure in which each invocation makes more than one recursive call is given the name *tree recursion* because of the relationship between this pattern and tree structures. It's tree recursion only if each call (other than a base case) gives rise to two or more recursive calls; it's not good enough to have two recursive calls of which only one is chosen each time, as in the following non-tree-recursive procedure:

```
(define (filter pred seq)
  (cond ((null? seq) '())
    ((pred (car seq)) (cons (car seq) (filter pred (cdr seq))))
    (else (filter pred (cdr seq)))))
```

There are two recursive calls to `filter`, but only one of them is actually carried out each time, so this is a sequential recursion, not a tree recursion.

A program can be tree recursive even if there is no actual tree-like data structure used, as in the Fibonacci number function:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

This procedure just handles numbers, not trees, but each non-base-case call adds the results of two recursive calls, so it's a tree recursive program.

- **Tree traversal**

Many problems involve visiting each node of a tree to look for or otherwise process some information there. Maybe we're looking for a particular node, maybe we're adding up all the values at all the nodes, etc. There is one obvious order in which to traverse a sequence (left to right), but many ways in which we can traverse a tree.

In the following examples, we "visit" each node by printing the datum at that node. If you apply these procedures to actual trees, you can see the order in which the nodes are visited.

Depth-first traversal: Look at a given node's children before its siblings.

```
;;;;;                         In file cs61a/lectures/2.2/search.scm
(define (depth-first-search tree)
  (print (datum tree))
  (for-each depth-first-search (children tree)))
```

This is the easiest way, because the program's structure follows the data structure; each child is traversed in its entirety (that is, including grandchildren, etc.) before looking at the next child.

Breadth-first traversal: Look at the siblings before the children.

What we want to do is take horizontal slices of the tree. First we look at the root node, then we look at the children of the root, then the grandchildren, and so on. The program is a little more complicated because the order in which we want to visit nodes isn't the order in which they're connected together.

To solve this, we use an extra data structure, called a *queue*, which is just an ordered list of tasks to be carried out. Each "task" is a node to visit, and a node is a tree, so a list of nodes is just a forest. The iterative helper procedure takes the first task in the queue (the car), visits that node, and adds its children at the end of the queue (using `append`).

```
;;;;;                          In file cs61a/lectures/2.2/search.scm
(define (breadth-first-search tree)
  (bfs-iter (list tree)))

(define (bfs-iter queue)
  (if (null? queue)
      'done
      (let ((task (car queue)))
        (print (datum task))
        (bfs-iter (append (cdr queue) (children task))))))
```

Why would we use this more complicated technique? For example, in some situations the same value might appear as a datum more than once in the tree, and we want to find the *shortest* path from the root node to a node containing that datum. To do that, we have to look at nodes near the root before looking at nodes far away from the root.

Another example is a game-strategy program that *generates* a tree of moves. The root node is the initial board position; each child is the result of a legal move I can make; each child of a child is the result of a legal move for my opponent, and so on. For a complicated game, such as chess, the move tree is much too large to generate in its entirety. So we use a breadth-first technique to generate the move tree up to a certain depth (say, ten moves), then we look for desirable board positions at that depth. (If we used a depth-first program, we'd follow one path all the way to the end of the game before starting to consider a different possible first move.)

For binary trees, within the general category of depth-first traversals, there are three possible variants:

Preorder: Look at a node before its children.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (pre-order tree)
  (cond ((null? tree) '())
        (else (print (entry tree))
              (pre-order (left-branch tree))
              (pre-order (right-branch tree)) )))
```

Inorder: Look at the left child, then the node, then the right child.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (in-order tree)
  (cond ((null? tree) '())
        (else (in-order (left-branch tree))
              (print (entry tree))
              (in-order (right-branch tree)) )))
```

Postorder: Look at the children before the node.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (post-order tree)
  (cond ((null? tree) '())
        (else (post-order (left-branch tree))
              (post-order (right-branch tree))
              (print (entry tree)) )))
```

For a tree of arithmetic operations, preorder traversal looks like Lisp; inorder traversal looks like conventional arithmetic notation; and postorder traversal is the HP calculator "reverse Polish notation."

• Path finding

As an example of a somewhat more complicated tree program, suppose we want to look up a place (e.g., a city) in the world tree, and find the path from the root node to that place:

```
> (find-place 'berkeley world-tree)
(world (united states) california berkeley)
```

If a place isn't found, `find-place` will return the empty list.

To find a place within some tree, first we see if the place is the datum of the root node. If so, the answer is a one-element list containing just the place. Otherwise, we look at each child of the root, and see if we can find the place within that child. If so, the path within the complete tree is the path within the child, but with the root datum added at the front of the path. For example, the path to Berkeley within the USA subtree is

```
((united states) california berkeley)
```

so we put `world` in front of that.

Broadly speaking, this program has the same mutually recursive tree/forest structure as the other examples we've seen, but one important difference is that once we've found the place we're looking for, there's no need to visit other subtrees. Therefore, we don't want to use `map` or anything equivalent to handle the children of a node; we want to check the first child, see if we've found a path, and only if we haven't found it should we go on to the second child (if any). This is the reason for the `let` in `find-forest`.

```
;;;;;                          In file cs61a/lectures/2.2/world.scm
(define (find-place place tree)
  (if (eq? place (datum tree))
      (cons (datum tree) '())
      (let ((try (find-forest place (children tree))))
    (if (not (null? try))
        (cons (datum tree) try)
        '()))))

(define (find-forest place forest)
  (if (null? forest)
      '()
      (let ((try (find-place place (car forest))))
        (if (not (null? try))
            try
            (find-forest place (cdr forest))))))
```

(Note: In 61B we come back to trees in more depth, including the study of *balanced* trees, i.e., using special techniques to make sure a search tree has about as much stuff on the left as on the right.)

CS 61A   Lecture Notes   Second Half of Week 3

Topic: Representing abstract data

Reading: Abelson & Sussman, Sections 2.4 through 2.5.2 (pages 169–200)

The overall problem we're addressing in the next two lectures is to control the complexity of large systems with many small procedures that handle several types of data. We are building toward the idea of *object-oriented programming*, which many people see as the ultimate solution to this problem, and which we discuss for two weeks starting next week.

Big ideas:

- tagged data
- data-directed programming
- message passing

The first problem is keeping track of types of data. If we see a pair whose `car` is 3 and whose `cdr` is 4, does that represent $\frac{3}{4}$ or does it represent $3 + 4i$?

The solution is *tagged data*: Each datum carries around its own type information. In effect we do `(cons 'rational (cons 3 4))` for the rational number $\frac{3}{4}$, although of course we use an ADT.

Just to get away from the arithmetic examples in the text, we'll use another example about geometric shapes. Our data types will be squares and circles; our operations will be area and perimeter.

We want to be able to say, e.g., `(area circle3)` to get area of a particular (previously defined) circle. To make this work, the function `area` has to be able to tell which type of shape it's seeing. We accomplish this by attaching a type tag to each shape:

```
;;;;;                        In file cs61a/lectures/2.4/geom.scm
(define pi 3.141592654)

(define (make-square side)
  (attach-tag 'square side))

(define (make-circle radius)
  (attach-tag 'circle radius))

(define (area shape)
  (cond ((eq? (type-tag shape) 'square)
         (* (contents shape) (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* pi (contents shape) (contents shape)))
        (else (error "Unknown shape -- AREA"))))

(define (perimeter shape)
  (cond ((eq? (type-tag shape) 'square)
         (* 4 (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* 2 pi (contents shape)))
        (else (error "Unknown shape -- PERIMETER"))))

;; some sample data

(define square5 (make-square 5))
(define circle3 (make-circle 3))
```

• Orthogonality of types and operators.

The next problem to deal with is the proliferation of functions because you want to be able to apply every operation to every type. In our example, with two types and two operations we need four algorithms.

What happens when we invent a new type? If we write our program in the *conventional* (i.e., old-fashioned) style as above, it's not enough to add new functions; we have to modify all the operator functions like `area` to know about the new type. We'll look at two different approaches to organizing things better: *data-directed programming* and *message passing.*

The idea in DDP is that instead of keeping the information about types versus operators inside functions, as `cond` clauses, we record this information in a data structure. A&S provide tools `put` to set up the data structure and `get` to examine it:

```
> (get 'foo 'baz)
#f
> (put 'foo 'baz 'hello)
> (get 'foo 'baz)
hello
```

Once you `put` something in the table, it stays there. (This is our first departure from functional programming. But our intent is to set up the table at the beginning of the computation and then to treat it as *constant* information, not as something that might be different the next time you call `get`, despite the example above.) For now we take `put` and `get` as primitives; we'll see how to build them in section 3.3 in two weeks

The code is mostly unchanged from the conventional version; the tagged data ADT and the two shape ADTs are unchanged. What's different is how we represent the four algorithms for applying some operator to some type:

```
;;;;;                         In file cs61a/lectures/2.4/geom.scm

(put 'square 'area (lambda (s) (* s s)))
(put 'circle 'area (lambda (r) (* pi r r)))
(put 'square 'perimeter (lambda (s) (* 4 s)))
(put 'circle 'perimeter (lambda (r) (* 2 pi r)))
```

Notice that the entry in each cell of the table is a *function*, not a symbol. We can now redefine the six generic operators ("generic" because they work for any of the types):

```
;;;;;                         In file cs61a/lectures/2.4/geom.scm

(define (area shape)
  (operate 'area shape))

(define (perimeter shape)
  (operate 'perimeter shape))

(define (operate op obj)      ;; like APPLY-GENERIC but for one operand
  (let ((proc (get (type-tag obj) op)))
    (if proc
        (proc (contents obj))
        (error "Unknown operator for type"))))
```

Now if we want to invent a new type, all we have to do is a few `put` instructions and the generic operators just automatically work with the new type.

Don't get the idea that DDP just means a two-dimensional table of operator and type names! DDP is a very general, great idea. It means putting the details of a system into data, rather than into programs, so you can write general programs instead of very specific ones.

In the old days, every time a company got a computer they had to hire a bunch of programmers to write things like payroll programs for them. They couldn't just use someone else's program because the details would be different, e.g., how many digits in the employee number. These days you have general business packages and each company can "tune" the program to their specific purpose with a data file.

Another example showing the generality of DDP is the *compiler compiler.* It used to be that if you wanted to invent a new programming language you had to start from scratch in writing a compiler for it. But now we have formal notations for expressing the syntax of the language. (See section 7.1, page 38, of the *Scheme Report* at the back of the course reader.) A single program can read these formal descriptions and compile any language. [The Scheme BNF is in `cs61a/lectures/2.4/bnf`.]

- Message-passing.

In conventional style, the operators are represented as functions that know about the different types; the types themselves are just data. In DDP, the operators and types are all data, and there is one universal `operate` function that does the work. We can also stand conventional style on its head, representing the *types* as functions and the operations as mere data.

In fact, not only are the types functions, but so are the individual data themselves. That is, there is a function (`make-circle` below) that represents the circle type, and when you invoke that function, it returns *a function* that represents the particular circle you give it as its argument. Each circle is an *object* and the function that represents it is a *dispatch procedure* that takes as its argument a *message* saying which operation to perform.

```
;;;;;                        In file cs61a/lectures/2.4/geom.scm

(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area)
           (* side side))
          ((eq? message 'perimeter)
           (* 4 side))
          (else (error "Unknown message")))))

(define (make-circle radius)
  (lambda (message)
    (cond ((eq? message 'area)
           (* pi radius radius))
          ((eq? message 'perimeter)
           (* 2 pi radius))
          (else (error "Unknown message")))))

(define square5 (make-square 5))
(define circle3 (make-circle 3))
```

The defines that produce the individual shapes look no different from before, but the results are different: Each shape is a function, not a list structure. So to get the area of the shape circle3 we invoke that shape with the proper message: (circle3 'area). That notation is a little awkward so we provide a little "syntactic sugar" that allows us to say (area circle3) as in the past:

```
;;;;;                          In file cs61a/lectures/2.4/msg.scm
(define (operate op obj)
  (obj op))

(define (area shape)
  (operate 'area shape))

(define (perimeter shape)
  (operate 'perimeter shape))
```

Message passing may seem like an overly complicated way to handle this problem of shapes, but we'll see next week that it's one of the key ideas in creating object-oriented programming. Message passing becomes much more powerful when combined with the idea of *local state* that we'll learn next week.

We seem to have abandoned tagged data; every shape type is just some function, and it's hard to tell which type of shape a given function represents. We could combine message passing with tagged data, if desired, by adding a type message that each object understands.

```
(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area)
           (* side side))
          ((eq? message 'perimeter)
           (* 4 side))
          ((EQ? MESSAGE 'TYPE) 'SQUARE)
          (else (error "Unknown message")))))
```

• Dyadic operations.

Our shape example is easier than the arithmetic example in the book because our operations only require one operand, not two. For arithmetic operations like +, it's not good enough to connect the operation with a type; the two operands might have two different types. What should you do if you have to add a rational number to a complex number?

There is no perfect solution to this problem. For the particular case of arithmetic, we're lucky in that the different types form a sequence of larger and larger sets. Every integer is a rational number; every rational is a real; every real is a complex. So we can deal with type mismatch by *raising* the less-complicated operand to the type of the other one. To add a rational number to a complex number, raise the rational number to complex and then you're left with the problem of adding two complex numbers. So we only need N addition algorithms, not N^2 algorithms, where N is the number of types.

Do we need N^2 raising algorithms? No, because we don't have to know directly how to raise a rational number to complex. We can raise the rational number to the next higher type (real), and then raise that real number to complex. So if we want to add $\frac{1}{3}$ and $2 + 5i$ the answer comes out $2.3333 + 5i$.

As this example shows, nonchalant raising can lose information. It would be better, perhaps, if we could get the answer $\frac{7}{3} + 5i$ instead of the decimal approximation. Numbers are a rat's nest full of traps for the unwary. You will live longer if you only write programs about integers.

CS 61A Lecture Notes First Half of Week 4

Topic: Object-oriented programming

Reading: OOP Above-the-line notes in course reader

OOP is an abstraction. Above the line we have the metaphor of multiple independent intelligent agents; instead of one computer carrying out one program we have hordes of *objects* each of which can carry out computations. To make this work there are three key ideas within this metaphor:

- Message passing: An object can ask other objects to do things for it.
- Local state: An object can remember stuff about its own past history.
- Inheritance: One object type can be just like another except for a few

We have invented an OOP language as an extension to Scheme. Basically you are still writing Scheme programs, but with the vocabulary extended to use some of the usual OOP buzzwords. For example, a *class* is a type of object; an *instance* is a particular object. "Complex number" is a class; $3 + 4i$ is an instance. Here's how the message-passing complex numbers from last week would look in OOP notation:

```
;;;;;                       In file cs61a/lectures/3.0/demo.scm
(define-class (complex real-part imag-part)
  (method (magnitude)
    (sqrt (+ (* real-part real-part)
             (* imag-part imag-part))))
  (method (angle)
    (atan (/ imag-part real-part))) )

> (define c (instantiate complex 3 4))
> (ask c 'magnitude)
5
> (ask c 'real-part)
3
```

This shows how we define the *class* `complex`; then we create the *instance* `c` whose value is $3 + 4i$; then we send `c` a message (we `ask` it to do something) in order to find out that its magnitude is 5. We can also ask `c` about its *instantiation variables*, which are the arguments used when the class is instantiated.

When we send a message to an object, it responds by carrying out a *method*, i.e., a procedure that the object associates with the message.

So far, although the notation is new, we haven't done anything different from what we did last week in chapter 2. Now we take the big step of letting an object remember its past history, so that we are no longer doing functional programming. The result of sending a message to an object depends not only on the arguments used right now, but also on what messages we've sent the object before:

```
;;;;;                       In file cs61a/lectures/3.0/demo.scm
(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (+ count 1))
    count) )

> (define c1 (instantiate counter))
> (ask c1 'next)
1
> (ask c1 'next)
2
```

```
> (define c2 (instantiate counter))
> (ask c2 'next)
1
> (ask c1 'next)
3
```

Each counter has its own *instance variable* to remember how many times it's been sent the `next` message.

Don't get confused about the terms *instance* variable versus *instantiation* variable. They are similar in that each instance has its own version; the difference is that instantiation variables are given values when an instance is created, using extra arguments to `instantiate`, whereas the initial values of instance variables are specified in the class definition and are generally the same for every instance (although the values may change as the computation goes on.)

Methods can have arguments. You supply the argument when you `ask` the corresponding message:

```
;;;;;                          In file cs61a/lectures/3.0/demo.scm
(define-class (doubler)
  (method (say stuff) (se stuff stuff)))

> (define dd (instantiate doubler))
> (ask dd 'say 'hello)
(hello hello)
> (ask dd 'say '(she said))
(she said she said)
```

Besides having a variable for each instance, it's also possible to have variables that are shared by every instance of the same class:

```
;;;;;                          In file cs61a/lectures/3.0/demo2.scm
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0))
  (method (next)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))

> (define c1 (instantiate counter))
> (ask c1 'next)
(1 1)
> (ask c1 'next)
(2 2)
> (define c2 (instantiate counter))
> (ask c2 'next)
(1 3)
> (ask c1 'next)
(3 4)
```

Now each `next` message tells us both the count for this particular counter and the overall count for all counters combined.

To understand the idea of inheritance, we'll first define a `person` class that knows about talking in various ways, and then define a `pigger` class that's just like a `person` except for talking in Pig Latin:

```
;;;;;                       In file cs61a/lectures/3.0/demo2.scm
(define-class (person name)
  (method (say stuff) stuff)
  (method (ask stuff) (ask self 'say (se '(would you please) stuff)))
  (method (greet) (ask self 'say (se '(hello my name is) name))) )

> (define marc (instantiate person 'marc))
> (ask marc 'say '(good morning))
(good morning)
> (ask marc 'ask '(open the door))
(would you please open the door)
> (ask marc 'greet)
(hello my name is marc)
```

Notice that an object can refer to itself by the name self; this is an automatically-created instance variable in every object whose value is the object itself. (We'll see when we look below the line that there are some complications about making this work.)

```
;;;;;                       In file cs61a/lectures/3.0/demo2.scm
(define-class (pigger name)
  (parent (person name))
  (method (pigl wd)
    (if (member? (first wd) '(a e i o u))
        (word wd 'ay)
        (ask self 'pigl (word (bf wd) (first wd))) ))
  (method (say stuff)
    (if (atom? stuff)
        (ask self 'pigl stuff)
        (map (lambda (w) (ask self 'pigl w)) stuff))) )

> (define porky (instantiate pigger 'porky))
> (ask porky 'say '(good morning))
(oodgay orningmay)
> (ask porky 'ask '(open the door))
(ouldway ouyay easeplay openay ethay oorday)
```

The crucial point here is that the pigger class doesn't have an ask method in its definition. When we ask porky to ask something, it uses the ask method in its parent (person) class.

Also, when the parent's ask method says (ask self 'say ...) it uses the say method from the pigger class, not the one from the person class. So Porky speaks Pig Latin even when asking something.

What happens when you send an object a message for which there is no method defined in its class? If the class has no parent, this is an error. If the class does have a parent, and the parent class understands the message, it works as we've seen here. But you might want to create a class that follows some rule of your own devising for unknown messages:

```
;;;;;                       In file cs61a/lectures/3.0/demo2.scm
(define-class (squarer)
  (default-method (* message message))
  (method (7) 'buzz) )

> (define s (instantiate squarer))
> (ask s 6)                > (ask s 7)                > (ask s 8)
36                         buzz                       64
```

Within the default method, the name `message` refers to whatever message was sent. (The name `args` refers to a list containing any additional arguments that were used.)

Let's say we want to maintain a list of all the instances that have been created in a certain class. It's easy enough to establish the list as a class variable, but we also have to make sure that each new instance automatically adds itself to the list. We do this with an `initialize` clause:

```
;;;;;                        In file cs61a/lectures/3.0/demo2.scm
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0) (counters '()))
  (initialize (set! counters (cons self counters)))
  (method (next)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))

> (define c1 (instantiate counter))
> (define c2 (instantiate counter))
> (ask counter 'counters)
(#<procedure> #<procedure>)
```

There was a bug in our `pigger` class definition; Scheme gets into an infinite loop if we ask Porky to `greet`, because it tries to translate the word `my` into Pig Latin but there are no vowels `aeiou` in that word. To get around this problem, we can redefine the `pigger` class so that its `say` method says every word in Pig Latin except for the word `my`, which it'll say using the usual method that `person`s who aren't `pigger`s use:

```
;;;;;                        In file cs61a/lectures/3.0/demo2.scm
(define-class (pigger name)
  (parent (person name))
  (method (pigl wd)
    (if (member? (first wd) '(a e i o u))
        (word wd 'ay)
        (ask self 'pigl (word (bf wd) (first wd))) ))
  (method (say stuff)
    (if (atom? stuff)
        (if (equal? stuff 'my) (usual 'say stuff) (ask self 'pigl stuff))
        (map (lambda (w) (ask self 'say w)) stuff))) )

> (define porky (instantiate pigger 'porky))
> (ask porky 'greet)
(ellohay my amenay isay orkypay)
```

(Notice that we had to create a new instance of the new class. Just doing a new define-class doesn't change any instances that have already been created in the old class. Watch out for this while you're debugging the OOP programming project.)

We invoke `usual` in the `say` method to mean "say this stuff in the usual way, the way that my parent class would use."

The OOP above-the-line section in the course reader talks about even more capabilities of the system, e.g., *multiple inheritance* with more than one parent class for a single child class.

CS 61A   Lecture Notes   Second Half of Week 4

Topic: Local state variables, environments

Reading: Abelson & Sussman, Section 3.1, 3.2; OOP below the line

We said the three big ideas in the OOP interface are message passing, local state, and inheritance. You know from section 2.4 how message passing is implemented below the line in Scheme, i.e., with a dispatch function that takes a message as argument and returns a method. For about a week, we're talking about how local state works.

A *local* variable is one that's only available within a particular part of the program; in Scheme this generally means within a particular procedure. We've used local variables before; `let` makes them. A *state* variable is one that remembers its value from one invocation to the next; that's the new part.

First of all let's look at *global* state—that is, let's try to remember some information about a computation but not worry about having separate versions for each object.

```
;;;;;                         In file cs61a/lectures/3.1/count1.scm
(define counter 0)

(define (count)
  (set! counter (+ counter 1))
  counter)

> (count)
1
> (count)
2
```

What's new here is the special form `set!` that allows us to change the value of a variable. This is not like `let`, which creates a temporary, local binding; this makes a permanent change in some variable that must have already existed. The syntax is just like `define` (but not the abbreviation for defining a function): it takes an unevaluated name and an expression whose value provides the new value.

A crucial thing to note about `set!` is that the substitution model no longer works. We can't substitute the value of `counter` wherever we see the name `counter`, or we'll end up with

```
(set! 0 (+ 0 1))
0
```

which doesn't make any sense. From now on we use a model of variables that's more like what you learned in 7th grade, in which a variable is a shoebox in which you can store some value. The difference from the 7th grade version is that we can have several shoeboxes with the same name (the instance variables in the different objects, for example) and we have to worry about how to keep track of that. Section 3.2 of A&S explains the *environment* model that keeps track for us.

Another new thing is that a procedure body can include more than one expression. In functional programming, the expressions don't *do* anything except compute a value, and a function can only return one value, so it doesn't make sense to have more than one expression in it. But when we invoke `set!` there is an *effect* that lasts beyond the computation of that expression, so now it makes sense to have that expression and then another expression that does something else. When a body has more than one expression, the expressions are evaluated from left to right (or top to bottom) and the value returned by the procedure is the value computed by the last expression. All but the last are just *for effect.*

We've seen how to have a global state variable. We'd like to try for *local* state variables. Here's an attempt that doesn't work:

```
;;;;;                        In file cs61a/lectures/3.1/count.lose
(define (count)
  (let ((counter 0))                    > (count)
    (set! counter (+ counter 1))        1
    counter))                           > (count)
                                        1
                                        > (count)
                                        1
```

It was a good idea to use `let`, because that's a way we know to create local variables. But `let` creates a *new* local variable each time we invoke it. Each call to `count` creates a new `counter` variable whose value is 0.

The secret is to find a way to call `let` only once, when we *create* the `count` function, instead of calling `let` every time we *invoke* `count`. Here's how:

```
;;;;;                        In file cs61a/lectures/3.1/count2.scm
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

Notice that there are no parentheses around the word `count` on the first line! Instead of

```
(define count (lambda () (let ...)))
```

(which is what the earlier version means) we have essentially interchanged the `lambda` and the `let` so that the former is inside the latter:

```
(define count (let ... (lambda () ...)))
```

We'll have to examine the environment model in detail before you can really understand why this works. A handwavy explanation is that the `let` creates a variable that's available to things in the body of the `let`; the `lambda` is in the body of the `let`; and so the variable is available to the function that the `lambda` creates.

The reason we wanted local state variables was so that we could have more than one of them. Let's take that step now. Instead of having a single procedure called `count` that has a single local state variable, we'll write a procedure `make-count` that, each time you call it, makes a new counter.

```
;;;;;                        In file cs61a/lectures/3.1/count3.scm

(define (make-count)                 > (define dracula (make-count))
  (let ((result 0))                  > (dracula)
    (lambda ()                       1
      (set! result (+ result 1))     > (dracula)
      result)))                      2
                                     > (define monte-cristo (make-count))
                                     > (monte-cristo)
                                     1
                                     > (dracula)
                                     3
```

Each of `dracula` and `monte-cristo` is the result of evaluating the expression `(lambda () ...)` to produce a procedure. Each of those procedures has access to its own local state variable called `result`. `Result` is temporary with respect to `make-count` but permanent with respect to `dracula` or `monte-cristo`, because the `let` is inside the `lambda` for the former but outside the `lambda` for the latter.

• Environment model of evaluation.

For now we're just going to introduce the central issues about environments, leaving out a lot of details. You'll get those next time.

The question is, what happens when you invoke a procedure? For example, suppose we've said

```
(define (square x) (* x x))
```

and now we say `(square 7)`; what happens? The substitution model says

1. Substitute the actual argument value(s) for the formal parameter(s) in the body of the function;

2. Evaluate the resulting expression.

In this example, the substitution of 7 for `x` in `(* x x)` gives `(* 7 7)`. In step 2 we evaluate that expression to get the result 49.

We now forget about the substitution model and replace it with the environment model:

1. Create a *frame* with the formal parameter(s) *bound to* the actual argument values;

2. Use this frame to extend the lexical environment;

3. Evaluate the body (without substitution!) in the resulting environment.

A frame is a collection of name-value associations or *bindings*. In our example, the frame has one binding, from `x` to `7`.

Skip step 2 for a moment and think about step 3. The idea is that we are going to evaluate the expression `(* x x)` but we are refining our notion of what it means to evaluate an expression. Expressions are no longer evaluated in a vacuum, but instead, every evaluation must be done with respect to some environment—that is, some collection of bindings between names and values. When we are evaluating `(* x x)` and we see the symbol `x`, we want to be able to look up `x` in our collection of bindings and find the value `7`.

Looking up the value bound to a symbol is something we've done before with global variables. What's new is that instead of one central collection of bindings we now have the possibility of *local* environments. The symbol `x` isn't always 7, only during this one invocation of `square`. So, step 3 means to evaluate the expression in the way that we've always understood, but looking up names in a particular place.

What's step 2 about? The point is that we can't evaluate `(* x x)` in an environment with nothing but the `x/7` binding, because we also have to look up a value for the symbol `*` (namely, the multiplication function). So, we create a new frame in step 1, but that frame isn't an environment by itself. Instead we use the new frame to *extend* an environment that already existed. That's what step 2 says.

Which old environment do we extend? In the `square` example there is only one candidate, the *global* environment. But in more complicated situations there may be several environments available. For example:

```
(define (f x)
  (define (g y)
    (+ x y))
  (g 3))

> (f 5)
```

When we invoke f, we create a frame (call it F1) in which x is bound to 5. We use that frame to extend the global environment (call it G), creating a new environment E1. Now we evaluate the body of f, which contains the internal definition for g and the expression (g 3). To invoke g, we create a frame in which y is bound to 3. (Call this frame F2.) We are going to use F2 to extend some old environment, but which? G or E1? The body of g is the expression (+ x y). To evaluate that, we need an envoironment in which we can look up all of + (in G), x (in F1), and y (in F2). So we'd better make our new environment by extending E1, not by extending G.

The example with f and g shows, in a very simple way, why the question of multiple environments comes up. But it still doesn't show us the full range of possible rules for choosing an environment. In the f and g example, the environment where g is defined is the same as the environment from which it's invoked. But that doesn't always have to be true:

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(define 3+ (make-adder 3))

(define n 7)

> (3+ n)
```

When we invoke make-adder, we create the environment E1 in which n is bound to 3. In the global environment G, we bind n to 7. When we evaluate the expression (3+ n), what environment are we in? What value does n have in this expression? Surely it should have the value 7, the global value. So we evaluate expressions that you type in G. When we invoke 3+ we create the frame F2 in which x is bound to 7. (Remember, 3+ is the function that was created by the lambda inside make-adder.

We are going to use F2 to extend some environment, and in the resulting environment we'll evaluate the body of 3+, namely (+ x n). What value should n have in this expression? It had better have the value 3 or we've defeated the purpose of make-adder. Therefore, the rule is that we do *not* extend the *current* environment at the time the function is invoked, which would be G in this case. Rather, we extend the environment in which the function was *created*, i.e., the environment in which we evaluated the lambda expression that created it. In this case that's E1, the environment that was created for the invocation of make-adder.

Scheme's rule, in which the procedure's defining environment is extended, is called *lexical* scope. The other rule, in which the current environment is extended, is called *dynamic* scope. We'll see in project 4 that a language with dynamic scope is possible, but it would have different features from Scheme.

Remember why we needed the environment model: We want to understand local state variables. The mechanism we used to create those variables was

```
(define some-procedure
  (let ((state-var initial-value))
    (lambda (...) ...)))
```

Roughly speaking, the let creates a frame with a binding for state-var. Within that environment, we evaluate the lambda. This creates a procedure within the scope of that binding. Every time that procedure is invoked, the environment where it was created—that is, the environment with state-var bound—is extended to form the new environment in which the body is evaluated. These new environments come and go, but the state variable isn't part of the new frames; it's part of the frame in which the procedure was defined. That's why it sticks around.

- Here are the complete rules for the environment model:

  Every expression is either an atom or a list.

  At any time there is a *current frame*, initially the global frame.

I. Atomic expressions.

   A. Numbers, strings, `#T`, and `#F` are self-evaluating.

   B. If the expression is a symbol, find the *first available* binding. (That is, look in the current frame; if not found there, look in the frame "behind" the current frame; and so on until the global frame is reached.)

II. Compound expressions (lists).

If the car of the expression is a symbol that names a special form, then follow its rules (II.B below). Otherwise the expression is a procedure invocation.

A. Procedure invocation.

Step 1: Evaluate all the subexpressions (using these same rules).

Step 2: Apply the procedure (the value of the first subexpression) to the arguments (the values of the other subexpressions).

(a) If the procedure is compound (user-defined):

a1: Create a frame with the formal parameters of the procedure bound to the actual argument values.

a2: Extend the procedure's defining environment with this new frame.

a3: Evaluate the procedure body, using the new frame as the current frame.
*** ONLY COMPOUND PROCEDURE INVOCATION CREATES A FRAME ***

(b) If the procedure is primitive:

Apply it by magic.

B. Special forms.

1. `Lambda` creates a procedure. The left circle points to the text of the `lambda` expression; the right circle points to the defining environment, i.e., to the current environment at the time the `lambda` is seen.
   *** ONLY `LAMBDA` CREATES A PROCEDURE ***
2. `Define` adds a *new* binding to the *current frame*.
3. `Set!` changes the *first available* binding (see I.B for the definition of "first available").
4. `Let` = `lambda` (II.B.1) + invocation (II.A)
5. `(define (...)  ...)` = `lambda` (II.B.1) + `define` (II.B.2)
6. Other special forms follow their own rules (`cond`, `if`).

• Environments and OOP.

Class and instance variables are both local state variables, but in different environments:

```
;;;;;                          In file cs61a/lectures/3.2/count4.scm
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda ()
          (set! loc (+ loc 1))
          (set! glob (+ glob 1))
          (list loc glob))))))
```

The class variable glob is created in an environment that surrounds the creation of the outer lambda, which represents the entire class. The instance variable loc is created in an environment that's inside the class lambda, but outside the second lambda that represents an instance of the class.

The example above shows how environments support state variables in OOP, but it's simplified in that the instance is not a message-passing dispatch procedure. Here's a slightly more realistic version:

```
;;;;;                          In file cs61a/lectures/3.2/count5.scm
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda (msg)
          (cond ((eq? msg 'local)
                 (lambda ()
                   (set! loc (+ loc 1))
                   loc))
                ((eq? msg 'global)
                 (lambda ()
                   (set! glob (+ glob 1))
                   glob))
                (else (error "No such method" msg)) ))))))
```

The structure of alternating lets and lambdas is the same, but the inner lambda now generates a dispatch procedure. Here's how we say the same thing in OOP notation:

```
;;;;;                          In file cs61a/lectures/3.2/count6.scm
(define-class (count)
  (class-vars (glob 0))
  (instance-vars (loc 0))
  (method (local)
    (set! loc (+ loc 1))
    loc)
  (method (global)
    (set! glob (+ glob 1))
    glob))
```

CS 61A Lecture Notes First Half of Week 5

Topic: Mutable data, queues, tables, vectors

Reading: Abelson & Sussman, Section 3.3.1–3

Play the animal game:

```
> (load "lectures/3.3/animal.scm")
#f
> (animal-game)
Does it have wings? no
Is it a rabbit? no

I give up, what is it? gorilla

Please tell me a question whose answer is YES for a gorilla
and NO for a rabbit.
Enclose the question in quotation marks.
"Does it have long arms?"
"Thanks.  Now I know better."
> (animal-game)
Does it have wings? no
Does it have long arms? no
Is it a rabbit? yes
"I win!"
```

The crucial point about this program is that its behavior changes each time it learns about a new animal. Such *learning* programs have to modify a data base as they run. We represent the animal game data base as a tree; we want to be able to splice a new branch into the tree (replacing what used to be a leaf node).

Changing what's in a data structure is called *mutation.* Scheme provides primitives `set-car!` and `set-cdr!` for this purpose.

They aren't special forms! The pair that's being mutated must be located by computing some expression. For example, to modify the second element of a list:

```
(set-car! (cdr lst) 'new-value)
```

They're different from `set!`, which changes the binding of a variable. We use them for different purposes, and the syntax is different. Still, they are connected in two ways: (1) Both make your program non-functional, by making a permanent change that can affect later procedure calls. (2) Each can be implemented in terms of the other; the book shows how to use local state variables to simulate mutable pairs, and later we'll see how the Scheme interpreter uses mutable pairs to implement environments, including the use of `set!` to change variable values.

The only purpose of mutation is efficiency. In principle we could write the animal game functionally by recopying the entire data base tree each time, and using the new one as an argument to the next round of the game. But the saving can be quite substantial.

Identity. Once we have mutation we need a subtler view of the idea of equality. Up to now, we could just say that two things are equal if they look the same. Now we need *two* kinds of equality, that old kind plus a new one: Two things are *identical* if they are the very same thing, so that mutating one also changes the other. Example:

```
> (define a (list 'x 'y 'z))
> (define b (list 'x 'y 'z))
> (define c a)
> (equal? b a)
#T
> (eq? b a)
#F
> (equal? c a)
#T
> (eq? c a)
#T
```

The two lists a and b are equal, because they print the same, but they're not identical. The lists a and c are identical; mutating one will change the other:

```
> (set-car! (cdr a) 'foo)
> a
(X FOO Z)
> b
(X Y Z)
> c
(X FOO Z)
```

If we use mutation we have to know what shares storage with what. For example, (cdr a) shares storage with a. (Append a b) shares storage with b but not with a. (Why not? Read the append procedure.)

The Scheme standard says you're not allowed to mutate quoted constants. That's why I said (list 'x 'y 'z) above and not '(x y z). The text sometimes cheats about this. The reason is that Scheme implementations are allowed to share storage when the same quoted constant is used twice in your program.

Here's the animal game:

```
;;;;;                        In file cs61a/lectures/3.3/animal.scm
(define (animal node)
  (define (type l) (car l))
  (define (question l) (cadr l))
  (define (yespart l) (caddr l))
  (define (nopart l) (cadddr l))
  (define (answer l) (cadr l))
  (define (leaf? l) (eq? (type l) 'leaf))
  (define (branch? l) (eq? (type l) 'branch))
  (define (set-yes! node x)
    (set-car! (cddr node) x))
  (define (set-no! node x)
    (set-car! (cdddr node) x))

  (define (yorn)
    (let ((yn (read)))
      (cond ((eq? yn 'yes) #t)
            ((eq? yn 'no) #f)
            (else (display "Please type YES or NO")
                  (yorn)))))
  (display (question node))
  (display " ")
  (let ((yn (yorn)) (correct #f) (newquest #f))
    (let ((next (if yn (yespart node) (nopart node))))
      (cond ((branch? next) (animal next))
            (else (display "Is it a ")
                  (display (answer next))
                  (display "? ")
                  (cond ((yorn) "I win!")
                        (else (newline)
                              (display "I give up, what is it? ")
                              (set! correct (read))
                              (newline)
                               (display "Please tell me a question whose answer ")
                              (display "is YES for a ")
                              (display correct)
                              (newline)
                              (display "and NO for a ")
                              (display (answer next))
                              (display ".")
                              (newline)
                              (display "Enclose the question in quotation marks.")
                              (newline)
                              (set! newquest (read))
                              (if yn
                                  (set-yes! node (make-branch newquest
                                                           (make-leaf correct)
                                                           next))
                                  (set-no! node (make-branch newquest
                                                          (make-leaf correct)
                                                          next)))
                              "Thanks.  Now I know better.")))))))

(define (make-branch q y n)
  (list 'branch q y n))

(define (make-leaf a)
  (list 'leaf a))

(define animal-list
  (make-branch "Does it have wings?"
               (make-leaf 'parrot)
               (make-leaf 'rabbit)))


(define (animal-game) (animal animal-list))
```

Things to note: Even though the main structure of the program is sequential and BASIC-like, we haven't abandoned data abstraction. We have constructors, selectors, and *mutators*—a new idea—for the nodes of the game tree.

• Tables. We're now ready to understand how to implement the put and get procedures that A&S used at the end of chapter 2. A table is a list of key-value pairs, with an extra element at the front just so that adding the first entry to the table will be no diffferent from adding later entries. (That is, even in an "empty" table we have a pair to set-cdr!)

```
;;;;;                         In file cs61a/lectures/3.3/table.scm
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        #f
        (cdr record))))

(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr! the-table
                  (cons (cons key value)
                        (cdr the-table)))
        (set-cdr! record value)))
  'ok)

(define the-table (list '*table*))
```

Assoc is in the book:

```
(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records))) ))
```

In chapter 2, A&S provided a single, global table, but we can generalize this in the usual way by taking an extra argument for which table to use. That's how lookup and insert! work.

One little detail that always confuses people is why, in creating two-dimensional tables, we don't need a *table* header on each of the subtables. The point is that lookup and insert! don't pay any attention to the car of that header pair; all they need is to represent a table by *some* pair whose cdr points to the actual list of key-value pairs. In a subtable, the key-value pair from the top-level table plays that role. That is, the entire subtable is a value of some key-value pair in the main table. What it means to be "the value of a key-value pair" is to be the cdr of that pair. So we can think of that pair as the header pair for the subtable.

• Memoization. Exercise 3.27 is a pain in the neck because it asks for a very complicated environment diagram, but it presents an extremely important idea. If we take the simple Fibonacci number program:

```
;;;;;                        In file cs61a/lectures/3.3/fib.scm
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

we recall that it takes $O(2^n)$ time because it ends up doing a lot of subproblems redundantly. For example, if we ask for `(fib 5)` we end up computing `(fib 3)` twice. We can fix this by *remembering* the values that we've already computed. The book's version does it by entering those values into a local table. It may be simpler to understand this version, using the global get/put:

```
;;;;;                        In file cs61a/lectures/3.3/fib.scm
(define (fast-fib n)
  (if (< n 2)
      n                               ; base case unchanged
      (let ((old (get 'fib n)))
        (if (number? old)             ; do we already know the answer?
            old
            (begin                    ; if not, compute and learn it
             (put 'fib n (+ (fast-fib (- n 1))
                            (fast-fib (- n 2))))
             (get 'fib n))))))
```

Is this functional programming? That's a more subtle question than it seems. Calling `memo-fib` makes a permanent change in the environment, so that a second call to `memo-fib` with the same argument will carry out a very different (and much faster) process. But the new process will get the same answer! If we look inside the box, `memo-fib` works non-functionally. But if we look only at its input-output behavior, `memo-fib` *is* a function because it always gives the same answer when called with the same argument.

What if we tried to memoize `random`? It would be a disaster; instead of getting a random number each time, we'd get the same number repeatedly! Memoization only makes sense if the underlying function really *is* functional.

This idea of using a non-functional implementation for something that has functional behavior will be very useful later in the course when we look at streams.

• **Vectors** So far we have seen one primitive data aggregation mechanism: the pair. We use linked pairs to represent *sequences* (an abstract type) in the form of *lists.*

The list suffers from one important weakness: Finding the nth element of a list takes time $O(n)$ because you have to call `cdr` $n-1$ times. Scheme, like most programming languages, also provides a primitive aggregation mechanism without this weakness. In Scheme it's called a *vector*; in many other languages it's called an *array,* but it's the same idea. Finding the nth element of a vector takes $O(1)$ time.

• **Vector primitives**

Some of the procedures for vectors are exact analogs to procedures for lists:

```
    (vector a b c d ...)              (list a b c d ...)
    (vector-ref vec n)                (list-ref lst n)
    (vector-length vec)               (length lst)
```

Most notably, the *selector* for vectors, `vector-ref`, is just like the selector for lists (except that it's faster).

What about constructors? There's a `vector` procedure, just like the `list` procedure, that's good for situations in which you know exactly how many elements the sequence will have, and all of the element values, all at once. But there are no vector analogs to the list constructors `cons` and `append`, which are useful for *extending* lists. In particular, `cons` is the workhorse of recursive list processing procedures; we'll see that vector processing is done quite differently.

The weakness of vectors is that they can't be extended. You have to know the length of the vector when you create it. So instead of `cons` and `append` we have

```
(make-vector len)
```

which creates a vector of length `len`, in which the element values are unspecified. (You then use mutation, discussed below, to fill in the desired values.) Alternatively, if you want to create a vector in which every element has the same initial value, you can say

```
(make-vector len value)
```

Because vectors are created all at once, rather than one element at a time, *mutation* is crucial to any useful vector program. The primitive mutator for vectors is

```
(vector-set! vec n value)
```

This procedure is comparable to `set-car!` and `set-cdr!` for pairs. (It's interesting to note that Scheme doesn't provide a mutator for the nth element of a list; this is because most list processing is done using functional programming style, and pair mutation is mainly for special cases such as tables.)

The printed format of a vector is

```
#(a b c d)
```

You can quote this to include a constant vector in a program. (Note: In STk, vectors are self-evaluating, so you can omit the quotation mark, but this is a nonstandard extension to Scheme.)

Scheme also provides functions `list->vector` and `vector->list` that let you convert between the two sequence implementations.

- **Vector programming style**

Let's write a mapping function for vectors; it will take a function and a vector as arguments, and return a vector.

For reference, here's the `map` function for lists:

```
(define (map fn lst)
  (if (null? lst)
      '()
      (cons (fn (car lst))
        (map fn (cdr lst)))))
```

To do the same task for vectors, we must first create a new vector of the same length as the argument vector, then fill in the values using mutation:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (vector-map fn vec)
  (define (loop newvec n)
    (if (< n 0)
    newvec
    (begin (vector-set! newvec n (fn (vector-ref vec n)))
           (loop newvec (- n 1)))))
  (loop (make-vector (vector-length vec)) (- (vector-length vec) 1)))
```

This is a lot more complicated! It requires a helper procedure, and an extra *index variable*, `n`, to keep track of the element number within the vector. By contrast, the list version of `map` never actually knows how long its argument list is.

- **Strengths and weaknesses**

Of course, if we wanted, we could write our own equivalent to `cons` for vectors:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (vector-cons value vec)
  (define (loop newvec n)
    (if (= n 0)
    (begin (vector-set! newvec n value)
           newvec)
    (begin (vector-set! newvec n (vector-ref vec (- n 1)))
           (loop newvec (- n 1)))))
  (loop (make-vector (+ (vector-length vec) 1)) (vector-length vec)))
```

If we wrote similar procedures `vector-car` and `vector-cdr`, we could then write `vector-map` in a style exactly like `map`. But this would be a bad idea, because our `vector-cons` requires $O(n)$ time to copy the elements from the old vector to the new one.

operation	**lists**	**vectors**
nth element	`list-ref`, $O(n)$	`vector-ref`, $O(1)$
add new element	`cons`, $O(1)$	`vector-cons`, $O(n)$

This is why there isn't one best way to represent sequences. Lists are faster (and allow for cleaner code) at adding elements, but vectors are faster at selecting arbitrary elements.

(Note, though, that if you want to select *all* the elements of a sequence, one after another, then lists are just as fast as arrays. It's only when you want to jump around within the sequence that arrays are faster.)

- **Example: Shuffling**

Suppose we want to shuffle a deck of cards — we want to reorder the cards randomly. We'll look at three solutions to this problem.

First, here's a solution using functional programming with lists. Because we aren't allowing mutation of pairs, this version does a lot of recopying:

```
;;;;;                        In file cs61a/lectures/vector.scm
(define (shuffle1 lst)
  (define (loop in out n)
    (if (= n 0)
    (cons (car in) (shuffle1 (append (cdr in) out)))
    (loop (cdr in) (cons (car in) out) (- n 1))))
  (if (null? lst)
      '()
      (loop lst '() (random (length lst)))))
```

This is a case in which functional programming has few virtues. The code is hard to read, and it takes $O(n^2)$ time to shuffle a list of length n. (There are n recursive calls to `shuffle1`, each of which calls the $O(n)$ primitives `append` and `length` as well as $O(n)$ calls to the helper function `loop`.)

We can improve things using list mutation. Any list-based solution will still be $O(n^2)$, because it takes $O(n)$ time to find one element at a randomly chosen position, and we have to do that n times. But we can improve the constant factor by avoiding the copying of pairs that `append` does in the first version:

```
;;;;;                        In file cs61a/lectures/vector.scm
(define (shuffle2! lst)
  (if (null? lst)
      '()
      (let ((index (random (length lst))))
    (let ((pair ((repeated cdr index) lst))
          (temp (car lst)))
      (set-car! lst (car pair))
      (set-car! pair temp)
      (shuffle2! (cdr lst))
      lst))))
```

(Note: This could be improved still further by calling `length` only once, and using a helper procedure to subtract one from the length in each recursive call. But that would make the code more complicated, so I'm not bothering. You can take it as an exercise if you're interested.)

Vectors allow a more dramatic speedup, because finding each element takes $O(1)$ instead of $O(n)$:

```
;;;;;                        In file cs61a/lectures/vector.scm
(define (shuffle3! vec)
  (define (loop n)
    (if (= n 0)
    vec
    (let ((index (random n))
          (temp (vector-ref vec (- n 1))))
      (vector-set! vec (- n 1) (vector-ref vec index))
      (vector-set! vec index temp)
      (loop (- n 1)))))
  (loop (vector-length vec)))
```

The total time for this version is $O(n)$, because it makes n recursive calls, each of which takes constant time.

- **How it works**

One handwavy paragraph on why vectors have the performance they do:

A pair is two pointers attached to each other in a single block of memory. A vector is similar, but it's a block of n pointers for an arbitrary (but fixed) number n. Since a vector is one contiguous block of memory, if you know the address of the beginning of the block, you can just add k to find the address of the kth element. The downside is that in order to get all the elements in a single block of memory, you have to allocate the block all at once.

If you don't understand that, don't worry about it until 61B.

CS 61A Lecture Notes Second Half of Week 5

Topic: Streams

Reading: Abelson & Sussman, Section 3.5.1-3, 3.5.5

Streams are an abstract data type, not so different from rational numbers, in that we have constructors and selectors for them. But we use a clever trick to achieve tremendously magical results. As we talk about the mechanics of streams, there are three big ideas to keep in mind:

- Efficiency: Decouple order of evaluation from the form of the program.
- Infinite data sets.
- Functional representation of time-varying information (versus OOP).

You'll understand what these all mean after we look at some examples.

How do we tell if a number n is prime? Never mind computers, how would you express this idea as a mathematician? Something like this: "N is prime if it has no factors in the range $2 \leq f < n$."

So, to implement this on a computer, we should

- Get all the numbers in the range $[2, n-1]$.
- See which of those are factors of n.
- See if the result is empty.

```
;;;;;                        In file cs61a/lectures/3.5/prime1.scm
(define (prime? n)
  (null? (filter (lambda (x) (= (remainder n x) 0))
                 (enumerate-interval 2 (- n 1)))))
```

But we don't usually program it that way. Instead, we write a *loop*:

```
;;;;;                        In file cs61a/lectures/3.5/prime0.scm
(define (prime? n)
  (define (iter factor)
    (cond ((= factor n) #t)
          ((= (remainder n factor) 0) #f)
          (else (iter (+ factor 1)))))
  (iter 2))
```

(Never mind that we can make small optimizations like only checking for factors up to $\sqrt{n}$. Let's keep it simple.)

Why don't we write it the way we expressed the problem in words? The problem is one of efficiency. Let's say we want to know if 1000 is prime. We end up constructing a list of 998 numbers and testing *all* of them as possible factors of 1000, when testing the first possible factor would have given us a false result quickly.

The idea of streams is to let us have our cake and eat it too. We'll write a program that *looks like* the first version, but *runs like* the second one. All we do is change the second version to use the stream ADT instead of the list ADT:

```
;;;;;                          In file cs61a/lectures/3.5/prime2.scm
(define (prime? n)
  (stream-null? (stream-filter (lambda (x) (= (remainder n x) 0))
                               (stream-enumerate-interval 2 (- n 1)))))
```

The only changes are `stream-enumerate-interval` instead of `enumerate-interval`, `stream-null?` instead of `null?`, and `stream-filter` instead of `filter`.

How does it work? A list is implemented as a pair whose `car` is the first element and whose `cdr` is the rest of the elements. A stream is almost the same: It's a pair whose `car` is the first element and whose `cdr` is a *promise* to compute the rest of the elements later.

For example, when we ask for the range of numbers $[2, 999]$ what we get is a single pair whose `car` is 2 and whose `cdr` is a promise to compute the range $[3, 999]$. The function `stream-enumerate-interval` returns that single pair. What does `stream-filter` do with it? Since the first number, 2, does satisfy the predicate, `stream-filter` returns a single pair whose `car` is 2 and whose `cdr` is a promise *to filter* the range $[3, 999]$. `Stream-filter` returns that pair. So far no promises have been "cashed in." What does `stream-null?` do? It sees that its argument stream contains the number 2, and maybe contains some more stuff, although maybe not. But at least it contains the number 2, so it's not empty. `Stream-null?` returns `#f` right away, without computing or testing any more numbers.

Sometimes (for example, if the number we're checking *is* prime) you do have to cash in the promises. If so, the stream program still follows the same order of events as the original loop program; it tries one number at a time until either a factor is found or there are no more numbers to try.

Summary: What we've accomplished is to decouple the form of a program—the order in which computations are presented—from the actual order of evaluation. This is one more step on the long march that this whole course is about, i.e., letting us write programs in language that reflects the problems we're trying to solve instead of reflecting the way computers work.

• Implementation. How does it work? The crucial point is that when we say something like

```
(cons-stream from (stream-enumerate-interval (+ from 1) to))
```

(inside `stream-enumerate-interval`) we can't actually evaluate the second argument to `cons-stream`. That would defeat the object, which is to defer that evaluation until later (or maybe never). Therefore, `cons-stream` has to be a special form. It has to `cons` its first argument onto a promise to compute the second argument. The expression

```
(cons-stream a b)
```

is equivalent to

```
(cons a (delay b))
```

`Delay` is itself a special form, the one that constructs a promise. Promises could be a primitive data type, but since this is Scheme, we can represent a promise as a function. So the expression

```
(delay b)
```

really just means

```
(lambda () b)
```

We use the promised expression as the body of a function with no arguments. (A function with no arguments is called a *thunk*.)

Once we have this mechanism, we can use ordinary functions to redeem our promises:

```
(define (force promise) (promise))
```

and now we can write the selectors for streams:

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

Notice that forcing a promise doesn't compute the entire rest of the job at once, necessarily. For example, if we take our range $[2, 999]$ and ask for its tail, we don't get a list of 997 values. All we get is a pair whose `car` is 3 and whose `cdr` is a new promise to compute $[4, 999]$ later.

The name for this whole technique is *lazy evaluation* or *call by need*.

• Reordering and functional programming. Suppose your program is written to include the following sequence of steps:

```
...
(set! x 2)
...
(set! y (+ x 3))
...
(set! x 7)
...
```

Now suppose that, because we're using some form of lazy evaluation, the actual sequence of events is reordered so that the third `set!` happens before the second one. We'll end up with the wrong value for `y`. This example shows that we can only get away with below-the-line reordering if the above-the-line computation is functional.

(Why isn't it a problem with `let`? Because `let` doesn't mutate the value of one variable in one environment. It sets up a local environment, and any expression within the body of the `let` has to be computed within that environment, even if things are reordered.)

• Infinite streams. Think about the plain old list function

```
(define (enumerate-interval from to)
  (if (> from to)
      '()
      (cons from (enumerate-interval (+ from 1) to)) ))
```

When we change this to a stream function, we change very little in the appearance of the program:

```
(define (stream-enumerate-interval from to)
  (if (> from to)
      THE-EMPTY-STREAM
      (cons-STREAM from (stream-enumerate-interval (+ from 1) to)) ))
```

but this tiny above-the-line change makes an enormous difference in the actual behavior of the program.

Now let's cross out the second argument and the end test:

```
(define (stream-enumerate-interval from)
  (cons-stream from (stream-enumerate-interval (+ from 1))) )
```

This is an *enormous* above-the-line change! We now have what looks like a recursive function with no base case—an infinite loop. And yet there is hardly any difference at all in the actual behavior of the program. The old version computed a range such as $[2, 999]$ by constructing a single pair whose `car` is 2 and whose `cdr` is a promise to compute $[3, 999]$ later. The new version computes a range such as $[2, \infty]$ by constructing a single pair whose `car` is 2 and whose `cdr` is a promise to compute $[3, \infty]$ later!

This amazing idea lets us construct even some pretty complicated infinite sets, such as the set of all the prime numbers. (Explain the sieve of Eratosthenes. The program is in the book so it's not reproduced here.)

• Time-varying information. Functional programming works great for situations in which we are looking for a timeless answer to some question. That is, the same question always has the same answer regardless of events in the world. We invented OOP because functional programming didn't let us model changing state. But with streams we *can* model state functionally. We can say

```
(define (user-stream)
  (cons-stream (read) (user-stream)) )
```

and this gives us *the stream of everything the user is going to type* from now on. Instead of using local state variables to remember the effect of each thing the user types, one at a time, we can write a program that computes the result of the (possibly infinite) collection of user requests all at once! This feels really bizarre, but it does mean that purely functional programming languages can handle user interaction. We don't *need* OOP.

CS 61A Lecture Notes First Half of Week 6

Topic: Metacircular evaluator

Reading: Abelson & Sussman, Section 4.1.1–6

We're going to investigate a Scheme interpreter written in Scheme. This interpreter implements the environment model of evaluation.

Why bother? What good is an interpreter for Scheme that we can't use unless we already have another interpreter for Scheme?

- It helps you understand the environment model.
- It lets us experiment with modifications to Scheme (new features).
- Even real Scheme interpreters are largely written in Scheme.
- It illustrates a big idea: *universality.*

Universality means we can write *one program* that's equivalent to all other programs. At the hardware level, this is the idea that made general-purpose computers possible. It used to be that they built a separate machine, from scratch, for every new problem. An intermediate stage was a machine that had a *patchboard* so you could rewire it, effectively changing it into a different machine for each problem, without having to re-manufacture it. The final step was a single machine that accepted a program *as data* so that it can do any problem without rewiring.

Instead of a function machine that computes a particular function, taking (say) a number in the input hopper and returning another number out the bottom, we have a *universal* function machine that takes *a function machine* in one input hopper, and a number in a second hopper, and returns whatever number the input machine would have returned. This is the ultimate in data-directed programming.

Our Scheme interpreter leaves out some of the important components of a real one. It gets away with this by taking advantage of the capabilities of the underlying Scheme. Specifically, we don't deal with storage allocation, tail recursion elimination, or implementing any of the Scheme primitives. All we *do* deal with is the evaluation of expressions. That turns out to be quite a lot in itself, and pretty interesting.

Here is a one-screenful version of the metacircular evaluator with most of the details left out:

```
;;;;;                        In file cs61a/lectures/4.1/micro.scm
(define (scheme)
  (display "> ")
  (print (eval (read) the-global-environment))
  (scheme) )

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((symbol? exp) (lookup-in-env exp env))
        ((special-form? exp) (do-special-form exp env))
        (else (apply (eval (car exp) env)
                     (map (lambda (e) (eval e env)) (cdr exp)) ))))

(define (apply proc args)
  (if (primitive? proc)
      (do-magic proc args)
      (eval (body proc)
            (extend-environment (formals proc)
                                args
                                (proc-env proc) ))))
```

Although the version in the book is a lot bigger, this really does capture the essential structure, namely, a mutual recursion between `eval` (evaluate an expression relative to an environment) and `apply` (apply a function to arguments). To evaluate a compound expression means to evaluate the subexpressions recursively, then apply the `car` (a function) to the `cdr` (the arguments). To apply a function to arguments means to evaluate the body of the function in a new environment.

What's left out? Primitives, special forms, and a lot of details.

In that other college down the peninsula, they wouldn't consider you ready for an interpreter until junior or senior year. At this point in the introductory course, they'd still be teaching you where the semicolons go. How do we get away with this? We have two big advantages:

- The *source language* (the language that we're interpreting) is simple and uniform. Its entire formal syntax can be described in one page, as we did in week 5. There's hardly anything to implement!
- The *implementation language* (the one in which the interpreter itself is written) is powerful enough to handle a program as data, and to let us construct data structures that are both hierarchical and circular.

The amazing thing is that the simple source language and the powerful implementation language are both Scheme! You might think that a powerful language has to be complicated, but it's not so.

• Introduction to Logo. For the programming project you're turning the metacircular evaluator into an interpreter for a *different* language, Logo. To do that you should know a little about Logo itself.

Logo is a dialect of Lisp, just as Scheme is, but its design has different priorities. The goal was to make it as natural-seeming as possible for kids. That means things like getting rid of all those parentheses, and that has other syntactic implications.

(To demonstrate Logo, run `~cs61a/logo` which is Berkeley Logo.)

Commands and operations: In Scheme, every procedure returns a value, even the ones for which the value is unspecified and/or useless, like `define` and `print`. In Logo, procedures are divided into operations, which return values, and commands, which don't return values but are called for their effect. You have to start each instruction with a command:

```
print sum 2 3
```

Syntax: If parentheses aren't used to delimit function calls, how do you know the difference between a function and an argument? When a symbol is used without punctuation, that means a function call. When you want the value of a variable to use as an argument, you put colon in front of it.

```
make "x 14
print :x
print sum :x :x
```

Words are quoted just as in Scheme, except that the double-quote character is used instead of single-quote. But since expressions aren't represented as lists, the same punctuation that delimits a list also quotes it:

```
print [a b c]
```

(Parentheses *can* be used, as in Scheme, if you want to give extra arguments to something, or indicate infix precedence.)

```
print (sum 2 3 4 5)
print 3*(4+5)
```

No special forms: Except `to`, the thing that defines a new procedure, all Logo primitives evaluate their arguments. How is this possible? We "proved" back in chapter 1 that `if` has to be a special form. But instead we just quote the arguments to `ifelse`:

```
ifelse 2=3 [print "hi] [print "bye]
```

You don't notice the quoting since you get it for free with the list grouping.

Functions not first class: In Logo every function has a name; there's no `lambda`. Also, the namespace for functions is separate from the one for variables; a variable can't have a function as its value. (This is convenient because we can use things like `list` or `sentence` as formal parameters without losing the functions by those names.) That's another reason why you need colons for variables.

So how do you write higher-order functions like `map`? Two answers. First, you can use the *name* of a function as an argument, and you can use that name to construct an expression and eval it with `run`. Second, Logo has first-class *expressions*; you can `run` a list that you get as an argument. (This raises issues about the scope of variables that we'll explore next week.)

```
print map "first [the rain in spain]
print map [? * ?] [3 4 5 6]
```

• Data abstraction in the evaluator. Here is a quote from the Instructor's Manual, regarding section 4.1.2:

"Point out that this section is boring (as is much of section 4.1.3), and explain why: Writing the selectors, constructors, and predicates that implement a representation is often uninteresting. It is important to say explicitly what you expect to be boring and what you expect to be interesting so that students don't ascribe their boredom to the wrong aspect of the material and reject the interesting ideas. For example, data abstraction isn't boring, although writing selectors is. The details of representing expressions (as given in section 4.1.2) and environments (as given in section 4.1.3) are mostly boring, but the evaluator certainly isn't."

• Dynamic scope. Logo uses dynamic scope, which we discussed in Section 3.2, instead of Scheme's lexical scope. There are advantages and disadvantages to both approaches.

Summary of arguments for lexical scope:

- Allows local state variables (OOP).
- Prevents name "capture" bugs.
- Faster compiled code.

Summary of arguments for dynamic scope:

- Allows first-class expressions (WHILE).
- Easier debugging.
- Allows "semi-global" variables.

Lexical scope is required in order to make possible Scheme's approach to local state variables. That is, a procedure that has a local state variable must be defined within the scope where that variable is created, and must carry that scope around with it. That's exactly what lexical scope accomplishes.

On the other hand, (1) most lexically scoped languages (e.g., Pascal) don't have `lambda`, and so they can't give you local state variables despite their lexical scope. And (2) lexical scope is needed for local state variables only if you want to implement the latter in the particular way that we've used. Object Logo, for example, provides OOP without relying on `lambda` because it includes local state variables as a primitive feature.

Almost all computer scientists these days hate dynamic scope, and the reason they give is the one about name captures. That is, suppose we write procedure P that refers to a global variable V. Example:

```
(define (area rad)
  (* pi rad rad))
```

This is intended as a reference to a global variable `pi` whose value, presumably, is 3.141592654. But suppose we invoke it from within another procedure like this:

```
(define (mess-up pi)
  (area (+ pi 5)))
```

If we say `(mess-up 4)` we intend to find the area of a circle with radius 9. But we won't get the right area if we're using dynamic scope, because the name `pi` in procedure `area` suddenly refers to the local variable in `mess-up`, rather than to the intended global value.

This argument about naming bugs is particularly compelling to people who envision a programming project in which 5000 programmers work on tiny slivers of the project, so that nobody knows what anyone else is doing. In such a situation it's entirely likely that two programmers will happen to use the same name for different purposes. But note that we had to do something pretty foolish—using the name `pi` for something that isn't π at all—in order to get in trouble.

Lexical scope lets you write compilers that produce faster executable programs, because with lexical scope you can figure out during compilation exactly where in memory any particular variable reference will be. With dynamic scope you have to defer the name-location correspondence until the program actually runs. This is the real reason why people prefer lexical scope, despite whatever they say about high principles.

As an argument for dynamic scope, consider this Logo implementation of the `while` control structure:

```
to while :condition :action
if not run :condition [stop]
run :action
while :condition :action
end

to example :x
while [:x > 0] [print :x make "x :x-1]
end

? example 3
3
2
1
```

This wouldn't work with lexical scope, because within the procedure `while` we couldn't evaluate the argument expressions, because the variable `x` is not bound in any environment lexically surrounding `while`. Dynamic scope makes the local variables of `example` available to `while`. That in turn allows first-class expressions. (That's what Logo uses in place of first-class functions.)

There are ways to get around this limitation of lexical scope. If you wanted to write `while` in Scheme, basically, you'd have to make it a special form that turns into something using thunks. That is, you'd have to make

```
(while cond act)
```

turn into

```
(while-helper (lambda () cond) (lambda () act))
```

sort of like what we did for `cons-stream`. But the Logo point of view is that it's easier for a beginning programmer to understand first-class expressions than to understand special forms and thunks.

Most Scheme implementations include a debugger that allows you to examine the values of variables after an error. But, because of the complexity of the scope rules, the debugging language isn't Scheme itself. Instead you have to use a special language with commands like "switch to the environment of the procedure that called this one." In Logo, when an error happens you can *pause* your program and type ordinary Logo expressions in an environment in which all the relevant variables are available. For example, here is a Logo program:

```
;;;;;                        In file cs61a/lectures/4.1/bug.logo
to assq :thing :list
if equalp :thing first first :list [op last first :list]
op assq :thing bf :list
end

to spell :card
pr (se assq bl :card :ranks "of assq last :card :suits)
end

to hand :cards
if emptyp :cards [stop]
spell first :cards
hand bf :cards
end

make "ranks [[a ace] [2 two] [3 three] [4 four] [5 five] [6 six] [7 seven]
             [8 eight] [9 nine] [10 ten] [j jack] [q queen] [k king]]
make "suits [[h hearts] [s spades] [d diamonds] [c clubs]]

? hand [10h 2d 3s]
TEN OF HEARTS
TWO OF DIAMONDS
THREE OF SPADES
```

Suppose we introduce an error into `hand` by changing the recursive call to

```
hand first bf :cards
```

The result will be an error message in `assq`—two procedure calls down—complaining about an empty argument to `first`. Although the error is caught in `assq`, the real problem is in `hand`. In Logo we can say `pons`, which stands for "print out names," which means to show the values of *all* variables accessible at the moment of the error. This will include the variable `cards`, so we'll see that the value of that variable is a single card instead of a list of cards.

Finally, dynamic scope is useful for allowing "semi-global" variables. Take the metacircular evaluator as an example. Lots of procedures in it require `env` as an argument, but there's nothing special about the value of `env` in any one of those procedures. It's almost always just the current environment, whatever that happens to be. If Scheme had dynamic scope, `env` could be a parameter of `eval`, and it would then automatically be available to any subprocedure called, directly or indirectly, by `eval`. (This is the flip side of the name-capturing problem; in this case we *want* `eval` to capture the name `env`.)

• Environments as circular lists. When we first saw circular lists in chapter 2, they probably seemed to be an utterly useless curiosity, especially since you can't print one. But in the MC evaluator, every environment is a circular list, because the environment contains procedures and each procedure contains a pointer to the environment in which it's defined. So, moral number 1 is that circular lists are useful; moral number 2 is not to try to trace a procedure in the evaluator that has an environment as an argument! The tracing mechanism will take forever to try to print the circular argument list.

CS 61A Lecture Notes Second Half of Week 6

Topic: Lazy evaluator

Reading: Abelson & Sussman, Sections 4.2

To load the lazy metacircular evaluator, say

```
(load "~cs61a/lib/lazy.scm")
```

Streams require careful attention

To make streams of pairs, the text uses this procedure:

```
;;;;;                        In file cs61a/lectures/4.2/pairs.scm
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

In exercise 3.68, Louis Reasoner suggests this simpler version:

```
(define (pairs s t)
  (interleave
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (pairs (stream-cdr s) (stream-cdr t))))
```

Of course you know because it's Louis that this doesn't work. But why not? The answer is that `interleave` is an ordinary procedure, so its arguments are evaluated right away, including the recursive call. So there is an infinite recursion before any pairs are generated. The book's version uses `cons-stream`, which is a special form, and so what looks like a recursive call actually isn't—at least not right away.

But in principle Louis is right! His procedure does correctly specify what the desired result should contain. It fails because of a detail in the implementation of streams. In a perfect world, a mathematically correct program such as Louis's version ought to work on the computer.

In section 3.5.4 they solve a similar problem by making the stream programmer use explicit `delay` invocations. (You skipped over that section because it was about calculus.) Here's how Louis could use that technique:

```
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (interleave-delayed (force delayed-s2)
                           (delay (stream-cdr s1))))))

(define (pairs s t)
  (interleave-delayed
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (delay (pairs (stream-cdr s) (stream-cdr t)))))
```

This works, but it's far too horrible to contemplate; with this technique, the stream programmer has to check carefully every procedure to see what might need to be delayed explicitly. This defeats the object of an abstraction. The user should be able to write a stream program just as if it were a list program, without any idea of how streams are implemented!

Lazy evaluation: delay everything automatically

Back in chapter 1 we learned about *normal order evaluation*, in which argument subexpressions are not evaluated before calling a procedure. In effect, when you type

```
(foo a b c)
```

in a normal order evaluator, it's equivalent to typing

```
(foo (delay a) (delay b) (delay c))
```

in ordinary (applicative order) Scheme. If every argument is automatically delayed, then Louis's `pairs` procedure will work without adding explicit delays.

Louis's program had explicit calls to `force` as well as explicit calls to `delay`. If we're going to make this process automatic, when should we automatically force a promise? The answer is that some primitives need to know the real values of their arguments, e.g., the arithmetic primitives. And of course when Scheme is about to print the value of a top-level expression, we need the real value.

How do we modify the evaluator?

What changes must we make to the metacircular evaluator in order to get normal order?

We've just said that the point at which we want to automatically delay a computation is when an expression is used as an argument to a procedure. Where does the ordinary metacircular evaluator evaluate argument subexpressions? In this excerpt from `eval`:

```
(define (eval exp env)
  (cond ...
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        ...))
```

It's `list-of-values` that recursively calls `eval` for each argument subexpression. Instead we could make thunks:

```
(define (eval exp env)
  (cond ...
        ((application? exp)
         (apply (ACTUAL-VALUE (operator exp) env)
                (LIST-OF-DELAYED-VALUES (operands exp) env)))
        ...))
```

Two things have changed:

1. To find out what procedure to invoke, we use `actual-value` rather than `eval`. In the normal order evaluator, what `eval` returns may be a promise rather than a final value; `actual-value` forces the promise if necessary.

2. Instead of `list-of-values` we call `list-of-delayed-values`. The ordinary version uses `eval` to get the value of each argument expression; the new version will use `delay` to make a list of thunks. (This isn't quite true, and I'll fix it in a few paragraphs.)

When do we want to force the promises? We do it when calling a primitive procedure. That happens in `apply`:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ...))
```

We change it to force the arguments first:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure (MAP FORCE ARGUMENTS)))
        ...))
```

Those are the crucial changes. The book gives a few more details: Some special forms must force their arguments, and the read-eval-print loop must force the value it's about to print.

Reinventing `delay` and `force`

I said earlier that I was lying about using `delay` to make thunks. The metacircular evaluator can't use Scheme's built-in `delay` because that would make a thunk in the underlying Scheme environment, and we want a thunk in the metacircular environment. (This is one more example of the idea of level confusion.) Instead, the book uses procedures `delay-it` and `force-it` to implement metacircular thunks.

What's a thunk? It's an expression and an environment in which we should later evaluate it. So we make one by combining an expression with an environment:

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

The rest of the implementation is straightforward.

Notice that the `delay-it` procedure takes an environment as argument; this is because it's part of the implementation of the language, not a user-visible feature. If, instead of a lazy evaluator, we wanted to add a `delay` special form to the ordinary metacircular evaluator, we'd do it by adding this clause to `eval`:

```
((delay? exp) (delay-it (cadr exp) env))
```

Here `exp` represents an expression like `(delay foo)` and so its `cadr` is the thing we really want to delay.

The book's version of `eval` and `apply` in the lazy evaluator is a little different from what I've shown here. My version makes thunks in `eval` and passes them to `apply`. The book's version has `eval` pass the argument expressions to `apply`, without either evaluating or thunking them, and also passes the current environment as a third argument. Then `apply` either evaluates the arguments (for primitives) or thunks them (for non-primitives). Their way is more efficient, but I think this way makes the issues clearer because it's more nearly parallel to the division of labor between `eval` and `apply` in the vanilla metacircular evaluator.

Memoization

Why didn't we choose normal order evaluation for Scheme in the first place? One reason is that it easily leads to redundant computations. When we talked about it in chapter 1, I gave this example:

```
(define (square x) (* x x))

(square (square (+ 2 3)))
```

In a normal order evaluator, this adds 2 to 3 four times!

```
(square (square (+ 2 3)))  ==>
```

```
(* (square (+ 2 3)) (square (+ 2 3)))  ==>
(* (* (+ 2 3) (+ 2 3)) (* (+ 2 3) (+ 2 3)))
```

The solution is memoization. If we force the same thunk more than once, the thunk should remember its value from the first time and not have to repeat the computation. (The four instances of `(+ 2 3)` in the last line above are all the same thunk forced four times, not four separate thunks.)

The details are straightforward; you can read them in the text.

CS 61A Lecture Notes First Half of Week 7

Topic: Nondeterministic evaluator

Reading: Abelson & Sussman, Sections 4.3

To load the nondeterministic metacircular evaluator, say

```
(load "~cs61a/lib/vambeval.scm")
```

Solution spaces, streams, and backtracking

Many problems are of the form "Find all A such that B" or "find an A such that B." For example: Find an even integer that is not the sum of two primes; find a set of integers a, b, c, and n such that $a^n + b^n = c^n$ and $n > 2$. (These problems might not be about numbers: Find all the states in the United States whose first and last letters are the same.)

In each case, the set A (even integers, sets of four integers, or states) is called the *solution space*. The condition B is a predicate function of a potential solution that's true for actual solutions.

One approach to solving problems of this sort is to represent the solution space as a stream, and use `filter` to select the elements that satisfy the predicate:

```
(filter sum-of-two-primes? even-integers)

(filter Fermat? (pairs (pairs integers integers)
                       (pairs integers integers)))

(filter (lambda (x) (equal? (first x) (last x))) states)
```

The stream technique is particularly elegant for infinite problem spaces, because the program seems to be generating the entire solution space A before checking the predicate B. (Of course we know that really the steps of the computation are reordered so that the elements are tested as they are generated.)

In the next couple of lectures, we consider a different way to express the same sort of computation, a way that makes the sequence of events in time more visible. In effect we'll say:

- Pick a possible solution.

- See if it's really a solution.

- If so, return it; if not, try another.

Here's an example of the notation:

```
> (let ((a (amb 2 3 4))
        (b (amb 6 7 8)))
    (require (= (remainder b a) 0))
    (list a b))
(2 6)
> try-again
(2 8)
> try-again
(3 6)
> try-again
(4 8)
> try-again
There are no more solutions.
```

The main new thing here is the special form `amb`. This is not part of ordinary Scheme! We are adding it as a new feature in the metacircular evaluator. `Amb` takes any number of argument expressions and returns the value of one of them. You can think about this using either of two metaphors:

• The computer clones itself into as many copies as there are arguments; each clone gets a different value.

• The computer magically knows which argument will give rise to a solution to your problem and chooses that one.

What really happens is that the evaluator chooses the first argument and returns its value, but if the computation later *fails* then it tries again with the second argument, and so on until there are no more to try. This introduces another new idea: the possibility of the failure of a computation. That's not the same thing as an error! Errors (such as taking the `car` of an empty list) are handled the same in this evaluator as in ordinary Scheme; they result in an error message and the computation stops. A failure is different; it's what happens when you call `amb` with no arguments, or when all the arguments you gave have been tried and there are no more left.

In the example above I used `require` to cause a failure of the computation if the condition is not met. `Require` is a simple procedure in the metacircular Scheme-with-`amb`:

```
(define (require condition)
  (if (not condition) (amb)))
```

So here's the sequence of events in the computation above:

```
a=2
    b=6; 6 is a multiple of 2, so return (2 6)

[try-again]
    b=7; 7 isn't a multiple of 2, so fail.
    b=8; 8 is a multiple of 2, so return (2 8)

[try-again]
    No more values for b, so fail.
a=3
    b=6; 6 is a multiple of 3, so return (3 6)

[try-again]
    b=7; 7 isn't a multiple of 3, so fail.
    b=8; 8 isn't a multiple of 3, so fail.
    No more values for b, so fail.
a=4
    b=6; 6 isn't a multiple of 4, so fail.
    b=7; 7 isn't a multiple of 4, so fail.
    b=8; 8 is a multiple of 4, so return (4 8)

[try-again]
    No more values for b, so fail.
No more values for a, so fail.
(No more pending AMBs, so report failure to user.)
```

Recursive **Amb**

Since amb accepts any argument expressions, not just literal values as in the example above, it can be used recursively:

```
(define (an-integer-between from to)
  (if (> from to)
      (amb)
      (amb from (an-integer-between (+ from 1) to))))
```

or if you prefer:

```
(define (an-integer-between from to)
  (require (>= to from))
  (amb from (an-integer-between (+ from 1) to)))
```

Further, since amb is a special form and only evaluates one argument at a time, it has the same delaying effect as cons-stream and can be used to make infinite solution spaces:

```
(define (integers-from from)
  (amb from (integers-from (+ from 1))))
```

This integers-from computation never fails—there is always another integer—and so it won't work to say

```
(let ((a (integers-from 1))
      (b (integers-from 1)))
  ...)
```

because a will never have any value other than 1, because the second amb never fails. This is analogous to the problem of trying to append infinite streams; in that case we could solve the problem with interleave but it's harder here.

Footnote on order of evaluation

In describing the sequence of events in these examples, I'm assuming that Scheme will evaluate the arguments of the unnamed procedure created by a let from left to right. If I wanted to be sure of that, I should use let* instead of let. But it matters only in my description of the sequence of events; considered abstractly, the program will behave correctly regardless of the order of evaluation, because all possible solutions will eventually be tried—although maybe not in the order shown here.

Success or failure

In the implementation of amb, the most difficult change to the evaluator is that any computation may either succeed or fail. The most obvious way to try to represent this situation is to have eval return some special value, let's say the symbol =failed=, if a computation fails. (This is analogous to the use of =no-value= in the Logo interpreter project.) The trouble is that if an amb fails, we don't want to continue the computation; we want to "back up" to an earlier stage in the computation. Suppose we are trying to evaluate an expression such as

```
(a (b (c (d 4))))
```

and suppose that procedures b and c use amb. Procedure d is actually invoked first; then c is invoked with the value d returned as argument. The amb inside procedure c returns its first argument, and c uses that to compute a return value that becomes the argument to b. Now suppose that the amb inside b fails. We don't want to invoke a with the value =failed= as its argument! In fact we don't want to invoke a at all; we want to re-evaluate the body of c but using the second argument to its amb.

A&S take a different approach. If an amb fails, they want to be able to jump right back to the previous amb, without having to propagate the failure explicitly through several intervening calls to eval. To make this work, intuitively, we have to give eval two different places to return to when it's finished, one for a success and the other for a failure.

Continuations

Ordinarily a procedure doesn't think explicitly about where to return; it returns to its caller, but Scheme takes care of that automatically. For example, when we compute

```
(* 3 (square 5))
```

the procedure square computes the value 25 and Scheme automatically returns that value to the eval invocation that's waiting to use it as an argument to the multiplication. But we could tell square explicitly, "when you've figured out the answer, pass it on to be multiplied by 3" this way:

```
(define (square x continuation)
  (continuation (* x x)))

> (square 5 (lambda (y) (* y 3)))
75
```

A *continuation* is a procedure that takes your result as argument and says what's left to be done in the computation.

Continuations for success and failure

In the case of the nondeterministic evaluator, we give eval *two* continuations, one for success and one for failure. Note that these continuations are part of the implementation of the evaluator; the user of amb doesn't deal explicitly with continuations.

Here's a handwavy example. In the case of

```
(a (b (c (d 4))))
```

procedure b's success continuation is something like

```
(lambda (value) (a value))
```

but its failure continuation is

```
(lambda () (a (b (redo-amb-in-c))))
```

This example is handwavy because these "continuations" are from the point of view of the user of the metacircular Scheme, who doesn't know anything about continuations, really. The true continuations are written in underlying Scheme, as part of the evaluator itself.

If a computation fails, the most recent amb wants to try another value. So a continuation failure will redo the amb with one fewer argument. There's no information that the failing computation needs to send back to that amb except for the fact of failure itself, so the failure continuation procedure needs no arguments.

On the other hand, if the computation succeeds, we have to carry out the success continuation, and that continuation needs to know the value that we computed. It also needs to know what to do if the continuation itself fails; most of the time, this will be the same as the failure continuation we were given, but it might not be. So a success continuation must be a procedure that takes two arguments: a value and a failure continuation.

The book bases the nondeterministic evaluator on the analyzing one, but I'll use a simplified version based on plain old eval (it's in cs61a/lib/vambeval.scm).

Most kinds of evaluation always succeed, so they invoke their success continuation and pass on the failure one. I'll start with a too-simplified version of eval-if in this form:

```
(define (eval-if exp env succeed fail)          ; WRONG!
  (if (eval (if-predicate exp) env succeed fail)
      (eval (if-consequent exp) env succeed fail)
      (eval (if-alternative exp) env succeed fail)))
```

The trouble is, what if the evaluation of the predicate fails? We don't then want to evaluate the consequent or the alternative. So instead, we just evaluate the predicate, giving it a success continuation that will evaluate the consequent or the alternative, supposing that evaluating the predicate succeeds.

In general, wherever the ordinary metacircular evaluator would say

```
(define (eval-foo exp env)
   (eval step-1 env)
   (eval step-2 env))
```

using `eval` twice for part of its work, this version has to `eval` the first part with a continuation that `evals` the second part:

```
(define (eval-foo exp env succeed fail)
  (eval step-1
        env
        (lambda (value-1 fail-1)
          (eval step-2 env succeed fail-1))
        fail))
```

(In either case, `step-2` presumably uses the result of evaluating `step-1` somehow.)

Here's how that works out for `if`:

```
(define (eval-if exp env succeed fail)
  (eval (if-predicate exp)            ; test the predicate
        env
        (lambda (pred-value fail2)    ; with this success continuation
          (if (true? pred-value)
              (eval (if-consequent exp) env succeed fail2)
              (eval (if-alternative exp) env succeed fail2)))
        fail))                        ; and the same failure continuation
```

What's `fail2`? It's the failure continuation that the evaluation of the predicate will supply. Most of the time, that'll be the same as our own failure continuation, just as `eval-if` uses `fail` as the failure continuation to pass on to the evaluation of the predicate. But if the predicate involves an `amb` expression, it will generate a new failure continuation. Think about an example like this one:

```
> (if (amb #t #f)
      (amb 1)
      (amb 2))
1

> try-again
2
```

(A more realistic example would have the predicate expression be some more complicated procedure call that had an `amb` in its body.) The first thing that happens is that the first `amb` returns `#t`, and so `if` evaluates its second argument, and that second `amb` returns 1. When the user says to try again, there are no more values for that `amb` to return, so it fails. What we must do is re-evaluate the first `amb`, but this time returning its second argument, `#f`. By now you've forgotten that we're trying to work out what `fail2` is for in `eval-if`, but this example shows why the failure continuation when we evaluate `if-consequent` (namely the `(amb 1)` expression) has to be different from the failure continuation for the entire `if` expression. If the entire `if` fails (which will happen if we say `try-again` again) then its failure continuation will tell us that there are no more values. That continuation is bound to the name `fail` in `eval-if`. What ends up bound to the name `fail2` is the continuation that re-evaluates the predicate `amb`.

How does `fail2` get that binding? When `eval-if` evaluates the predicate, which turns out to be an `amb` expression, `eval-amb` will evaluate whatever argument it's up to, but with a new failure continuation:

```
(define (eval-amb exp env succeed fail)
  (if (null? (cdr exp))            ; (car exp) is the word AMB
      (fail)                       ; no more args, call failure cont.
      (eval (cadr exp)             ; Otherwise evaluate the first arg
            env
            succeed                ; with my same success continuation
            (lambda ()             ; but with a new failure continuation:
              (eval-amb (cons 'amb (cddr exp))    ; try the next argument
                        env
                        succeed
                        fail)))))
```

Notice that `eval-if`, like most other cases, provides a new success continuation but passes on the same failure continuation that it was given as an argument. But `eval-amb` does the opposite: It passes on the same success continuation it was given, but provides a new failure continuation.

Of course there are a gazillion more details, but the book explains them, once you understand what a continuation is. The most important of these complications is that anything involving mutation is problematic. If we say

```
(define x 5)
(set! x (+ x (amb 2 3)))
```

it's clear that the first time around `x` should end up with the value 7 $(5 + 2)$. But if we try again, we'd like `x` to get the value 8 $(5 + 3)$, not 10 $(7 + 3)$. So `set!` must set up a failure continuation that undoes the change in the binding of `x`, restoring its original value of 5, before letting the `amb` provide its second argument.

CS 61A Lecture Notes Second Half of Week 7

Topic: Nondeterministic evaluator

Reading:

Note: For the second half of week 7, we will finish the nondeterministic evaluator and, time permitting, perhaps move on to new material.

Any additional reading will be posted on the webpage.

CS 61A Lecture Notes First Half of Week 8

Topic: Logic programming

Reading: Abelson & Sussman, Section 4.4.1–3

This week's big idea is *logic programming* or *declarative programming.*

It's the biggest step we've taken away from expressing a computation in hardware terms. When we discovered streams, we saw how to express an algorithm in a way that's independent of the *order* of evaluation. Now we are going to describe a computation in a way that has no (visible) algorithm at all!

We are using a logic programming language that A&S implemented in Scheme. Because of that, the notation is Scheme-like, i.e., full of lists. Standard logic languages like Prolog have somewhat different notations, but the idea is the same.

All we do is assert facts:

```
> (load "~cs61a/lib/query.scm")
> (query)

;;; Query input:
(assert! (Brian likes potstickers))
```

and ask questions about the facts:

```
;;; Query input:
(?who likes potstickers)

;;; Query results:
(BRIAN LIKES POTSTICKERS)
```

Although the assertions and the queries take the form of lists, and so they look a little like Scheme programs, they're not! There is no application of function to argument here; an assertion is just data.

This is true even though, for various reasons, it's traditional to put the verb (the *relation*) first:

```
(assert! (likes Brian potstickers))
```

We'll use that convention hereafter, but that makes it even easier to fall into the trap of thinking there is a *function* called `likes`.

• Rules. As long as we just tell the system isolated facts, we can't get extraordinarily interesting replies. But we can also tell it *rules* that allow it to infer one fact from another. For example, if we have a lot of facts like

```
(mother Eve Cain)
```

then we can establish a rule about grandmotherhood:

```
(assert! (rule (grandmother ?elder ?younger)
               (and (mother ?elder ?mom)
                    (mother ?mom ?younger) )))
```

The rule says that the first part (the conclusion) is true *if* we can find values for the variables such that the second part (the condition) is true.

Again, resist the temptation to try to do composition of functions!

```
(assert! (rule (grandmother ?elder ?younger)         ;; WRONG!!!!
               (mother ?elder (mother ?younger)) ))
```

Mother isn't a function, and you can't ask for the mother of someone as this incorrect example tries to do. Instead, as in the correct version above, you have to establish a variable (?mom) that has a value that satisfies the two motherhood relationships we need.

In this language the words assert!, rule, and, or, and not have special meanings. Everything else is just a word that can be part of assertions or rules.

Once we have the idea of rules, we can do real magic:

```
;;;;;                           In file cs61a/lectures/4.4/logic-utility.scm
(assert! (rule (append (?u . ?v) ?y (?u . ?z))
               (append ?v ?y ?z)))

(assert! (rule (append () ?y ?y)))
```

(The actual online file uses a Scheme procedure aa to add the assertion. It's just like saying assert! to the query system, but you say it to Scheme instead. This lets you load the file. Don't get confused about this small detail—just ignore it.)

```
;;; Query input:
(append (a b) (c d e) ?what)

;;; Query results:
(APPEND (A B) (C D E) (A B C D E))
```

So far this is just like what we could do in Scheme.

```
;;; Query input:
(append ?what (d e) (a b c d e))

;;; Query results:
(APPEND (A B C) (D E) (A B C D E))

;;; Query input:
(append (a) ?what (a b c d e))

;;; Query results:
(APPEND (A) (B C D E) (A B C D E))
```

The new thing in logic programming is that we can run a "function" backwards! We can tell it the answer and get back the question. But the real magic is...

```
;;; Query input:
(append ?this ?that (a b c d e))

;;; Query results:
(APPEND () (A B C D E) (A B C D E))
(APPEND (A) (B C D E) (A B C D E))
(APPEND (A B) (C D E) (A B C D E))
(APPEND (A B C) (D E) (A B C D E))
(APPEND (A B C D) (E) (A B C D E))
(APPEND (A B C D E) () (A B C D E))
```

We can use logic programming to compute multiple answers to the same question! Somehow it found all the possible combinations of values that would make our query true.

How does the append program work? Compare it to the Scheme append:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b)) ))
```

Like the Scheme program, the logic program has two cases: There is a base case in which the first argument is empty. In that case the combined list is the same as the second appended list. And there is a recursive case in which we divide the first appended list into its car and its cdr. We reduce the given problem into a problem about appending (cdr a) to b. The logic program is different in form, but it says the same thing. (Just as, in the grandmother example, we had to give the mother a name instead of using a function call, here we have to give (car a) a name—we call it ?u.)

Unfortunately, this "working backwards" magic doesn't always work.

```
;;;;;                        In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (reverse (?a . ?x) ?y)
               (and (reverse ?x ?z)
                    (append ?z (?a) ?y) )))

(assert! (reverse () ()))
```

This works for (reverse (a b c) ?what) but not the other way around; it gets into an infinite loop. We can also write a version that works *only* backwards:

```
;;;;;                        In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (backward (?a . ?x) ?y)
               (and (append ?z (?a) ?y)
                    (backward ?x ?z) )))

(assert! (backward () ()))
```

But it's much harder to write one that works both ways. Even as we speak, logic programming fans are trying to push the limits of the idea, but right now, you still have to understand something about the below-the-line algorithm to be confident that your logic program won't loop.

• Below-the-line implementation.

Think about eval in the MC evaluator. It takes two arguments, an expression and an environment, and it returns the value of the expression.

In logic programming, there's no such thing as "the value of the expression." What we're given is a query, and there may or may not be some number of variable bindings that make the query true. The query evaluator qeval is analogous to eval in that it takes two arguments, something to evaluate and a context in which to work. But the thing to evaluate is a query, not an expression; the context isn't just one environment but a whole collection of environments—one for each set of variable values that satisfy some previous query. And the result returned by qeval isn't a value. It's a new collection of environments! It's as if eval returned an environment instead of a value.

The “collection” of environments we’re talking about here is represented as a stream. That’s because there might be infinitely many of them! We use the stream idea to reorder the computation; what really happens is that we take one potential set of satisfying values and work it all the way through; then we try another potential set of values. But the program looks as if we compute all the satisfying values at once for each stage of a query.

Just as every top-level Scheme expression is evaluated in the global environment, every top-level query is evaluated in an *empty* stream of environments. (No variables have been assigned values yet.)

If we have a query like `(and p q)`, what happens is that we recursively use `qeval` to evaluate `p` in the empty stream. The result is a stream of variable bindings that satisfy `p`. Then we use `qeval` to evaluate `q` in that result stream! The final result is a stream of bindings that satisfy `p` and `q` simultaneously.

If the query is `(or p q)` then we use `qeval` to evaluate each of the pieces independently, starting in both cases with the empty frame. Then we *merge* the two result streams to get a stream of bindings that satisfy either `p` or `q`.

If the query is `(not q)`, we can’t make sense of that unless we already have a stream of environments to work with. That’s why we can only use `not` in a context such as `(and p (not q))`. We take the stream of environments that we already have, and we `filter` that stream, using as the test predicate the function

```
(lambda (env) (empty-stream? (qeval q env)))
```

That is, we keep only those environments for which we *can’t* satisfy `q`.

That explains how `qeval` reduces compound queries to simple ones. How do we evaluate a simple query? The first step is to *pattern match* the query against every assertion in the data base. Pattern matching is just like the recursive `equal?` function, except that a variable in the pattern (the query) matches anything in the assertion. (But if the same variable appears more than once, it must match the same thing each time. That’s why we need to keep an environment of matches so far.)

The next step is to match the query against the *conclusions* of rules. This is tricky because now there can be variables in both things being matched. Instead of the simple pattern matching we have to use a more complicated version called *unification.* (See the details in the text.) If we find a match, then we take the condition part of the rule (the body) and use that as a new query, to be satisfied within the environment(s) that `qeval` gave us when we matched the conclusion. In other words, first we look at the conclusion to see whether this rule can possibly be relevant to our query; if so, we see if the conditions of the rule are true.

Here's an example, partly traced:

```
;;; Query input:
(append ?a ?b (aa bb))

(unify-match (append ?a ?b (aa bb))         ; MATCH ORIGINAL QUERY
             (append () ?1y ?1y)            ; AGAINST BASE CASE RULE
             ())                            ; WITH NO CONSTRAINTS

RETURNS: ((?1y . (aa bb)) (?b . ?1y) (?a . ()))
PRINTS:  (append () (aa bb) (aa bb))
```

Since the base-case rule has no body, once we've matched it, we can print a successful result. (Before printing, we have to look up variables in the environment so what we print is variable-free.) Now we unify the original query against the conclusion of the other rule:

```
(unify-match (append ?a ?b (aa bb))                ; MATCH ORIGINAL QUERY
             (append (?2u . ?2v) ?2y (?2u . ?2z))  ; AGAINST RECURSIVE RULE
             ())                                   ; WITH NO CONSTRAINTS

RETURNS: ((?2z . (bb)) (?2u . aa) (?b . ?2y) (?a . (?2u . ?2v)))
         [call it F1]
```

This was successful, but we're not ready to print anything yet, because we now have to take the body of that rule as a new query. Note the indenting to indicate that this call to `unify-match` is within the pending rule.

```
    (unify-match (append ?2v ?2y ?2z)   ; MATCH BODY OF RECURSIVE RULE
                 (append () ?3y ?3y)    ; AGAINST BASE CASE RULE
                 F1)                    ; WITH CONSTRAINTS FROM F1

    RETURNS: ((?3y . (bb)) (?2y . ?3y) (?2v . ()) [plus F1])
    PRINTS:  (append (aa) (bb) (aa bb))

    (unify-match (append ?2v ?2y ?2z)                  ; MATCH SAME BODY
                 (append (?4u . ?4v) ?4y (?4u . ?4z))  ; AGAINST RECURSIVE RULE
                 F1)                                   ; WITH F1 CONSTRAINTS

    RETURNS: ((?4z . ()) (?4u . bb) (?2y . ?4y) (?2v . (?4u . ?4v))
              [plus F1])  [call it F2]

        (unify-match (append ?4v ?4y ?4z)   ; MATCH BODY FROM NEWFOUND MATCH
                     (append () ?5y ?5y)    ; AGAINST BASE CASE RULE
                     F2)                    ; WITH NEWFOUND CONSTRAINTS

        RETURNS: ((?5y . ()) (?4y . ?5y) (?4v . ()) [plus F2])
        PRINTS:  (append (aa bb) () (aa bb))

        (unify-match (append ?4v ?4y ?4z)                  ; MATCH SAME BODY
                     (append (?6u . ?6v) ?6y (?6u . ?6z))  ; AGAINST RECUR RULE
                     F2)                                   ; SAME CONSTRAINTS

        RETURNS: ()                                        ; BUT THIS FAILS

done
```

CS 61A Lecture Notes Second Half of Week 8

Topic: Review

Reading: No new reading; study for the final.

• Go over first-day handout about abstraction; show how each topic involves an abstraction barrier and say what's above and what's below the line.

• Go over the big ideas within each programming paradigm:

Functional Programming:
composition of functions
first-class functions (function as object)
higher-order functions
recursion
delayed (lazy) evaluation
(vocabulary: parameter, argument, scope, iterative process)

Object-Oriented Programming:
actors
message passing
local state
inheritance
identity vs. equal value
(vocabulary: dispatch procedure, delegation, mutation)

Logic Programming:
focus on ends, not means
multiple solutions
running a program backwards
(vocabulary: pattern matching, unification)

• Review where 61A fits into the curriculum. (See the CS abstraction hierarchy in week 1.)

Please, please, don't forget the ideas of 61A just because you're not programming in Scheme!

Object-Oriented Programming — Above the line view

This document should be read before Section 3.1 of the text. A second document, “Object-Oriented Programming — Below the line view,” should be read after Section 3.1 and perhaps after Section 3.2; the idea is that you first learn how to use the object-oriented programming facility, then you learn how it’s implemented.

Object-oriented programming is a metaphor. It expresses the idea of several independent agents inside the computer, instead of a single process manipulating various data. For example, the next programming project is an adventure game, in which several people, places, and things interact. We want to be able to say things like “Ask Fred to pick up the potstickers.” (Fred is a *person* object, and the potstickers are a *thing* object.)

Programmers who use the object metaphor have a special vocabulary to describe the components of an object-oriented programming (OOP) system. In the example just above, “Fred” is called an *instance* and the general category “person” is called a *class.* Programming languages that support OOP let the programmer talk directly in this vocabulary; for example, every OOP language has a “define class” command in some form. For this course, we have provided an extension to Scheme that supports OOP in the style of other OOP languages. Later we shall see how these new features are implemented using Scheme capabilities that you already understand. OOP is not magic; it’s a way of thinking and speaking about the structure of a program.

When we talk about a “metaphor,” in technical terms we mean that we are providing an abstraction. The above-the-line view is the one about independent agents. Below the line there are three crucial technical ideas: message-passing (section 2.3), local state (section 3.1), and inheritance (explained below). This document will explain how these ideas look to the OOP programmer; later we shall see how they are implemented.

A simpler version of this system and of these notes came from MIT; this version was developed at Berkeley by Matt Wright.

In order to use the OOP system, you must load the file `~cs61a/lib/obj.scm` into Scheme.

Message Passing

The way to get things to happen in an object oriented system is to send messages to objects asking them to do something. You already know about message passing; we used this technique in Section 2.3 to implement generic operators using “smart” data. For example, in Section 3.1 much of the discussion will be about *bank account* objects. Each account has a *balance* (how much money is in it); you can send messages to a particular account to *deposit* or *withdraw* money. The book’s version shows how these objects can be created using ordinary Scheme notation, but now we’ll use OOP vocabulary to do the same thing. Let’s say we have two objects `Matt-Account` and `Brian-Account` of the bank account class. (You can’t actually type this into Scheme yet; the example assumes that we’ve already created these objects.)

```
> (ask Matt-Account ’balance)
1000
```

```
> (ask Brian-Account 'balance)
10000
> (ask Matt-Account 'deposit 100)
1100
> (ask Brian-Account 'withdraw 200)
9800
> (ask Matt-Account 'balance)
1100
> (ask Brian-Account 'withdraw 200)
9600
```

We use the procedure `ask` to send a message to an object. In the above example we assumed that bank account objects knew about three messages: `balance`, `deposit`, and `withdraw`. Notice that some messages require additional information; when we asked for the `balance`, that was enough, but when we ask an account to `withdraw` or `deposit` we needed to specify the amount also.

The metaphor is that an object "knows how" to do certain things. These things are called *methods*. Whenever you send a message to an object, the object carries out the method it associates with that message.

Local State

Notice that in the above example, we repeatedly said

```
(ask Brian-Account 'withdraw 200)
```

and got a different answer each time. It seemed perfectly natural, because that's how bank accounts work in real life. However, until now we've been using the functional programming paradigm, in which, by definition, calling the same function twice with the same arguments must give the same result.

In the OOP paradigm, the objects have *state*. That is, they have some knowledge about what has happened to them in the past. In this example, a bank account has a balance, which changes when you deposit or withdraw some money. Furthermore, each account has its own balance. In OOP jargon we say that `balance` is a *local state variable*.

You already know what a local variable is: a procedure's formal parameter is one. When you say

```
(define (square x) (* x x))
```

the variable `x` is local to the square procedure. If you had another procedure `(cube x)`, its variable `x` would be entirely separate from that of `square`. Likewise, the `balance` of `Matt-Account` is kept separate from that of `Brian-Account`.

On the other hand, every time you invoke `square`, you supply a new value for `x`; there is no memory of the value `x` had last time around. A *state* variable is one whose value survives between invocations. After you deposit some money to `Matt-Account`, the `balance` variable's new value is remembered the next time you access the account.

To create objects in this system you *instantiate* a class. For example, `Matt-Account` and

Brian-Account are instances of the account class:

```
> (define Matt-Account (instantiate account 1000))
Matt-Account
> (define Brian-Account (instantiate account 10000))
Brian-Account
```

The instantiate function takes a class as its first argument and returns a new object of that class. Instantiate may require additional arguments depending on the particular class: in this example you specify an account's initial balance when you create it.

Most of the code in an object-oriented program consists of definitions of various classes. Here is the account class:

```
(define-class (account balance)
  (method (deposit amount)
    (set! balance (+ amount balance))
    balance)
  (method (withdraw amount)
    (if (< balance amount)
        "Insufficient funds"
        (begin
         (set! balance (- balance amount))
         balance))) )
```

There's a lot to say about this code. First of all, there's a new special form, define-class. The syntax of define-class is analogous to that of define. Where you would expect to see the name of the procedure you're defining comes the name of the class you're defining. In place of the parameters to a procedure come the *initialization variables* of the class: these are local state variables whose initial values must be given as the extra arguments to instantiate. The body of a class consists of any number of *clauses*; in this example there is only one kind of clause, the method clause, but we'll learn about others later. The order in which clauses appear within a define-class doesn't matter.

The syntax for defining methods was also chosen to resemble that for defining procedures. The "name" of the method is actually the *message* used to access the method. The parameters to the method correspond to extra arguments to the ask procedure. For example, when we said

```
(ask Matt-Account 'deposit 100)
```

we associated the argument 100 with the parameter amount.

You're probably wondering where we defined the balance method. For each local state variable in a class, a corresponding method of the same name is defined automatically. These methods have no arguments, and they just return the current value of the variable with that name.

This example also introduced two new special forms that are not unique to the object system. The first is set!, whose job it is to change the value of a state variable. Its first argument is unevaluated; it is the name of the variable whose value you wish to change. The second argument *is* evaluated; the value of this expression becomes the new value of the variable. The return value of set! is undefined.

This looks a lot like the kind of `define` without parentheses around the first argument, but the meaning is different. `Define` creates a new variable, while `set!` changes the value of an existing variable.

The name `set!` has an exclamation point in its name because of a Scheme convention for procedures that modify something. (This is just a convention, like the convention about question marks in the names of predicate functions, not a firm rule.) The reason we haven't come across this convention before is that functional programming rules out the whole idea of modifying things; there is no memory of past history in a functional program.

The other Scheme primitive special form in this example is `begin`, which evaluates all of its argument expressions in order and returns the value of the last one. Until now, in every procedure we've evaluated only one expression, to provide the return value of that procedure. It's still the case that a procedure can only return one value. Now, though, we sometimes want to evaluate an expression for what it *does* instead of what it *returns*, e.g. changing the value of a variable. The call to `begin` indicates that the `(set! amount (- amount balance))` and the `balance` together form a single argument to `if`. You'll learn more about `set!` and `begin` in Chapter 3.

Inheritance

Imagine using OOP in a complicated program with many different kinds of objects. Very often, there will be a few classes that are almost the same. For example, think about a window system. There might be different kinds of windows (text windows, graphics windows, and so on) but all of them will have certain methods in common, e.g., the method to move a window to a different position on the screen. We don't want to have to reprogram the same method in several classes. Instead, we create a more general class (such as "window") that knows about these general methods; the specific classes (like "text window") *inherit* from the general class. In effect, the definition of the general class is included in that of the more specific class.

Let's say we want to create a checking account class. Checking accounts are just like regular bank accounts, except that you can write checks as well as withdrawing money in person. But you're charged ten cents every time you write a check.

```
> (define Hal-Account (instantiate checking-account 1000))
Hal-Account
> (ask Hal-Account 'balance)
1000
> (ask Hal-Account 'deposit 100)
1100
> (ask Hal-Account 'withdraw 50)
1050
> (ask Hal-Account 'write-check 30)
1019.9
```

One way to do this would be to duplicate all of the code for regular accounts in the definition of the `checking-account`. This isn't so great, though; if we want to add a new feature to the `account` class we would need to remember to add it to the `checking-account` class as well.

It is very common in object-oriented programming that one class will be a *specialization* of another: the new class will have all the methods of the old, plus some extras, just as in this bank account example. To describe this situation we use the metaphor of a *family* of object classes. The original class is the *parent* and the specialized version is the *child* class. We say that the child inherits the methods of the parent. (The names *subclass* for child and *superclass* for parent are also sometimes used.)

Here's how we create a subclass of the `account` class:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

This example introduces the `parent` clause in `define-class`. In this case, the parent is the `account` class. Whenever we send a message to a `checking-account` object, where does the corresponding method come from? If a method of that name is defined in the `checking-account` class, it is used; otherwise, the OOP system looks for a method in the parent `account` class. (If that class also had a parent, we might end up inheriting a method from that twice-removed class, and so on.)

Notice also that the `write-check` method refers to a variable called `self`. Each object has a local state variable `self` whose value is the object itself. (Notice that you might write a method within the definition of a class `C` thinking that `self` will always be an instance of `C`, but in fact `self` might turn out to be an instance of another class that has `C` as its parent.)

Methods defined in a certain class only have access to the local state variables defined in the same class. For example, a method defined in the `checking-account` class can't refer to the `balance` variable defined in the `account` class; likewise, a method in the `account` class can't refer to the `init-balance` variable. This rule corresponds to the usual Scheme rule about scope of variables: each variable is only available within the block in which it's defined. (Not every OOP implementation works like this, by the way.)

If a method in the `checking-account` class needs to refer to the `balance` variable defined in its parent class, the method could say

```
(ask self 'balance)
```

This invocation of `ask` sends a message to the `checking-account` object, but because there is no `balance` method defined within the `checking-account` class itself, the method that's inherited from the `account` class is used.

We used the name `init-balance` for the new class's initialization variable, rather than just `balance`, because we want that name to mean the variable belonging to the parent class. Since the OOP system automatically creates a method named after every local variable in the class, if we called this variable `balance` then we couldn't use a `balance` message to get at the parent's `balance` state variable. (It is the parent, after all, in which the account's balance is changed for each transaction.)

We have now described the three most important parts of the OOP system: message passing, local state, and inheritance. In the rest of this document we introduce some "bells and whistles"—additional features that make the notation more flexible, but don't really involve major new ideas.

Three Kinds of Local State Variables

So far the only local state variables we've seen have been *instantiation* variables, whose values are given as arguments when an object is created. Sometimes we'd like each instance to have a local state variable, but the initial value is the same for every object in the class, so we don't want to have to mention it at each instantiation. To achieve this purpose, we'll use a new kind of `define-class` clause, called `instance-vars`:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (instance-vars (check-fee 0.10))
  (method (write-check amount)
    (ask self 'withdraw (+ amount check-fee)))
  (method (set-fee! fee)
    (set! check-fee fee)) )
```

We've set things up so that every new checking account will have a ten-cent fee for each check. It's possible to change the fee for any given account, but we don't have to say anything if we want to stick with the ten cent value.

Instantiation variables are *also* instance variables; that is, every instance has its own private value for them. The only difference is in the notation—for instantiation variables you give a value when you call `instantiate`, but for other instance variables you give the value in the class definition.

The third kind of local state variable is a *class* variable. Unlike the case of instance variables, there is only one value for a class variable for the entire class. Every instance of the class shares this value. For example, let's say we want to have a class of workers that are all working on the same project. That is to say, whenever any of them works, the total amount of work done is increased. On the other hand, each worker gets hungry separately as he or she works. Therefore, there is a common `work-done` variable for the class, and a separate `hunger` variable for each instance.

```
(define-class (worker)
  (instance-vars (hunger 0))
  (class-vars (work-done 0))
  (method (work)
    (set! hunger (1+ hunger))
    (set! work-done (1+ work-done))
    'whistle-while-you-work ))

> (define brian (instantiate worker))
BRIAN
> (define matt (instantiate worker))
MATT
> (ask matt 'work)
WHISTLE-WHILE-YOU-WORK
> (ask matt 'work)
WHISTLE-WHILE-YOU-WORK
> (ask matt 'hunger)
2
```

```
> (ask matt 'work-done)
2
> (ask brian 'work)
WHISTLE-WHILE-YOU-WORK
> (ask brian 'hunger)
1
> (ask brian 'work-done)
3
> (ask worker 'work-done)
3
```

As you can see, asking any worker object to work increments the `work-done` variable. In contrast, each worker has its own `hunger` instance variable, so that when Brian works, Matt doesn't get hungry.

You can ask any instance the value of a class variable, or you can ask the class itself. This is an exception to the usual rule that messages must be sent to instances, not to classes.

Initialization

Sometimes we want every new instance of some class to carry out some initial activity as soon as it's created. For example, let's say we want to maintain a list of all the worker objects. We'll create a class variable called `all-workers` to hold the list, but we also have to make sure that each newly created instance adds itself to the list. We do this with an `initialize` clause:

```
(define-class (worker)
  (instance-vars (hunger 0))
  (class-vars (all-workers '())
              (work-done 0))
  (initialize (set! all-workers (cons self all-workers)))
  (method (work)
    (set! hunger (1+ hunger))
    (set! work-done (1+ work-done))
    'whistle-while-you-work ))
```

The body of the `initialize` clause is evaluated when the object is instantiated. (By the way, don't get confused about those two long words that both start with "I." *Instantiation* is the process of creating an instance (that is, a particular object) of a class. *Initialization* is some optional, class-specific activity that the newly instantiated object might perform.)

If a class and its parent class both have `initialize` clauses, the parent's clause is evaluated first. This might be important if the child's initialization refers to local state that is maintained by methods in the parent class.

Classes That Recognize Any Message

Suppose we want to create a class of objects that return the value of the previous message they received whenever you send them a new message. Obviously, each such object needs an instance variable in which it will remember the previous message. The hard part is that we want objects of this class to accept *any* message, not just a few specific messages. Here's how:

```
(define-class (echo-previous)
  (instance-vars (previous-message 'first-time))
  (default-method
    (let ((result previous-message))
      (set! previous-message message)
      result)))
```

We used a `default-method` clause; the body of a `default-method` clause gets evaluated if an object receives a message for which it has no method. (In this case, the `echo-previous` object doesn't have any regular methods, so the `default-method` code is executed for any message.)

Inside the body of the `default-method` clause, the variable `message` is bound to the message that was received and the variable `args` is bound to a list of any additional arguments to `ask`.

Using a Parent's Method Explicitly

In the example about checking accounts earlier, we said

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

Don't forget how this works: Because the `checking-account` class has a parent, whatever messages it doesn't understand are processed in the same way that the parent (`account`) class would handle them. In particular, `account` objects have `deposit` and `withdraw` methods.

Although a `checking-account` object asks itself to `withdraw` some money, we really intend that this message be handled by a method defined within the parent `account` class. There is no problem here because the `checking-account` class itself does not have a `withdraw` method.

Imagine that we want to define a class with a method of the same name as a method in its parent class. Also, we want the child's method to invoke the parent's method of the same name. For example, we'll define a `TA` class that is a specialization of the `worker` class. The only difference is that when you ask a `TA` to work, he or she returns the sentence "Let me help you with that box and pointer diagram" after invoking the `work` method defined in the `worker` class.

We can't just say `(ask self 'work)`, because that will refer to the method defined in the child class. That is, suppose we say:

```
(define-class (TA)
  (parent (worker))
  (method (work)
    (ask self 'work)     ;; WRONG!
    '(Let me help you with that box and pointer diagram))
  (method (grade-exam) 'A+) )
```

When we ask a TA to `work`, we are hoping to get the result of asking a worker to `work` (increasing hunger, increasing work done) but return a different sentence. But what actually happens is an infinite recursion. Since `self` refers to the TA, and the TA does have its own `work` method, that's what gets used. (In the earlier example with checking accounts, `ask self` works because the checking account does *not* have its own `withdraw` method.)

Instead we need a way to access the method defined in the parent (`worker`) class. We can accomplish this with `usual`:

```
(define-class (TA)
  (parent (worker))
  (method (work)
    (usual 'work)
    '(Let me help you with that box and pointer diagram))
  (method (grade-exam) 'A+) )
```

`Usual` takes one or more arguments. The first argument is a message, and the others are whatever extra arguments are needed. Calling `usual` is just like saying `(ask self ...)` with the same arguments, except that only methods defined within an ancestor class (parent, grandparent, etc.) are eligible to be used. It is an error to invoke `usual` from a class that doesn't have a parent class.

You may be thinking that `usual` is a funny name for this function. Here's the idea behind the name: We are thinking of subclasses as specializations. That is, the parent class represents some broad category of things, and the child is a specialized version. (Think of the relationship of checking accounts to accounts in general.) The child object does almost everything the same way its parent does. The child has some special way to handle a few messages, different from the usual way (as the parent does it). But the child can explicitly decide to do something in the *usual* (parent-like) way, rather than in its own specialized way.

Multiple Superclasses

We can have object types that inherit methods from more than one type. We'll invent a `singer` class and then create `singer-TA`s and `TA-singer`s.

```
(define-class (singer)
  (parent (worker))
  (method (sing) '(tra-la-la)) )

(define-class (singer-TA)
  (parent (singer) (TA)) )

(define-class (TA-singer)
  (parent (TA) (singer)) )

> (define Matt (instantiate singer-TA))
> (define Chris (instantiate TA-singer))
> (ask Matt 'grade-exam)
A+
> (ask Matt 'sing)
(TRA-LA-LA)
> (ask Matt 'work)
WHISTLE-WHILE-YOU-WORK
> (ask Chris 'work)
(LET ME HELP YOU WITH THAT BOX AND POINTER DIAGRAM)
```

Both Matt and Chris can do anything a TA can do, such as grading exams, and anything a singer can do, such as singing. The only difference between them is how they handle messages that TAs and singers process *differently.* Matt is primarily a singer, so he responds to the work message as a singer would. Chris, however, is primarily a TA, and uses the work method from the TA class.

In the example above, Matt used the work method from the worker class, inherited through two levels of parent relationships. (The worker class is the parent of singer, which is a parent of singer-TA.) In some situations it might be better to choose a method inherited directly from a second-choice parent (the TA class) over one inherited from a first-choice grandparent. Much of the complexity of contemporary object-oriented programming languages has to do with specifying ways to control the order of inheritance in situations like this.

There are only three procedures that you need to use: `define-class`, which defines a class; `instantiate`, which takes a class as its argument and returns an instance of the class; and `ask`, which asks an object to do something. Here are the explanations of the procedures:

ASK: (ask *object message* . *args*)

`Ask` gets a method from *object* corresponding to *message*. If the object has such a method, invoke it with the given *args*; otherwise it's an error.

INSTANTIATE: (instantiate *class* . *arguments*)

`Instantiate` creates a new instance of the given *class*, initializes it, and returns it. To initialize a class, `instantiate` runs the `initialize` clauses of all the parent classes of the object and then runs the `initialize` clause of this class.

The extra arguments to `instantiate` give the values of the new object's instantiation variables. So if you say

```
(define-class (account balance) ...)
```

then saying

```
(define my-acct (instantiate account 100))
```

will cause `my-acct`'s `balance` variable to be bound to 100.

DEFINE-CLASS:
(define-class (*class-name args*...) *clauses*...)

This defines a new class named *class-name*. The instantiation arguments for this class are *args*. (See the explanation of `instantiate` above.)

The rest of the arguments to `define-class` are various *clauses* of the following types. All clauses are optional. You can have any number of `method` clauses, in any order.

> **(METHOD (*message arguments*...) *body*)**
>
> A `method` clause gives the class a method corresponding to the *message*, with the given *arguments* and *body*. A class definition may contain any number of `method` clauses. You invoke methods with `ask`. For example, say there's an object with a
>
> ```
> (method (add x y) (+ x y))
> ```
>
> clause. Then `(ask object 'add 2 5)` returns 7.
>
> Inside a method, the variable `self` is bound to the object whose method this is. (Note that `self` might be an instance of a child class of the class in which the method is defined.) A method defined within a particular class has access to the instantiation

variables, instance variables, and class variables that are defined within the same class, but does *not* have access to variables defined in parent or child classes. (This is similar to the scope rules for variables within procedures outside of the OOP system.)

Any method that is usable within a given object can invoke any other such method by invoking `(ask self` *message*`)`. However, if a method wants to invoke the method of the same name within a parent class, it must instead ask for that explicitly by saying

`(usual` *message args...*`)`

where *message* is the name of the method you want and *args...* are the arguments to the method.

(INSTANCE-VARS (*var1 value1*) (*var2 value2*) ...)

`Instance-vars` sets up local state variables `var1`, `var2`, etc. Each instance of the class will have its own private set of variables with these names. These are visible inside the bodies of the methods and the initialization code within the same class definition. The initial values of the variables are calculated when an instance is created by evaluating the expressions `value1`, `value2`, etc. There can be any number of variables. Also, a method is automatically created for each variable that returns its value. If there is no `instance-vars` clause then the instances of this class won't have any instance variables. It is an error for a class definition to contain more than one `instance-vars` clause.

(CLASS-VARS (*var1 value1*) (*var2 value2*) ...)

`Class-vars` sets up local state variables `var1`, `var2`, etc. The class has only one set of variables with these names, shared by every instance of the class. (Compare the `instance-vars` clause described above.) These variables are visible inside the bodies of the methods and the initialization code within the same class definition. The initial values of the variables are calculated when the class is defined by evaluating the expressions `value1`, `value2`, etc. There can be any number of variables. Also, a method is automatically created for each variable that returns its value. If there is no `class-vars` clause then the class won't have any class variables. It is an error for a class definition to contain more than one `class-vars` clause.

(PARENT (*parent1 args...*) (*parent2 args...*))

`Parent` defines the parents of a class. The *args* are the arguments used to instantiate the parent objects. For example, let's say that the `rectangle` class has two arguments: `height` and `width`:

```
(define-class (rectangle height width) ...)
```

A `square` is a kind of `rectangle`; the `height` and `width` of the `square`'s `rectangle` are both the `side-length` of the `square`:

```
(define-class (square side-length)
  (parent (rectangle side-length side-length))
  ...)
```

When an object class doesn't have an explicit method for a message it receives, it looks for methods of that name (or default methods, as explained below) in the definitions of the parent classes, in the order they appear in the `parent` clause. The method that gets invoked is from the first parent class that recognizes the message.

A method can invoke a parent's method of the same name with `usual`; see the notes on the `method` clause above.

(DEFAULT-METHOD *body*)

A `default-method` clause specifies the code that an object should execute if it receives an unrecognized message (i.e., a message that does not name a method in this class or any of its superclasses). When the body is executed, the variable `message` is bound to the message, and the variable `args` is bound to a list of the additional arguments to `ask`.

(INITIALIZE *body*)

The body of the `initialize` clause contains code that is executed whenever an instance of this class is created.

If the class has parents, their `initialize` code gets executed before the `initialize` clause in the class itself. If the class has two or more parents, their `initialize` code is executed in the order that they appear in the `parent` clause.

Object-Oriented Programming — Below the line view

This document documents the Object Oriented Programming system for CS 61A in terms of its implementation in Scheme. It assumes that you already know what the system does, i.e. that you've read "Object-Oriented Programming — Above the line view." Also, this handout will assume a knowledge of how to implement message passing and local state variables in Scheme, from chapters 2.3 and 3.1 of A&S. (Chapter 3.2 from A&S will also be helpful.)

Almost all of the work of the object system is handled by the special form `define-class`. When you type a list that begins with the symbol `define-class`, Scheme translates your class definition into Scheme code to implement that class. This translated version of your class definition is written entirely in terms of `define`, `let`, `lambda`, `set!`, and other Scheme functions that you already know about.

We will focus on the implementation of the three main technical ideas in OOP: message passing, local state, and inheritance.

Message Passing

The text introduces message-passing with this example from Section 2.3.3 (page 141):

```
(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-RECTANGULAR" m))))
  dispatch)
```

In this example, a complex number object is represented by a dispatch procedure. The procedure takes a *message* as its argument, and returns a number as its result. Later, in Section 3.1.1 (page 173), the text uses a refinement of this representation in which the dispatch procedure returns a *procedure* instead of a number. The reason they make this change is to allow for extra arguments to what we are calling the *method* that responds to a message. The user says

```
((acc 'withdraw) 100)
```

Evaluating this expression requires a two-step process: First, the dispatch procedure (named `acc`) is invoked with the message `withdraw` as its argument. The dispatch procedure returns the withdraw method procedure, and that second procedure is invoked with `100` as its argument to do the actual work. All of an object's activity comes from invoking its method procedures; the only job of the object itself is to return the right procedure when it gets sent a message.

Any OOP system that uses the message-passing model must have some below-the-line mechanism for associating methods with messages. In Scheme, with its first-class procedures, it is very natural to use a dispatch procedure as the association mechanism. In some other language the object might instead be represented as an array of message-method pairs.

If we are treating objects as an abstract data type, programs that use objects shouldn't have to know that we happen to be representing objects as procedures. The two-step notation for invoking a method violates this abstraction barrier. To fix this we invent the `ask` procedure:

```
(define (ask object message . args)
  (let ((method (object message)))     ; Step 1: invoke dispatch procedure
    (if (method? method)
        (apply method args)            ; Step 2: invoke the method
        (error "No method" message (cadr method)))))
```

`Ask` carries out essentially the same steps as the explicit notation used in the text. First it invokes the dispatch procedure (that is, the object itself) with the message as its argument. This should return a method (another procedure). The second step is to invoke that method procedure with whatever extra arguments have been provided to `ask`.

The body of `ask` looks more complicated than the earlier version, but most of that has to do with error-checking: What if the object doesn't recognize the message we send it? These details aren't very important. `Ask` does use two features of Scheme that we haven't discussed before:

The dot notation used in the formal parameter list of `ask` means that it accepts any number of arguments. The first two are associated with the formal parameters `object` and `message`; all the remaining arguments (zero or more of them) are put in a list and associated with the formal parameter `args`.

The procedure `apply` takes a procedure and a list of arguments and applies the procedure to the arguments. The reason we need it here is that we don't know in advance how many arguments the method will be given; if we said `(method args)` we would be giving the method *one* argument, namely, a list.

In our OOP system, you generally send messages to instances, but you can also send some messages to classes, namely the ones to examine class variables. When you send a message to a class, just as when you send one to an instance, you get back a method. That's why we can use `ask` with both instances and classes. (The OOP system itself also sends the class an `instantiate` message when you ask it to create a new instance.) Therefore, both the class and each instance is represented by a dispatch procedure. The overall structure of a class definition looks something like this:

```
(define (class-dispatch-procedure class-message)
  (cond ((eq? class-message 'some-var-name) (lambda () (get-the-value)))
        (...)
        ((eq? class-message 'instantiate)
         (lambda (instantiation-var ...)
           (define (instance-dispatch-procedure instance-message)
             (cond ((eq? instance-message 'foo) (lambda ...))
                   (...)
                   (else (error "No method in instance")) ))
           instance-dispatch-procedure))
        (else (error "No method in class")) ))
```

(Please note that this is *not* exactly what a class really looks like. In this simplified version we have left out many details. The only crucial point here is that there are two dispatch procedures, one inside the other.) In each dispatch procedure, there is a `cond` with a clause for each allowable message. The consequent expression of each clause is a `lambda` expression that defines the corresponding method. (In the text, the examples often use named method procedures, and the consequent expressions are names rather than `lambda`s. We found it more convenient this way, but it doesn't really matter.)

Local State

You learned in section 3.1 that the way to give a procedure a local state variable is to define that procedure inside another procedure that establishes the variable. That outer procedure might be the implicit procedure in the `let` special form, as in this example from page 171:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
         balance)
          "Insufficient funds"))))
```

In the OOP system, there are three kinds of local state variables: class variables, instance variables, and instantiation variables. Although instantiation variables are just a special kind of instance variable above the line, they are implemented differently. Here is another simplified view of a class definition, this time leaving out all the message passing stuff and focusing on the variables:

```
(define class-dispatch-procedure
  (LET ((CLASS-VAR1 VAL1)
        (CLASS-VAR2 VAL2) ...)
    (lambda (class-message)
      (cond ((eq? class-message 'class-var1) (lambda () class-var1))
            ...
            ((eq? class-message 'instantiate)
             (lambda (INSTANTIATION-VARIABLE1 ...)
               (LET ((INSTANCE-VAR1 VAL1)
                     (INSTANCE-VAR2 VAL2) ...)
                 (define (instance-dispatch-procedure instance-message)
                   ...)
                 instance-dispatch-procedure)))))))
```

The scope of a class variable includes the class dispatch procedure, the instance dispatch procedure, and all of the methods within those. The scope of an instance variable does not include the class dispatch procedure in its methods. Each invocation of the class `instantiate` method gives rise to a new set of instance variables, just as each new bank account in the book has its own local state variables.

Why are class variables and instance variables implemented using let, but not instantiation variables? The reason is that class and instance variables are given their (initial) values by the class definition itself. That's what let does: It establishes the connection between a name and a value. Instantiation variables, however, don't get values until each particular instance of the class is created, so we implement these variables as the formal parameters of a lambda that will be invoked to create an instance.

Inheritance and Delegation

Inheritance is the mechanism through which objects of a child class can use methods from a parent class. Ideally, all such methods would just be part of the repertoire of the child class; the parent's procedure definitions would be "copied into" the Scheme implementation of the child class.

The actual implementation in our OOP system, although it has the same purpose, uses a somewhat different technique called *delegation.* Each object's dispatch procedure contains entries only for the methods of its own class, not its parent classes. But each object has, in an instance variable, an object of its parent class. To make it easier to talk about all these objects and classes, let's take an example that we looked at before:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

Let's create an instance of that class:

```
(define Gerry-account (instantiate checking-account 20000))
```

Then the object named Gerry-account will have an instance variable named my-account whose value is an instance of the account class. (The variables my-whatever are created automatically by define-class.)

What good is this parent instance? If the dispatch procedure for Gerry-account doesn't recognize some message, then it reaches the else clause of the cond. In an object without a parent, that clause will generate an error message. But if the object does have a parent, the else clause passes the message on to the parent's dispatch procedure:

```
(define (make-checking-account-instance init-balance)
  (LET ((MY-ACCOUNT (INSTANTIATE ACCOUNT INIT-BALANCE)))
    (lambda (message)
      (cond ((eq? message 'write-check) (lambda (amount) ...))
            ((eq? message 'init-balance) (lambda () init-balance))
            (ELSE (MY-ACCOUNT MESSAGE)) ))))
```

(Naturally, this is a vastly simplified picture. We've left out the class dispatch procedure, among other details. There isn't really a procedure named make-checking-account-instance in the implementation; this procedure is really the instantiate method for the class, as we explained earlier.)

When we send `Gerry-account` a `write-check` message, it's handled in the straightforward way we've been talking about before this section. But when we send `Gerry-account` a `deposit` message, we reach the `else` clause of the `cond` and the message is delegated to the parent `account` object. That object (that is, its dispatch procedure) returns a method, and `Gerry-account` returns the method too.

The crucial thing to understand is why the `else` clause does *not* say

```
(else (ask my-parent message))
```

The `Gerry-account` dispatch procedure takes a message as its argument, and returns *a method* as its result. `Ask`, you'll recall, carries out a two-step process in which it first gets the method and then invokes that method. Within the dispatch procedure we only want to get the method, not invoke it. (Somewhere there is an invocation of `ask` waiting for `Gerry-account`'s dispatch procedure to return a method, which `ask` will then invoke.)

There is one drawback to the delegation technique. As we mentioned in the above-the-line handout, when we ask `Gerry-account` to `deposit` some money, the `deposit` method only has access to the local state variables of the `account` class, not those of the `checking-account` class. Similarly, the `write-check` method doesn't have access to the `account` local state variables like `balance`. You can see why this limitation occurs: Each method is a procedure defined within the scope of one or the other class procedure, and Scheme's lexical scoping rules restrict each method to the variables whose scope contains it. The technical distinction between *inheritance* and *delegation* is that an inheritance-based OOP system does not have this restriction.

We can get around the limitation by using messages that ask the other class (the child asks the parent, or *vice versa*) to return (or modify) one of its variables. The `(ask self 'withdraw ...)` in the `write-check` method is an example.

Bells and Whistles

The simplified Scheme implementation shown above hides several complications in the actual OOP system. What we have explained so far is really the most important part of the implementation, and you shouldn't let the details that follow confuse you about the core ideas. We're giving pretty brief explanations of these things, leaving out the gory details.

One complication is multiple inheritance. Instead of delegating an unknown message to just one parent, we have to try more than one. The real `else` clauses invoke a procedure called `get-method` that accepts any number of objects (i.e., dispatch procedures) as arguments, in addition to the message. `Get-method` tries to find a method in each object in turn; only if all of the parents fail to provide a method does it give an error message. (There will be a `my-`whatever variable for each of the parent classes.)

Another complication that affects the `else` clause is the possible use of a `default-method` in the class definition. If this optional feature is used, the body of the `default-method` clause becomes part of the object's `else` clause.

When an instance is created, the `instantiate` procedure sends it an `initialize` message. Every dispatch procedure automatically has a corresponding method. If the `initialize` clause is used in define-class, then the method includes that code. But even if there is no initialize clause, the OOP system has some initialization tasks of its own to perform.

In particular, the initialization must provide a value for the self variable. Every initialize method takes the desired value for self as an argument. If there are no parents or children involved, self is just another name for the object's own dispatch procedure. But if an instance is the my-whatever of some child instance, then self should mean that child. The solution is that the child's initialize method invokes the parent's initialize method with the child's own self as the argument. (Where does the child get its self argument? It is provided by the instantiate procedure.)

Finally, usual involves some complications. Each object has a send-usual-to-parent method that essentially duplicates the job of the ask procedure, except that it only looks for methods in the parents, as the else clause does. Invoking usual causes this method to be invoked.

A useful feature

To aid in your understanding of the below-the-line functioning of this system, we have provided a way to look at the translated Scheme code directly, i.e., to look at the below-the-line version of a class definition. To look at the definition of the class foo, for example, you type

```
(show-class 'foo)
```

If you do this, you will see the complete translation of a define-class, including all the details we've been glossing over. But you should now understand the central issues well enough to be able to make sense of it.

We end this document with one huge example showing every feature of the object system. Here are the above-the-line class definitions:

```
(define-class (person) (method (smell-flowers) 'Mmm!))
(define-class (fruit-lover fruit) (method (favorite-food) fruit))

(define-class (banana-holder name)
  (class-vars (list-of-banana-holders '()))
  (instance-vars (bananas 0))
  (method (get-more-bananas amount)
    (set! bananas (+ bananas amount)))
  (default-method 'sorry)
  (parent (person) (fruit-lover 'banana))
  (initialize
   (set! list-of-banana-holders (cons self list-of-banana-holders))) )
```

On the next page we show the translation of the banana-holder class definition into ordinary Scheme. Of course this is hideously long, since we have artificially defined the class to use every possible feature at once. The translations aren't meant to be read by people, ordinarily. The comments in the translated version were added just for this handout; you won't see comments if you use show-class yourself.

```
(define banana-holder
 (let ((list-of-banana-holders '()))                ;; class vars set up
  (lambda (class-message)                            ;; class dispatch proc
    (cond
     ((eq? class-message 'list-of-banana-holders)
      (lambda () list-of-banana-holders))
     ((eq? class-message 'instantiate)
      (lambda (name)                                 ;; Instantiation vars
        (let ((self '())                             ;; Instance vars
              (my-person (instantiate-parent person))
              (my-fruit-lover (instantiate-parent fruit-lover 'banana))
              (bananas 0))
          (define (dispatch message)                 ;; Object dispatch proc
            (cond
             ((eq? message 'initialize)              ;; Initialize method:
              (lambda (value-for-self)               ;;  set up self variable
                (set! self value-for-self)
                (ask my-person 'initialize self)
                (ask my-fruit-lover 'initialize self)
                (set! list-of-banana-holders         ;;  user's init code
                      (cons self list-of-banana-holders))))
             ((eq? message 'send-usual-to-parent)    ;; How USUAL works
              (lambda (message . args)
                (let ((method (get-method
                               'banana-holder
                               message
                               my-person
                               my-fruit-lover)))
                  (if (method? method)
                      (apply method args)
                      (error "No USUAL method" message 'banana-holder)))))
             ((eq? message 'name) (lambda () name))
             ((eq? message 'bananas) (lambda () bananas))
             ((eq? message 'list-of-banana-holders)
              (lambda () list-of-banana-holders))
             ((eq? message 'get-more-bananas)
              (lambda (amount) (set! bananas (+ bananas amount))))
             (else                                   ;; Else clause:
              (let ((method (get-method
                             'banana-holder
                             message
                             my-person
                             my-fruit-lover)))
                (if (method? method)                 ;; Try delegating...
                    method
                    (lambda args 'sorry))))))        ;; default-method

          dispatch)))                                ;; Class' instantiate
                                                     ;; proc returns object
     (else (error "Bad message to class" class-message))))))
```

UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61A **P. N. Hilfinger**

Highlights of GNU Emacs

This document describes the major features of GNU Emacs (called "Emacs" hereafter), a customizable, self-documenting text editor. In the interests of truth, beauty, and justice—and to undo, in some small part, the damage Berkeley has done by foisting `vi` on an already-unhappy world—Emacs will be the official CS61A text editor this semester.

Emacs carries with it on-line documentation of most of its commands, along with a tutorial for first-time users (see §7). Because this documentation is available, I have not attempted to present a complete Emacs reference manual here.

To run Emacs, simply enter the command `emacs` to the shell. Within Emacs, as described below, you can edit any number of files simultaneously, run UNIX shells, read and send mail, and run the Scheme interpreter to execute your programs. As a result, *it should seldom be necessary to leave Emacs before you are ready to logout and seldom necessary to create new windows.*

1 Basic Concepts

We'll begin with some fundamental definitions and notational conventions.

1.1 Buffers, windows, and what's in them

At any given time, Emacs maintains one or more *buffers* containing text. Each buffer may, but need not, be associated with a file. A buffer may be associated with a UNIX process, in which case the buffer generally contains input and output produced by that process (see, for example, sections 8 and 10). Within each buffer, there is a position called the *point*, where most of the action takes place.

Emacs displays one or more *windows* into its buffers, each showing some portion of the text of some buffer. A buffer's text is retained even when no window displays it; it can be displayed at any time by giving it a window. Each window has its own point (as just described); when only one window displays a buffer, its point is the same as the buffer's point. Two windows can simultaneously display text (not necessarily the same text) from the same buffer with a different point in each window, although it is most often useful to use multiple windows to display multiple files. At the bottom of each window, Emacs displays a *mode line*, which generally identifies the buffer being displayed and (if applicable) the file associated with it. At any given time, the *cursor*, which generally marks the point of text insertion, is in one of the windows (called the *current window*) at that window's point.

1.2 Commands

At the bottom of Emacs' display is a single *echo area*, displaying the contents of the *minibuffer*. This is a one-line buffer in which one types commands. It is, for many purposes, an ordinary Emacs buffer; standard Emacs text-editing commands for moving left or right and for inserting or deleting characters generally work in it. To issue a command by name, one types `M-x` ("meta-x"; this notation is described below) followed by the name of the command and `RET` (the return key); the echo area displays the command as it is typed. It is only necessary to type as much of the command name as suffices to identify it uniquely. For example, to run the command for looking at a UNIX manual entry—for which the full command is `M-x manual-entry`—it suffices to type `M-x man`, followed by a `RET`.

All Emacs commands have names, and you can issue them with `M-x`. You'll invoke most commands, however, by using control characters and escape sequences to which these commands are *bound*. Almost every character typed to Emacs actually executes a command. By default, typing any of the printable characters executes a command that inserts that character at the cursor. Many of the control characters are bound to commonly-used commands (see the quick-reference guide at the end for a summary of particularly important ones). At any time, it is possible to bind an arbitrary key or sequence of keys to an arbitrary command, thus *customizing* Emacs to your own tastes. Hence, all descriptions of key bindings in this document are actually descriptions of standard or default bindings.

1.3 Notations for special keys

In referring to non-graphic keys (control characters and the like), we'll use the following notations.

`ESC` denotes the escape character.

`DEL` denotes the delete character. On HP workstations, as we've set them up for this class, the '`Backspace`' key has the same effect.

`SPC` denotes the space character.

`RET` denotes the result of pressing the 'Return' key. (Confusingly, the result of typing this into a file is not a return character (ASCII code 13), but rather a linefeed character (ASCII code 10). Nevertheless, Emacs distinguishes the two keys.) On the HP workstations, this is the wide key labelled "Enter" in the main section of the keyboard.

`LFD` denotes the result of typing the linefeed key. On the HP workstations, this is the tall key labelled "Enter" in the numeric keypad at the far right of the keyboard.

`TAB` denotes the tab (also `C-i`) key.

C-α denotes "control-α"—the result of holding down the Control (or Ctrl) key while typing α.

M-α denotes "meta-α," which one gets either by typing the two-character sequence ESC followed by α, or (on our HP workstations when running the X window system) holding down either Alt key while typing α.

C-M-α denotes the result of typing the two-character sequence ESC C-α, or (on HP workstations when running X) holding down both Control and Alt simultaneously with typing α).

1.4 Command arguments

Certain commands take arguments, and take these arguments from a variety of sources. Any command may be given a numeric argument. To enter the number comprising the digits $d_0d_1\cdots d_n$ as a numeric argument (d_0 may also be a minus sign), type either 'M-$d_0d_1\cdots d_n$' or 'C-u$d_0d_1\cdots d_n$' before the command. When using C-u, the digits may be omitted, in which case '4' is assumed. The most common use for numeric arguments is as repetition counts. Thus, M-4 C-n moves down four lines and M-72 * inserts a line of 72 asterisks in the file. Other commands give other interpretations, as described below. In describing commands, we will use the notation *ARG* to refer to the value of the numeric argument, if present.

When commands prompt for arguments, Emacs will often allow provide a *completion* facility. When entering a file name on the echo line, you can usually save time by typing TAB, which fills in as much of the file name as possible, or SPC which fills in as much as possible up to a punctuation mark in the file name. Here, "as much as possible" means as much as is possible without having to guess which of several possible names you must have meant. A similar facility will attempt to complete the names of functions or buffers that are prompted for in the echo line.

1.5 Modes

The binding of keys to commands depends on the buffer that currently contains the cursor. This allows different buffers to respond to characters in different ways. In this document, we will refer to the set of key bindings in effect within a given buffer as the *(major) mode* of that buffer (the term "*mode*" is actually somewhat ill-defined in Emacs). A set of key bindings that simply modifies a few characteristics is called a minor mode.

Emacs will automatically establish a mode for buffers containing certain files depending on the name of their associated file. Thus, buffers start out in 'C' mode for files whose names end in '.c' or '.h'; 'C++' mode for .cc or .C; or 'Scheme' mode (see §9) for .scm. These modes affect the behavior of the TAB key, for example, causing program text to be indented according to the conventions for a particular programming language. The shell buffer runs in Shell mode, which (among many other things) causes the RET key to send the last line typed to the shell. Files with unclassifiable names generally start in Fundamental mode.

There is one useful minor mode that's worth knowing about.

`M-x auto-fill-mode` toggles (reverses the setting) of auto-fill mode, which by default is usually off. In auto-fill mode, lines get broken automatically as they are being typed when they get too long. When you are typing comments in C programs, auto-fill mode will automatically start a new comment on the next line when the current line gets near to filling up.

2 Important special-purpose commands

`C-g` quits the current command. Generally useful for cancelling a `M-x`-style command or other multi-character command that you have started entering. When in doubt, use it.

`C-x C-c` exits from Emacs. It prompts (in the echo area) if there are any buffers that have not been properly saved.

`C-x u` undoes the effects of the last editing command. If repeated, it undoes each of the preceding commands in reverse order (there is a limit). This is an extremely important command; be sure to master it. This does not undo other kinds of commands; the cursor may end up at some rather odd places.

`C-l` redraws the screen, and positions the current line to the center of the current window.

3 Basic Editing

The simple commands in this section will enable you to do most of the text entering and editing that you'll ordinarily need. Periodic browsing through the on-line documentation (see section 7.3) will uncover many more.

3.1 Simple text.

To enter text, simply position the cursor to the desired buffer and character position (using the commands to be described) and type the desired text. Carriage return behaves as you would expect. To enter control characters and other special characters as if they were ordinary characters, precede them with a `C-q`.

3.2 Navigation within a buffer.

The following commands move the cursor within a given buffer. Later sections describe how to move around between buffers.

`C-f` moves forward one character (at the end of a line, this goes to the next).

`C-b` moves backward one character.

`M-f` moves forward one “word.”

`M-b` moves backward one word.

`C-e` moves to the end of the current line.

`C-a` moves to the beginning of the current line.

`C-M-f` moves forward one Lisp (Scheme) S-expression.

`C-M-b` moves backward one Lisp (Scheme) S-expression.

`M-a` moves backward to next beginning-of-sentence. The precise meaning of "sentence" depends on the mode.

`M-{` moves backward to next beginning-of-paragraph. The precise meaning of "paragraph" depends on the mode.

`M-e` moves to the next end-of-sentence.

`M-}` moves to the next end-of-paragraph.

`C-n` moves down to the next line (at roughly the same horizontal position, if possible).

`C-p` moves up to the previous line.

`C-v` scrolls the text of the current window up roughly one window-full (i.e., exposes text *later* in the buffer). If *ARG* is supplied, it scrolls up *ARG* lines.

`M-v` scrolls the text of the current window down roughly one window-full (i.e., exposes text *earlier* in the buffer). If *ARG* is supplied, it scrolls down *ARG* lines.

`C-M-v` scrolls up the text in another window (if any) roughly one window-full. If *ARG* is supplied, it scrolls up *ARG* lines.

`M-<` moves to the beginning of the current buffer, after setting the mark (see §3.3) to the current point. If *ARG* is supplied, it moves to a point *ARG*/10 of the way through the buffer, instead of the beginning.

`M->` moves to the end of the current buffer. If *ARG* is supplied, it moves to a point *ARG*/10 of the way back from the end of the buffer, instead of the end.

`M-g` goes to the line number given by the argument (prompts for a number in the echo line, if you haven't supplied an argument).

`M-x what-line` displays the number of the current line in the current buffer.

3.3 Regions

In addition to a point (marked by the cursor in the current window), each buffer may contain a *mark*. Everything between the point and mark is called the *current region*. The current region typically delimits text to be manipulated by certain commands.

`C-@` sets the mark at the current point, and pushes the previous mark on a ring of marks. If *ARG* is present, it instead puts the point at the current mark and pops a new mark off this ring.

`C-SPC` is the same as `C-@`.

`C-x C-x` exchanges the point and the mark.

`M-@` sets the mark after the end of the next word.

`M-h` sets the region (point and mark) around the current paragraph.

`C-x h` sets the region (point and mark) around the entire current buffer.

3.4 Deletion

`DEL` deletes the character preceding the cursor. At the beginning of a line, it deletes the preceding end-of-line character, thus joining the current and preceding lines.

`M-DEL` deletes the word preceding the cursor. The deleted word moves to the kill buffer, described later.

`C-d` deletes the character under the cursor (which can be the end-of-line).

`M-d` deletes the word following the cursor.

`C-k` deletes the rest of the line following the cursor. If the cursor is on the end-of-line, delete the end-of-line. The deleted line moves to the kill buffer.

`M-\` deletes all horizontal blank space on either side of the cursor.

`M-SPC` deletes all but one horizontal blank space surrounding the cursor.

`C-x C-o` on non-blank line, deletes all immediately following blank lines; on isolated blank line, deletes the line; on other blank lines, deletes all but one.

`C-w` deletes everything between the point and the mark, moving the deleted text to the kill buffer.

`M-w` copies everything between point and mark to the kill buffer, without actually deleting it.

3.5 Insertion and the kill buffer

Several of the preceding commands mention the *kill buffer.* Text that is deleted is appended to the end of the current kill buffer, and can later be retrieved and inserted ("pasted" or "yanked") elsewhere in the text (even in another buffer different from its original source). Normally, each time a command that does not append to the kill buffer is executed, the current kill buffer is saved in a ring of kill buffers, and the next deletion command starts with an empty kill buffer. Hence, to move a sequence of lines, one can issue a sequence of `C-k` commands, with no intervening commands, move to the desired destination, and yank them back (with `C-y`).

`C-y` inserts the contents of the current kill buffer at the cursor, and moves cursor to end of inserted text. If a numeric value of *ARG* is supplied, inserts the ARG^{th} most recent kill buffer in the ring.

`C-u C-y` inserts current kill buffer, as for `C-y`, but leaves point unchanged.

`M-y` when issued *immediately* after a `C-y` or `M-y`, deletes the text inserted by the `C-y` or `M-y` and substitutes the text from the next kill buffer in sequence in the kill ring.

`C-M-w` causes the next command, if a kill command, to append to the end of previous kill buffer, rather than starting with a new one. This allows you, for example, to delete lines from several different places and then yank them back into one place.

3.6 Indentation

Indentation generally depends on the mode of the buffer. When a buffer is associated with a '.scm' file, in particular, it is by default in Scheme mode, in which the standard indentation referred to below is appropriate for Scheme source programs.

`TAB` indents as appropriate for the current mode. In text files, this is just an ordinary typewriter-style tab command. In Scheme source files, it indents to the appropriate point for a standard set of indentation conventions.

`LFD` is the same as `RET TAB`. Thus, if in typing in a Scheme program, you end each line with `LFD` instead of `RET`, your program will be indented as you enter it.

`M-;` indents for a comment according to the current mode. In Scheme mode, this inserts `;`.

`M-LFD` when used inside a comment, will close the comment, if necessary, go to a new line, and start a properly-indented comment on that line.

`C-x TAB` indents the current region "rigidly" by *ARG* spaces to the right (default 4). Negative arguments indent to the left. Tabs are correctly counted as the appropriate number of blanks.

`C-M-\` indents the current region according to the current mode. For an improperly-indented Scheme program, for example, this will correct all the indentation within the region.

3.7 Miscellaneous manipulations

`C-o` inserts a newline after the cursor. This has the same effect as `RET C-b` (return and then back up one character).

`C-t` transposes the character under the cursor with the preceding character. If an end-of-line is under the cursor, transposes the preceding two characters.

`M-t` transposes the next word that begins left of the cursor with the word following.

`C-x C-t` transposes the current and preceding lines.

`M-c` capitalizes the next word (making all characters other than the first lower case).

`M-u` converts the next word to all upper case.

`M-l` converts the next word to all lower case.

3.8 Using the mouse

When you are using Emacs with the X window system, you may use the mouse for simple positioning, text deletion, and text insertion. The three mouse buttons indicate the operation to be performed, and the mouse pointer (the slanting arrow, which we'll usually just call the *pointer*) usually indicates the position at which to perform it. In the following, the mouse buttons are called 'LB', 'MB', and 'RB', for left button, middle button, and right button. We'll use `C-`B to indicate the result of holding down "Control" while pushing button B.

`LB` places the point and mark at the position (and in the buffer) indicated by the pointer. You may then drag the mouse with LB depressed; this leaves the mark at the point you pressed `LB` and moves the point (and cursor) to the point at which you release `LB`, thus defining a new current region.

`RB` first extends the current region to include all the text between the existing current region (or the point, if there is no current region) and the pointer. Next, it copies the text in the current region into the kill buffer, as for `M-w` above. When clicked twice for the same text, it also deletes the text. Finally, it also copies the text into something called the *window-system cut buffer*. Text in the window-system cut buffer may be "pasted" (inserted) by `MB`, as described below, not only into Emacs buffers, but also into any other X-windows buffer.

`MB` pastes (inserts) text from the window system cut buffer at the point indicated by the mouse, and puts the cursor at the beginning and the mark at the end of the inserted text. This is somewhat like a mouse version of `C-y`. However, since it takes its text from the window system cut buffer (common to all windows on the screen), it allows the insertion of text from or to a window other than the one running Emacs.

`C-LB` Displays a menu of buffers to move to and allows you to select one (a mouse version of `C-x b`, described later).

You may also use the mouse to select from menus that sprout from the menu bar at the top of your Emacs screen. The content of these menus depends on the kind of buffer you are in.

4 Context searches

The search commands provide a convenient way to position the cursor quickly over long distances. One can search either for specific strings or for patterns specified by *regular expressions*. Both kinds of searches are carried out *incrementally*; that is, as you type in the target string or pattern, the cursor's position is continually changed to point to the first point in the buffer (if any) that matches what you have typed so far.

`C-s` searches forward incrementally.

`C-s C-s` is as for `C-s`, but initialize the search string to the one used in the last string search.

`C-M-s` is as for `C-s`, but searches for a regular expression.

`C-M-s C-s` As for `C-M-s`, but initialize the search pattern to the last pattern used.

`C-r` Search backward incrementally.

`C-r C-r` As for `C-r`, but initialize the search string as for `C-s C-s`.

`M-x occur` prompts for a regular expression and lists each line that follows the point and contains a match for the expression in a buffer. If you give an *ARG*, it will list that number of lines of context around each match.

`M-x count-matches` prompts for a regular expression and displays in the echo area the number of lines following the point that contain a match for it.

`M-x grep` prompts for arguments to the UNIX `grep` utility (which searches files for lines matching a given regular expression) and runs it asynchronously, allowing other editing while the search continues. See the command `` C-x ` `` in §10.1 for a description of how to look at each of the lines found in turn.

`M-x kill-grep` stops a `grep` that was started by `M-x grep`.

As you type the search string or pattern, the cursor moves in the appropriate direction to the first matching string, if any (specifically, to the right end of that string for a forward search and to the left end for a reverse search). By default, the case (upper or lower) of characters is ignored as long as the pattern you type contains no upper-case characters; 'a' will each match either 'a' or 'A'. When the pattern contains at least one upper-case character, the search becomes case-sensitive; 'a' will not match 'A', nor will 'A' match 'a'. If matching fails at any point, you will receive a message to that effect in the echo area. While entering a search string or pattern, certain command characters have altered effects, as follows.

`RET` ends the search, leaving the point at the string found, and setting the mark at the original position of the point.

`DEL` undoes the effect of the last character typed (and not previously DELed), moving the search back to wherever it was previously.

`C-g` aborts the search and returns the cursor to where it was at the beginning of the search.

`C-q` quotes the next character. That is, it causes the next character to be added to the search string or pattern as an ordinary character, ignoring any control action it might normally have. Use this, for example to search for a `C-g` character or, in a regular-expression search, to search for a '.'.

`C-s` begins searching forward at the point of the cursor for the next string satisfying the search string or pattern. If used in a reverse search, therefore, this reverses the sense of the search. If used at the point of a failing search, this starts the search over at the beginning of the buffer ("wraps around").

`C-r` is like `C-s`, but searches in the reverse direction, and can reverse the direction of a forward search.

`C-w` adds the next word beginning at the cursor to the end of the search string or pattern. It follows that this has the effect of moving the cursor forward over that word.

`LFD` adds the rest of the line to the end of the current search string or pattern.

Other control characters terminate the search, and then have their ordinary effect.

Ordinary searches (`C-s` and `C-r`) treat all ordinary characters as search characters. For regular-expression searches, several of these characters have special significance. See also the on-line documentation.

`.` matches any character, except end-of-line.

`^` matches the beginning of a line (that is, it matches the empty string, and only at the beginning of a line.)

`$` matches the end of a line.

`[···]` matches any of the characters between the square brackets. A range of characters may be denoted using '`-`', as in `[a-z0-9]`, which denotes any digit or letter. To include '`]`' as one of the characters, put it first. To include `'-'`, use `'---'`. To include '`^`', do *not* make it the first character.

`[^···]` matches any of the characters `not` included in the '···'. Thus, if end-of-line is not one of the characters, this will match it.

`*` when following another regular expression, denotes zero or more occurrences of that regular expression—in other words, an optional occurrence. This character applies to the immediately preceding regular expression; it has "highest precedence." There are special parentheses (see below) for cases where this is not what you want. Hence, the pattern '`.*`' denotes any number of characters, other than end-of-line. The pattern '`[a-z][a-z0-9_]*`' denotes a letter optionally followed by string of letters, digits, and underscores.

`+` is like '`*`', but denotes at least one occurrence. Thus, '`[0-9]+`' denotes an integer literal.

`?` is like '`*`', but denotes zero or one occurrence. Hence, the pattern '`[0-9]+,?`' denotes an integer literal optionally followed by a comma.

`\(···\)` groups the items '···'. Hence, '`\([0-9]+,\)?`' denotes an optional string consisting of an integer literal followed by a comma. The pattern '`\(01\)*` denotes zero or more occurrences of the two-character string '`01`'.

`\b` matches the empty string at the beginning or end of a word. Hence, '`\bring\b`' matches "ring" standing alone, but not "string" or "rings".

`\B` matches the empty string, provided that it is not at the beginning or end of a word.

`\|` matches a string matching either the regular expression to its left or to its right. Use '`\(\)`' to limit what regular expressions it applies to. Thus, '`\bf[a-z]+\|[0-9]+`' matches any integer literal or any word that begins with 'f', while '`\bf\([a-z]+\|[0-9]+\)`' matches any "word" that begins with 'f' and continues with either all letters or with all digits.

\\n where n is any digit, denotes the string that matched the pattern within the n^{th} set of '`\(\)`' brackets in the current regular expression. Thus, '`\b\([0-9]+\), *\1`' matches any integer literal that is followed by a comma, an optional space, and a repetition of the same literal; it matches "23, 23" and "10,10", but not "23, 24".

5 Replacement

The following commands allow you to do systematic replacement of one string or pattern with another within a given buffer.

`M-%` performs a query-replace operation. It prompts for a search string and a replacement string. Terminate each of the two with a `RET`. The command will then display each instance of the search string found, and prompt for its disposal. The options are described below. If *ARG* is supplied, it will only match things surrounded by word boundaries, so that if the search string is "top", there will be no replacement inside the string "stop" or "topping".

`M-X query-replace-regexp` is the same as `M-%`, but replaces patterns designated by regular expressions, rather than just simple strings. The replacement string may contain instances of '\\n', for n a digit, which, as described in the section on regular expressions, denotes the string matched by the n^{th} regular expression in '`\(\)`' braces in the search string. Thus, for example, the search pattern '`(\([a-z_][a-z0-9_]+\))`' with the replacement pattern '`[\1]`' will replace each C identifier surrounded by parentheses by the same identifier surrounded by square brackets.

By default, the replacement will preserve the case of the letters replaced if the search string or pattern has no upper-case letters, and otherwise will use the case supplied in the replacement string.

At each instance of the search string or pattern, you are prompted for an action. Here are some common ones.

`SPC` replaces the indicated occurrence and goes to the next.

`DEL` keeps the indicated occurrence unchanged and go to the next.

`RET` exits with no further replacements.

`,` makes one replacement, but waits for another `SPC` or `DEL` before moving to the next match.

`.` makes one replacement and then exits.

`!` replaces all remaining occurrences without prompting again.

`?` prints a help message.

`C-r` enters a recursive edit level. That is, you are put back in ordinary Emacs at the point of the current occurrence and can edit in the usual manner. Typing `C-M-c` then goes back to the query-replace command.

`y` same as `SPC`.

`n` same as `DEL`.

`q` same as `RET`.

In addition to replacement, there are two often-useful commands for deleting selected lines.

`M-x delete-matching-lines` prompts for a regular expression and deletes (*without* prompting) each line after the point that contains a match for it.

`M-x delete-non-matching-lines` prompts for a regular expression and deletes each line after the point that does not contain a match for it.

6 Files, buffers, and windows

Each buffer has a name. By default, buffers that are associated with particular files have the name of that file (not including the name of the directory containing it), possibly followed by a number in angle brackets to distinguish multiple files (from different directories with the same name.

6.1 Loading into and storing from buffers

`C-x C-f` prompts for a file name and sets the current window to displaying that file in a buffer having the same name. If a buffer displaying that file already exists, this command merely switches the window to that buffer. If the file does not exist, the buffer is initially empty. The buffer is subsequently associated with the file. This process is called *finding* the file.

`C-x 4 C-f` prompts for a file name, goes to the next window on the screen (creating a new one, if there is only one), and then acts like `C-x C-f`.

`C-x C-s` saves the current buffer in its associated file, if the buffer has been modified. If the file being saved exists, then the old version is first renamed to have a tilde (˜) appended to its name, if no such file yet exists.

`C-x C-w` prompts for a file name and saves the current buffer into that file. Generally, it is preferable and safer to use `C-x C-f` or `C-x 4 C-f` and then use `C-x C-s`, but sometimes this command is handy.

`C-x i` prompts for a file name and inserts that file at the point. It does not associate the inserted file with the current buffer.

`M-x revert-buffer` throws away the contents of the current buffer and restores the contents of the associated file. It will ask you to confirm these actions before taking them.

6.2 Manipulating buffers and windows

`C-x o` makes another window on the screen (if any) the current window.

`C-x 0` deletes the current window, expanding another window to take its place. The buffer being displayed in the current window is not affected.

`C-x 1` makes the current window the only window on the screen, deleting all others. The buffers being displayed in the deleted windows are not affected.

`C-x 2` splits the current window into two vertically (one on top of the other), both displaying the same buffer.

`C-x 3` splits the current window into two horizontally (beside each other), each displaying the same buffer.

`C-x b` prompts for a buffer name and switches the current window to that buffer. When trying to move to a buffer associated with a file, it is better to use the file finding commands.

`C-x C-b` lists the active buffers in a window.

`C-x k` prompts for a buffer name and deletes that buffer, displaying some other buffer in the current window. You will be warned if the contents of the buffer have been modified and not yet saved.

6.3 Auto-saving and recovery

Buffers that are associated with files are periodically saved ("auto-saved") in files whose names begin and end with '`#`'. After a crash, you can return yourself to the point at which the last auto-save of a given file took place by using the following command in place of `C-x C-f` or `C-x 4 C-f`.

`M-x recover-file` prompts for a file name, *F*. It then tries to recover the contents of that file from an auto-save file (named `#`*F*`#`) in the same directory, if such a file exists and is younger than the any file named *F* in the directory. After completing this command, `C-x C-s` will save the recovered file to *F*.

7 On-line documentation

7.1 UNIX documentation

Emacs has a simple interface to the standard UNIX ‘`man`’ command, which provides documentation to UNIX commands:

`M-x manual-entry` prompts for a topic (a UNIX command or subprogram name, usually), and displays the man page for it, if any, in a buffer. The buffer is a perfectly ordinary buffer; you may put the cursor in it and move around using ordinary Emacs navigational commands.

7.2 Basic Emacs help

The help command, `C-h`, provides a variety of useful documentation. The character following `C-h` indicates the specific kind of service desired; the descriptions of several of these follow.

`C-h a` prompts for a pattern (regular expression) and displays a buffer containing all commands whose name contains a match to that pattern, together with a short description and the key sequence to which the command is bound, if any.

`C-h b` displays a buffer containing all bindings of commands to keys. The display is in two parts: the *global bindings* that apply by default in any buffer, and the *local bindings* that apply only when one is in the current buffer, and override any global binding in that buffer.

`C-h f` prompts for a function name and then displays its full documentation in a buffer.

`C-h C-h` documents the help command itself.

`C-h i` runs the ‘info’ documentation reader (see below).

`C-h k` prompts for a command key sequence and describes the function invoked by that sequence.

`C-h m` prints documentation about the mode of the current buffer.

`C-h t` puts you into an Emacs tutorial.

`C-h w` prompts for a function name and tells what key, if any, invokes it.

7.3 The info browser

The key sequence `C-h i` invokes the documentation browsing system, `info`. Actually, this is little more than a buffer with some special bindings to the keys. Aside from the special bindings, the ordinary Emacs commands will work while inside the `info` buffer. At any time, the `info` buffer, whose name is `*info*`, contains a *node*, a section of text documenting something. These nodes are connected to each other in such a way that one can move quickly from one node to another that covers a related topic. Some nodes contain *menus*, indicated by lines that begin

```
  * Menu:
```

The lines after this give the names of other nodes, and descriptions of their contents. One such entry reads as follows.

```
  * Commands::      Named functions run by key sequences to do editing.
```

The word(s) between the asterisk and the double-colon name another node. The following key commands, defined only when in the buffer `*info*`, allow one to move through the documentation. They are only a few of the ones provided.

`m` prompts for the name of a node from the menu in the current buffer and displays that node. You need only enter enough to identify the desired entry unambiguously; case is ignored.

`f` follows a cross-reference. Cross references are indicated in the text of a node by a phrase of the form "`* Note` *foo*`::`". One follows them by typing 'f' followed by the name (*foo*) of the referenced node, as for the 'm' command.

`l` goes back to the last-visited node.

`u` goes up to the parent of this node. The definition of parent is actually arbitrary, but is usually a node that contains the current one in its menu.

`d` returns to the top (initial) node of the Info system.

`q` suspends the browser and goes back to where you were when you issued `C-h i`.

`.` returns to the beginning of the text of the current node.

`?` furnishes help about the browser commands.

8 The shell

It is possible to run a UNIX shell under Emacs, and this allows any number of useful effects. The command `M-x shell` moves to a buffer named `*shell*` that is running a UNIX shell (creating it if necessary). Anything typed into this buffer is sent to the shell, just it would be outside of Emacs. Any output produced as a result of the commands sent to the shell is placed at the end of the shell buffer. Because the shell is running in an Emacs window, the contents of the shell can be edited and navigated freely, and the entire record of the input and output to the shell is available at all times. A few keys have slightly different-from-usual meanings in the shell buffer.

`RET` sends whatever line the cursor is on to the shell and moves to the end of the shell buffer. Hence, one can repeat a command by placing the cursor anywhere in it and typing `RET`.

`TAB` attempts to complete the immediately preceding file name.

`C-c C-c` is the same as a single `C-c` outside Emacs.

`C-c C-d` is the same as `C-d` (end-of-file) outside Emacs.

`C-c C-z` is the same as `C-z` outside Emacs.

`C-c C-u` kills the current line of input to the shell.

It is sometimes useful to run a single shell command over a region of text in a buffer.

`M-|` prompts for a shell command and executes it, giving the current region as the standard input. If the `M-|` is preceded by `C-u`, the output of the command replaces the region. Otherwise, the output goes to a separate buffer. For example, to sort the lines in the current region, enter the command `C-u M-| sort`.

9 Running Scheme under Emacs

The best way to run Scheme from a workstation is to do so through Emacs. Just as you can create an Emacs buffer for communicating with a UNIX shell (§8), you can also do so to communicate with a Scheme interpreter. Not only can you interact with the interpreter, but you can also feed files or definitions that you are editing to a running interpreter conveniently without having to load them explicitly.

The command `M-x run-scheme` moves to a buffer named `*scheme*` that is running the Scheme interpreter, creating this buffer if necessary. Each line that you type into this buffer gets sent to the interpreter, just as if you had typed it in while running the interpreter outside of Emacs. Any output from the interpreter in response to your input is appended to the `*scheme*` buffer.

The usual way to create and execute a Scheme program is as follows.

- Using Emacs, create a file to contain your program (or load one that you've already started) using `C-x C-f` or `C-x 4 C-f`; let's suppose the file is named `something.scm` (so that within Emacs, it lives in a buffer of the same name). We have configured Emacs so that any file ending in `.scm` gets edited in Scheme mode, which gives a special meaning to the keys `TAB`, `LFD`, and others described below.

- Edit or add to your file as needed. When typing definitions into the Emacs buffer for `something.scm`, using the `TAB` key at the beginning of each line will automatically indent that line properly. Alternatively, you can end each line by typing `LFD` instead of `RET`; in Scheme mode, `LFD` is short for `RET TAB`. If in the process of editing the buffer, you mess up the indentation of a definition, place the cursor at the beginning of the definition (on or before the opening '(') and type `M-C-q`, which will correctly indent the entire definition.

- Make sure you have a Scheme buffer (named `*scheme*`) running under Emacs (`M-x run-scheme`) will create one if you don't).

- In the buffer for `something.scm`, type `C-c M-l` to load your program into the running Scheme interpreter. Emacs will ask you for a file name; just type `RET`, which will use `something.scm`. If you haven't saved your changes to `something.scm`, Emacs will ask if it should do it for you. The effect of `C-c M-l` is to send the command `(load "something.scm")` to the Scheme interpreter and also to put the cursor in the `*scheme*` buffer, ready to enter Scheme expressions. You'll see the usual response to the `load` command in the `*scheme*` buffer.

- Sometimes—especially when you are correcting a file whose contents you've already loaded into Scheme—it is convenient to send just a single revised definition to the Scheme interpreter. To so do, place the cursor at the beginning of the definition (on or before the opening '(') and type `C-c M-e`. This also puts you into the Scheme buffer.

Here is a concise summary of the Scheme-related commands. These commands are also available from the menu bar. With the exception of `M-x` commands, all of these commands are in effect only in buffers that are in Scheme mode (normally, those containing files whose name ends in `.scm`).

`M-x run-scheme` when used for the first time, creates a buffer named `*scheme*` and runs the Scheme interpreter in it, displaying input from you and output from the interpreter. If the buffer already exists, this command simply moves to that buffer.

`C-c C-z` puts the cursor in the `*scheme*` buffer.

`C-c M-e` sends the definition after the cursor to Scheme (that is, it copies it the `*scheme*` buffer and then sends it to the Scheme interpreter that attached to that buffer). The command also places the cursor at the end of the `*scheme*` buffer.

`C-c C-e` is the same as `C-c M-e`, but leaves the cursor where it is.

`C-c M-l` loads an entire file into Scheme Prompts for a file name; the default is the current buffer's file. Puts the cursor at the end of the `*scheme*` buffer.

`C-c C-l` is the same as `C-c M-l`, but leaves the cursor where it is.

`C-c M-r` sends all the text in the current region to Scheme and puts the cursor at the end of the `*scheme*` buffer.

`C-c C-r` is the same as `C-c M-r`, but leaves the cursor where it is.

`M-C-q` indents the definition after the cursor according to the usual rules for indenting Scheme expressions.

`M-C-\` indents all Scheme expressions in the current region.

`TAB` indents the current line of Scheme code as appropriate for the surrounding context.

`LFD` is the same as `RET TAB`.

10 Compiling, debugging, and tags

[This section is not relevant to CS61A.] Emacs provides rather nice ways of compiling programs, correcting any compilation errors, and debugging the results. It is so much more convenient than entering compilation commands directly from a shell that there is no excuse not to use it.

10.1 Compilation

`M-x compile` prompts for a shell command, and then executes that command in a special buffer, named `*compilation*`. The current file at the time the `M-x compile` is issued determines the directory in which the shell command executes. The default command is simply `make -k`. Assuming you follow the convention of putting an appropriate `make` input file named `makefile` or `Makefile` in each source directory, this command will generally do the right thing. While the compilation proceeds, you are free to edit or use the `*shell*` buffer.

`` C-x ` `` finds the next error message in the buffer `*compilation*` (if any), finds the source files and line referred to by the error message, and displays the error message in one window and the source file in another. Thus, after a compilation is complete (actually, even while it proceeds), you can step through the error messages produced, going automatically to the offending points in the source file so that they can be corrected. The buffer `*compilation*` also contains the output from the `M-x grep` command described in §4.

`M-x kill-compiler` cancels a compilation started by `M-x compile`, if any.

10.2 Using GDB under Emacs

[This section is not relevant to CS61A.] The GNU debugger, GDB, is an interactive source-level debugger for C and several other languages. It can be run under Emacs, which provides a few rather nifty additional features. Full on-line documentation of gdb is available using the `C-h i` command in Emacs. The command `M-x gdb` will prompt for an executable file name, and then run GDB on that file, displaying the interaction in a buffer that acts much like a shell buffer described previously. Within that buffer, however, several commands have a slightly different meaning. In addition, whenever GDB displays the current position in the program (for example, after a step, at a breakpoint, or after an interrupt), Emacs will try to display the indicated source file and line in another window, with an arrow ('`=>`') pointing at the corresponding line in the source text (this arrow is not actually in the file being displayed).

The following commands are peculiar to GDB buffers.

`C-c C-n` performs a GDB 'next' command (step to next line in the source program).

`C-c C-s` performs a GDB 'step' command (step to next line in the source program to be executed, stopping at the beginning of any procedure that gets called.)

`C-c C-i` performs a GDB 'stepi' command (step to next machine-language instruction—not usually used unless you are programming in assembly language.

`C-c <` performs a GDB 'up' command (go up to procedure that called current one).

`C-c >` performs a GDB 'down' command (opposite of 'up').

`C-c C-r` performs a GDB 'finish' command (continues from last breakpoint).

`C-c C-b` set a breakpoint at the current position in the program (as indicated by the position of the '`=>`' arrow).

`C-c C-d` delete a breakpoint (if any) at the current position in the program (as indicated by the position of the '`=>`' arrow).

In addition, within any source file buffer, there is the following command.

`C-x SPC` puts a break point at the point in the program indicated by the cursor.

10.3 Tags

In UNIX terminology, a *tag table* is an index that tells how to find the definition of any certain identifiers ('tags') defined in some collection of source files. In effect, it provides a smart, multi-file search that is particularly useful when navigating in non-trivial directories of source files. Typically, you set things up by going into the directory containing the source text to be indexed and issuing the UNIX command

`etags` *options files*

where *files* is a list of all the source files that need to be indexed. This creates a file named 'TAGS' containing the tag table. For C programs, the tags are the names of functions defined in the named source files. The `-t` option causes `etags` to record **typedef** declarations as well. The tag table produced is organized in such a way that simple edits to a source file will not invalidate it. The following Emacs commands deal with tag tables.

`M-x visit-tags-table` prompts for the name of a tags table file, and uses its contents in future tag-related searches.

`M-.` prompts for a tag and then positions the current window in the file containing its first definition and puts the cursor on that definition. You may also give a null response (just `RET`), in which case the word before or around the point is used as the tag.

`C-u M-.` finds the next alternate definition of the last tag specified.

`C-x 4 .` is the same as `M-.`, but displays the text containing the tag in the other window instead of the current one.

`M-x tags-search` prompts and searches for a regular expression as for `C-M-s`, but is does a non-incremental search through all the files given in the currently-visited tag table.

`M-x tags-query-replace` acts like `M-Q`, but looks through all the files given in the currently-visited tag table.

`M-,` restarts the last `tags-search` or `tags-query-replace` from the current location of the point.

`M-x tags-apropos` prompts for a regular expression and displays a list of all tags in the currently-visited table that match it.

11 But wait; there's more!

As indicated at the beginning, this is not a complete reference manual. It has not covered scrolling sideways, tab setting, the mail system, the Emacs internal Lisp dialect, automatic abbreviation, the spelling checker, the directory editor, the change-log editor, or how to replace all groups of lines of your program that are indented more than *ARG* spaces by '...'[1]. You can learn about these and other topics by using `C-h i`. You might also try typing `C-h f SPC C-x o`, which creates a buffer containing the names of all Emacs functions and then puts the cursor there so that you can scroll through and look for likely-sounding names.

Just use it. Every session is an adventure.

[1] You probably think I'm kidding, don't you? Guess again.

GNU Emacs
Quick Reference Guide
for CS61A

Bullets (●) mark a suggested starting set of commands. Daggers (†) denote key bindings that are not standard in GNU Emacs. *ARG* denotes the prefix numeric argument (entered with `C-u` or `M-`*digit*). The notation '`C-`*x*' means "control-*x*", the result of holding down the control key while typing *x*. '`M-`*x*' means "meta-*x*", the result of holding down the '`Meta`', `Alt`, or ◇ key (depending on keyboard) while typing *x*. If you are not using a window system or have a keyboard without these keys, the sequence of two characters `ESC` *x* is equivalent. The notation '`C-M-`*x*' is equivalent to holding down both control and meta keys while typing *x*, or of typing the two characters `ESC C-`*x*.

Cursor motion.

`C-f`	Forward character.●
`C-b`	Backward character.●
`M-f`	Forward word.●
`M-b`	Backward word.●
`C-e`	Forward to end of line.●
`C-a`	Backward to start of line.●
`C-M-f`	Forward S-expression.●
`C-M-b`	Backward S-expression.●
`M-e`	Forward sentence.
`M-[`	Forward paragraph.
`M-a`	Backward sentence.
`M-]`	Backward paragraph.

`C-n`	Next line.●
`C-p`	Previous line.●
`M-<`	Beginning of buffer.●
`M->`	End of buffer.●
`C-v`	Scroll text up one screen (or *ARG* lines).●
`M-v`	Scroll text down (or *ARG* lines).●
`M-g`	Go to line number *ARG*.†
`M-x what-line`	Display line number.
`C-M-v`	Scroll other window up one screen (or *ARG* lines).

Marking regions of text

`C-@`	Set mark at point.●
`C-x C-x`	Exchange mark and point.●
`C-SPC`	Same as `C-@`.
`M-@`	Set mark after end of next word.
`M-h`	Set mark and point around current paragraph.
`C-x h`	Set mark and point around current buffer.

Deletion and yanking

`DEL`	Delete character before cursor.●
`M-DEL`	Delete word before cursor and add to kill buffer.●
`C-d`	Delete character at cursor.●
`M-d`	Delete word at and after cursor and add to kill buffer.●
`C-k`	Delete to end of line and add to kill buffer.●
`C-w`	Delete current region, and add to kill buffer.●
`M-w`	Copy current region to kill buffer without deleting.●
`M-\`	Delete surrounding blanks and tabs.
`M-SPC`	Delete all but one surrounding blank.
`C-x C-o`	Delete all but one surrounding blank line.
`C-M-w`	Cause next command, if a kill, to append to previous kill buffer, instead of new one.

`C-y`	Insert text from kill buffer at point.●
`C-u C-y`	Insert text from kill buffer at point without moving point.
`M-y`	Replace preceding `C-y` text with next most recent kill buffer.
`M-w`	Copy region to kill buffer, no deletion.

Indentation

`TAB`	Indent according to mode.●
`LFD`	Same as RET TAB.
`M-;`	Indent and start comment.
`M-LFD`	Continue comment on next line.
`C-x TAB`	Indent region rigidly by *ARG*.
`C-M-\`	Indent region according to mode.●

Search

`C-s`	Search forward.●
`C-s C-s`	Same as `C-s` with last string.●
`C-r`	Search backward.●

`C-r C-r`	Same as `C-r` with last string.●
`C-u C-s`	Search forward for regular expression.
`M-x occur`	Display lines matching a regular expression.
`M-x grep`	Display results of UNIX `grep` utility.
`M-x count-matches`	

The following subcommands are valid during a search.

`RET`	End search.●
`DEL`	Undo effect of last search character typed.●
`C-g`	Abort search.●
`C-s`	Search for next match forward.●
`C-r`	Search for next match backward.●
`C-q`	Quote next character.
`C-w`	Extend search string with next word.
`LFD`	Extend search string with rest of line.

Replacement

`M-%`	Query replace.●
`M-x delete-matching-lines`	
`M-x delete-non-matching-lines`	

During a query-replacement, the following are valid responses to prompts.

`SPC`	Make replacement and go to next.●
`DEL`	Skip replacement and go to next.●
`RET`	End replacement.●
`!`	Replace all remaining instances without asking.●
`C-r`	Enter recursive edit; return with `C-M-c`.

Regular expressions

`.`	Match any character.●
`^`	Match at start of line.●
`$`	Match at end of line.●
`[...]`	Match any character in the '...'.●
`[^...]`	Match any character except those in '...'.●
`*`	Match 0 or more of pattern to left.●
`+`	Match 1 or more of pattern to left.
`?`	Match 0 or 1 of pattern to left.
\c	Quotes *c*, except for the following.
`\b`	Match at beginning or end of word.
`\B`	Match except at beginning or end of word.
`\\|`	Match either pattern to left or right.
`\(...\)`	Grouping.
\n	Match copy of whatever matched n^{th} group.

Miscellaneous editing

`C-o`	Insert newline after cursor.
`C-t`	Transpose characters.
`M-t`	Transpose words.
`C-x C-t`	Transpose lines.
`M-u`	Convert whole word to upper case.
`M-l`	Convert whole word to lower case.
`M-c`	Capitalize word.

Files

`C-x C-f`	Find file; load if needed.●
`C-x 4 C-f`	Find file in other window.●
`C-x C-s`	Save file.●
`C-x C-w`	Write to explicitly-named file.
`C-x i`	Insert file at cursor.
`M-x recover-file`	Recover file after disaster from auto-save file.
`M-x revert-buffer`	Throw away changes to buffer and restore from file.

Buffers and windows

`C-x o`	Put cursor in other window.●
`C-x 1`	Grow current window to full screen.●
`C-x 2`	Split current window vertically.●
`C-x b`	Put named buffer in window.●
`C-x 0`	Remove current window.
`C-x 3`	Split current window horizontally.
`C-x C-b`	List all buffers.
`C-x k`	Delete buffer.

Shells

`M-x shell`	Run UNIX shell in a buffer.●
`M-|`	Execute single shell command on region. With *ARG*, replaces region.

Commands active in shell buffers:

`RET`	Send current line to shell.●
`TAB`	Complete preceding file name.●
`C-c C-c`	Send interrupt to shell.●

C-c C-u	Erase current input line.●
C-c C-z	Send stop signal to shell.●
C-c C-d	Send EOF to shell.

Scheme

M-x run-scheme	Run Scheme interpreter in buffer *scheme*.●
C-c C-z	Put the cursor in buffer *scheme*.●
C-c C-e	Send definition to *scheme*.
C-c M-e	Same as C-c C-e C-c C-z.
C-c C-l	Load file into *scheme*.
C-c M-l	Same as C-c C-l C-c C-z.●†
C-c C-r	Send current region to *scheme*.
C-c M-r	Same as C-c C-r C-c C-z.
M-C-q	Indent Scheme expression.●
M-C-\	Indent current region.
TAB	Indent the current line.●
LFD	Same as RET TAB.

Compilation, debugging, and tags

M-x compile	Execute command (by default, make) asynchronously.
C-x ‘	Position to next error or next line found by M-x grep command.
M-x kill-compiler	Stop active compile.

M-x visit-tags-table	Specify file containing tags produced by etags.
M-.	Display source for given tag.
C-u M-.	Find next alternate definition for last tag.
C-x 4 .	Display source for tag in other window.
M-x tags-search	
M-x tags-query-replace	Look for pattern in all files named in tags table.
M-x tags-apropos	Display matching tags.

M-x gdb	Run GNU debugger on file.

Commands valid in gdb mode.

C-c C-s	Step.
C-c C-n	Next.
C-c <	Up stack.
C-c >	Down stack.
C-c C-r	Finish.
C-x SPC	In any source file, sets a break point.
C-c C-i	Stepi.

Help and documentation

M-x manual-entry	UNIX man page for given topic.
C-h a	Look up names of matching Emacs commands.●
C-h b	Display key bindings.●
C-h f	Help for M-x function.●
C-h C-h	Help for C-h.●
C-h i	Run info browser.●
C-h k	Help for key.●
C-h m	Help for current mode.
C-h t	Tutorial.
C-h w	Key containing function.

Inside an *info* buffer (result of C-h i), the following are defined.

m	Select menu item.●
l	Go to last-visited node.●
?	Get help for browser.●
u	Go to node's parent.●
n	Go to next node in sequence.●
q	Leave browser.●
.	Go to top of node.
d	Go to top-level node.

Mouse commands

Left, middle, and right buttons are LB, MB, and RB.

LB	Put cursor at mouse. Dragging marks region.●
MB	Paste text from window-system cut buffer at mouse.●
RB	Extend region to pointer and copy into cut and kill buffers. Clicking twice deletes region.●
C-LB	Select a buffer.

Errata for Structure and Interpretation of Computer Programs · 2nd edition

Positive line numbers count from the top of the page or exercise, negative line numbers count from the bottom.

Page 45, line -13: Exponent should be n/2, not b/2

Page 112, line 2 of exercise 2.30: Square-LIST should be square-TREE. ("That is, square-tree should behave as follows:")

Page 118, lines 1-2: Should say "...the product OF THE SQUARES of the odd integers..."

Page 176, before procedures rectangular? and polar?: Should say "rectangular and polar numbers, respectively"

Page 181, line -5: Should not refer to exercise 3.24, just to section 3.3.3.

Page 185, exercise 2.73a: Should ask about VARIABLE?, not SAME-VARIABLE?

Pages 246 and 247, figures 3.7 and 3.8: There is an extra ')' at the end of the code.

Page 287, figure 3.28: Rightmost box should have +, not *

Page 324, exercise 3.50: Should refer to section 2.2.1, not 2.2.3.

Page 341, line 3 of exercise 3.66: Should say "For example, APPROXIMATELY how many pairs..."

Page 375, line 1 of exercise 4.7: Last LET should be LET* ("...bindings of the let* variables...")

Last updated 08/09/99

BERKELEY SCHEME EXTENSIONS:
WORD AND SENTENCE MANIPULATION PROCEDURES

The first chapter of the textbook deals exclusively with numeric data. To allow some variety, with interesting examples that aren't about calculus, we are going to use some additional Scheme procedures that manipulate linguistic data: words and sentences. A word can be considered as a string of characters, such as letters and digits. (Numbers can be treated as words.) A sentence is a string of words in parentheses.

PROCEDURES TO TAKE APART WORDS AND SENTENCES:

FIRST returns the first character of a word, or the first word of a sentence

BUTFIRST returns all but the first character of a word, or all but the first word of a sentence

BF same as BUTFIRST

LAST returns the last character of a word, or the last word of a sentence

BUTLAST returns all but the last character of a word, or all but the last word of a sentence

BL same as BUTLAST

Examples:

```
> (first 'hello)
h

> (butlast '(symbolic data are fun))
(symbolic data are)
```

PROCEDURES TO COMBINE WORDS AND SENTENCES

WORD arguments must be words; returns the word with all the arguments strung together

SENTENCE returns the sentence with all the arguments (words or sentences) strung together

SE same as SENTENCE

Examples:

```
> (word 'now 'here)
nowhere

> (se 'lisp '(is cool))
(lisp is cool)
```

PREDICATE PROCEDURES

EQUAL? returns true if its two arguments are the same word or the same sentence (a one-word sentence is not equal to the word inside it)

MEMBER? returns true if the first argument is a member of the second; the members of a word are its letters and the members of a sentence are its words

EMPTY? returns true if the argument is either the empty word [which can be represented as ""] or the empty sentence [which can be represented as '()]

MISCELLANEOUS

COUNT returns the number of letters in the argument word, or the number of words in the argument sentence.

ITEM takes two arguments: a positive integer N, and a word or sentence; returns the Nth letter of the word, or the Nth word of the sentence (counting from 1).

Examples:

```
(define (buzz n)
  (cond ((member? 7 n) 'buzz)
     ((= (remainder n 7) 0) 'buzz)
     (else n) ))

(define (plural wd)
  (if (equal? (last wd) 'y)
      (word (bl wd) 'ies)
      (word wd 's) ))
```

Revised^5 Report on the Algorithmic Language Scheme

RICHARD KELSEY, WILLIAM CLINGER, AND JONATHAN REES (*Editors*)

H. ABELSON
N. I. ADAMS IV
D. H. BARTLEY
G. BROOKS
R. K. DYBVIG
D. P. FRIEDMAN
R. HALSTEAD
C. HANSON
C. T. HAYNES
E. KOHLBECKER
D. OXLEY
K. M. PITMAN
G. J. ROZAS
G. L. STEELE JR.
G. J. SUSSMAN
M. WAND

Dedicated to the Memory of Robert Hieb

20 February 1998

SUMMARY

The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.

The introduction offers a brief history of the language and of the report.

The first three chapters present the fundamental ideas of the language and describe the notational conventions used for describing the language and for writing programs in the language.

Chapters 4 and 5 describe the syntax and semantics of expressions, programs, and definitions.

Chapter 6 describes Scheme's built-in procedures, which include all of the language's data manipulation and input/output primitives.

Chapter 7 provides a formal syntax for Scheme written in extended BNF, along with a formal denotational semantics. An example of the use of the language follows the formal syntax and semantics.

The report concludes with a list of references and an alphabetic index.

CONTENTS

INTRODUCTION

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially goto's that pass arguments. Scheme was the first widely used programming language to embrace first class escape procedures, from which all previously known sequential control structures can be synthesized. A subsequent version of Scheme introduced the concept of exact and inexact numbers, an extension of Common Lisp's generic arithmetic. More recently, Scheme became the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended in a consistent and reliable manner.

Background

The first description of Scheme was written in 1975 [28]. A revised report [25] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [26]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [21, 17, 10]. An introductory computer science textbook using Scheme was published in 1984 [1].

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Their report [4] was published at MIT and Indiana University in the summer of 1985. Further revision took place in the spring of 1986 [23], and in the spring of 1988 [6]. The present report reflects further revisions agreed upon in a meeting at Xerox PARC in June 1992.

We intend this report to belong to the entire Scheme community, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementors of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.

Acknowledgements

We would like to thank the following people for their help: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Marc Feeley, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu, and Ozan Yigit. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual*[30]. We gladly acknowledge the influence of manuals for MIT Scheme[17], T[22], Scheme 84[11],Common Lisp[27], and Algol 60[18].

We also thank Betty Dexter for the extreme effort she put into setting this report in TEX, and Donald Knuth for designing the program that caused her troubles.

The Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, the Computer Science Department of Indiana University, the Computer and Information Sciences Department of the University of Oregon, and the NEC Research Institute supported the preparation of this report. Support for the MIT work was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Support for the Indiana University work was provided by NSF grants NCS 83-04567 and NCS 83-03325.

DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

1.1. Semantics

This section gives an overview of Scheme's semantics. A detailed informal semantics is the subject of chapters 3 through 6. For reference purposes, section 7.2 provides a formal semantics of Scheme.

Following Algol, Scheme is a statically scoped programming language. Each use of a variable is associated with a lexically apparent binding of that variable.

Scheme has latent as opposed to manifest types. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are APL, Snobol, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, Pascal, and C.

All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include APL and other Lisp dialects.

Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar. See section 3.5.

Scheme procedures are objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp and ML.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section 6.4.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. ML, C, and APL are three other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Scheme, every integer is a rational number, every rational is a real, and every real is a complex number. Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Scheme. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. As in Common Lisp, exact arithmetic is not limited to integers.

1.2. Syntax

Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. For example, the `eval` procedure evaluates a Scheme program expressed as data.

The `read` procedure performs syntactic as well as lexical decomposition of the data it reads. The `read` procedure parses its input as data (section 7.1.2), not as program.

The formal syntax of Scheme is described in section 7.1.

1.3. Notation and terminology

1.3.1. Primitive, library, and optional features

It is required that every implementation of Scheme support all features that are not marked as being *optional*. Implementations are free to omit optional features of Scheme or to add extensions, provided the extensions are not in conflict with the language reported here. In particular, implementations must support portable code by providing a syntactic mode that preempts no lexical conventions of this report.

To aid in understanding and implementing Scheme, some features are marked as *library*. These can be easily implemented in terms of the other, primitive, features. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

1.3.2. Error situations and unspecified behavior

When speaking of an error situation, this report uses the phrase "an error is signalled" to indicate that implementations must detect and report the error. If such wording does not appear in the discussion of an error, then implementations are not required to detect or report the error, though they are encouraged to do so. An error situation that implementations are not required to detect is usually referred to simply as "an error."

For example, it is an error for a procedure to be passed an argument that the procedure is not explicitly specified to handle, even though such domain errors are seldom mentioned in this report. Implementations may extend a procedure's domain of definition to include such arguments.

This report uses the phrase "may report a violation of an implementation restriction" to indicate circumstances under which an implementation is permitted to report that it is unable to continue execution of a correct program because of some restriction imposed by the implementation. Implementation restrictions are of course discouraged, but implementations are encouraged to report violations of implementation restrictions.

For example, an implementation may report a violation of an implementation restriction if it does not have enough storage to run a program.

If the value of an expression is said to be "unspecified," then the expression must evaluate to some object without signalling an error, but the value depends on the implementation; this report explicitly does not say what value should be returned.

1.3.3. Entry format

Chapters 4 and 6 are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

template *category*

for required, primitive features, or

template *qualifier category*

where *qualifier* is either "library" or "optional" as defined in section 1.3.1.

If *category* is "syntax", the entry describes an expression type, and the template gives the syntax of the expression type. Components of expressions are designated by syntactic variables, which are written using angle brackets, for example, ⟨expression⟩, ⟨variable⟩. Syntactic variables should be understood to denote segments of program text; for example, ⟨expression⟩ stands for any string of characters which is a syntactically valid expression. The notation

$\langle\text{thing}_1\rangle \ldots$

indicates zero or more occurrences of a ⟨thing⟩, and

$\langle\text{thing}_1\rangle\ \langle\text{thing}_2\rangle \ldots$

indicates one or more occurrences of a ⟨thing⟩.

If *category* is "procedure", then the entry describes a procedure, and the header line gives a template for a call to the procedure. Argument names in the template are *italicized*. Thus the header line

`(vector-ref` *vector* *k*`)` procedure

indicates that the built-in procedure `vector-ref` takes two arguments, a vector *vector* and an exact non-negative integer *k* (see below). The header lines

`(make-vector` *k*`)` procedure
`(make-vector` *k* *fill*`)` procedure

indicate that the `make-vector` procedure must be defined to take either one or two arguments.

It is an error for an operation to be presented with an argument that it is not specified to handle. For succinctness, we follow the convention that if an argument name is also the name of a type listed in section 3.2, then that argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions also imply type restrictions:

obj	any object
$list$, $list_1$, ... $list_j$, ...	list (see section 6.3.2)
z, z_1, ... z_j, ...	complex number
x, x_1, ... x_j, ...	real number
y, y_1, ... y_j, ...	real number
q, q_1, ... q_j, ...	rational number
n, n_1, ... n_j, ...	integer
k, k_1, ... k_j, ...	exact non-negative integer

1.3.4. Evaluation examples

The symbol "⟹" used in program examples should be read "evaluates to." For example,

```
(* 5 8)                ⟹  40
```

means that the expression `(* 5 8)` evaluates to the object `40`. Or, more precisely: the expression given by the sequence of characters "`(* 5 8)`" evaluates, in the initial environment, to an object that may be represented externally by the sequence of characters "`40`". See section 3.3 for a discussion of external representations of objects.

1.3.5. Naming conventions

By convention, the names of procedures that always return a boolean value usually end in "?". Such procedures are called predicates.

By convention, the names of procedures that store values into previously allocated locations (see section 3.4) usually end in "`!`". Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is unspecified.

By convention, "`->`" appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, `list->vector` takes a list and returns a vector whose elements are the same as those of the list.

2. Lexical conventions

This section gives an informal account of some of the lexical conventions used in writing Scheme programs. For a formal syntax of Scheme, see section 7.1.

Upper and lower case forms of a letter are never distinguished except within character and string constants. For example, `Foo` is the same identifier as `FOO`, and `#x1AB` is the same number as `#X1ab`.

2.1. Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. The precise rules for forming identifiers vary among implementations of Scheme, but in all implementations a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, `+`, `-`, and `...` are identifiers. Here are some examples of identifiers:

```
lambda                q
list->vector          soup
+                     V17a
<=?                   a34kTMNs
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

See section 7.1.1 for a formal syntax of identifiers.

Identifiers have two uses within Scheme programs:

- Any identifier may be used as a variable or as a syntactic keyword (see sections 3.1 and 4.3).

- When an identifier appears as a literal or within a literal (see section 4.1.2), it is being used to denote a *symbol* (see section 6.3.3).

2.2. Whitespace and comments

Whitespace characters are spaces and newlines. (Implementations typically provide additional whitespace characters such as tab or page break.) Whitespace is used for improved readability and as necessary to separate tokens from each other, a token being an indivisible lexical unit such as an identifier or number, but is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

A semicolon (`;`) indicates the start of a comment. The comment continues to the end of the line on which the semicolon appears. Comments are invisible to Scheme, but the end of the line is visible as whitespace. This prevents a comment from appearing in the middle of an identifier or number.

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
        1        ;Base case: return 1
        (* n (fact (- n 1))))))
```

2.3. Other notations

For a description of the notations used for numbers, see section 6.2.

`. + -` These are used in numbers, and may also occur anywhere in an identifier except as the first character. A delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (section 6.3.2), and to indicate a rest-parameter in a formal parameter list (section 4.1.4). A delimited sequence of three successive periods is also an identifier.

`( )` Parentheses are used for grouping and to notate lists (section 6.3.2).

`'` The single quote character is used to indicate literal data (section 4.1.2).

`` ` `` The backquote character is used to indicate almost-constant data (section 4.2.6).

`, ,@` The character comma and the sequence comma at-sign are used in conjunction with backquote (section 4.2.6).

`"` The double quote character is used to delimit strings (section 6.3.5).

`\` Backslash is used in the syntax for character constants (section 6.3.4) and as an escape character within string constants (section 6.3.5).

`[ ] { } |` Left and right square brackets and curly braces and vertical bar are reserved for possible future extensions to the language.

`#` Sharp sign is used for a variety of purposes depending on the character that immediately follows it:

`#t #f` These are the boolean constants (section 6.3.1).

`#\` This introduces a character constant (section 6.3.4).

`#(` This introduces a vector constant (section 6.3.6). Vector constants are terminated by `)` .

`#e #i #b #o #d #x` These are used in the notation for numbers (section 6.2.4).

3. Basic concepts

3.1. Variables, syntactic keywords, and regions

An identifier may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a type of syntax is called a *syntactic keyword* and is said to be *bound* to that syntax. An identifier that names a location is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new kinds of syntax and bind syntactic keywords to those new syntaxes, while other expression types create new locations and bind variables to those locations. These expression types are called *binding constructs.* Those that bind syntactic keywords are listed in section 4.3. The most fundamental of the variable binding constructs is the `lambda` expression, because all other variable binding constructs can be explained in terms of `lambda` expressions. The other variable binding constructs are `let`, `let*`, `letrec`, and `do` expressions (see sections 4.1.4, 4.2.2, and 4.2.4).

Like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp, Scheme is a statically scoped language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program text within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a `lambda` expression, for example, then its region is the entire `lambda` expression. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use. If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any (chapters 4 and 6); if there is no binding for the identifier, it is said to be *unbound.*

3.2. Disjointness of types

No object satisfies more than one of the following predicates:

```
boolean?          pair?
symbol?           number?
char?             string?
vector?           port?
procedure?
```

These predicates define the types *boolean*, *pair*, *symbol*, *number*, *char* (or *character*), *string*, *vector*, *port*, and *procedure.* The empty list is a special object of its own type; it satisfies none of the above predicates.

Although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section 6.3.1, all values count as true in such a test except for `#f`. This report uses the word "true" to refer to any Scheme value except `#f`, and the word "false" to refer to `#f`.

3.3. External representations

An important concept in Scheme (and Lisp) is that of the *external representation* of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters "28", and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters "`(8 13)`".

The external representation of an object is not necessarily unique. The integer 28 also has representations "`#e28.000`" and "`#x1c`", and the list in the previous paragraph also has the representations "`( 08 13 )`" and "`(8 . (13 . ()))`" (see section 6.3.2).

Many objects have standard external representations, but some, such as procedures, do not have standard representations (although particular implementations may define representations for them).

An external representation may be written in a program to obtain the corresponding object (see `quote`, section 4.1.2).

External representations can also be used for input and output. The procedure read (section 6.6.2) parses external representations, and the procedure write (section 6.6.3) generates them. Together, they provide an elegant and powerful input/output facility.

Note that the sequence of characters "(+ 2 6)" is *not* an external representation of the integer 8, even though it *is* an expression evaluating to the integer 8; rather, it is an external representation of a three-element list, the elements of which are the symbol + and the integers 2 and 6. Scheme's syntax has the property that any sequence of characters that is an expression is also the external representation of some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa).

The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects in the appropriate sections of chapter 6.

3.4. Storage model

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the string-set! procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as car, vector-ref, or string-ref, is equivalent in the sense of eqv? (section 6.1) to the object last stored in the location before the fetch.

Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use. Whenever this report speaks of storage being allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them.

In many systems it is desirable for constants (i.e. the values of literal expressions) to reside in read-only-memory. To express this, it is convenient to imagine that every object that denotes locations is associated with a flag telling whether that object is mutable or immutable. In such systems literal constants and the strings returned by symbol->string are immutable objects, while all objects created by the other procedures listed in this report are mutable. It is an error to attempt to store a new value into a location that is denoted by an immutable object.

3.5. Proper tail recursion

Implementations of Scheme are required to be *properly tail-recursive*. Procedure calls that occur in certain syntactic contexts defined below are 'tail calls'. A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls. A call is *active* if the called procedure may still return. Note that this includes calls that may be returned from either by the current continuation or by continuations captured earlier by call-with-current-continuation that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned. A formal definition of proper tail recursion can be found in [8].

Rationale:

Intuitively, no space is needed for an active tail call because the continuation that is used in the tail call has the same semantics as the continuation passed to the procedure containing the call. Although an improper implementation might use a new continuation in the call, a return to this new continuation would be followed immediately by a return to the continuation passed to the procedure. A properly tail-recursive implementation returns to that continuation directly.

Proper tail recursion was one of the central ideas in Steele and Sussman's original version of Scheme. Their first Scheme interpreter implemented both functions and actors. Control flow was expressed using actors, which differed from functions in that they passed their results on to another actor instead of returning to a caller. In the terminology of this section, each actor finished with a tail call to another actor.

Steele and Sussman later observed that in their interpreter the code for dealing with actors was identical to that for functions and thus there was no need to include both in the language.

A *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined inductively. Note that a tail context is always determined with respect to a particular lambda expression.

- The last expression within the body of a lambda expression, shown as ⟨tail expression⟩ below, occurs in a tail context.

```
(lambda ⟨formals⟩
  ⟨definition⟩* ⟨expression⟩* ⟨tail expression⟩)
```

- If one of the following expressions is in a tail context, then the subexpressions shown as ⟨tail expression⟩ are in a tail context. These were derived from rules in the grammar given in chapter 7 by replacing some occurrences of ⟨expression⟩ with ⟨tail expression⟩. Only those rules that contain tail contexts are shown here.

```
(if ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩)
(if ⟨expression⟩ ⟨tail expression⟩)

(cond ⟨cond clause⟩+)
(cond ⟨cond clause⟩* (else ⟨tail sequence⟩))

(case ⟨expression⟩
  ⟨case clause⟩+)
(case ⟨expression⟩
  ⟨case clause⟩*
  (else ⟨tail sequence⟩))

(and ⟨expression⟩* ⟨tail expression⟩)
(or ⟨expression⟩* ⟨tail expression⟩)

(let (⟨binding spec⟩*) ⟨tail body⟩)
(let ⟨variable⟩ (⟨binding spec⟩*) ⟨tail body⟩)
(let* (⟨binding spec⟩*) ⟨tail body⟩)
(letrec (⟨binding spec⟩*) ⟨tail body⟩)

(let-syntax (⟨syntax spec⟩*) ⟨tail body⟩)
(letrec-syntax (⟨syntax spec⟩*) ⟨tail body⟩)

(begin ⟨tail sequence⟩)

(do (⟨iteration spec⟩*)
    (⟨test⟩ ⟨tail sequence⟩)
  ⟨expression⟩*)
```

where

```
⟨cond clause⟩ ⟶ (⟨test⟩ ⟨tail sequence⟩)
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨tail sequence⟩)

⟨tail body⟩ ⟶ ⟨definition⟩* ⟨tail sequence⟩
⟨tail sequence⟩ ⟶ ⟨expression⟩* ⟨tail expression⟩
```

- If a `cond` expression is in a tail context, and has a clause of the form (⟨expression$_1$⟩ `=>` ⟨expression$_2$⟩) then the (implied) call to the procedure that results from the evaluation of ⟨expression$_2$⟩ is in a tail context. ⟨expression$_2$⟩ itself is not in a tail context.

Certain built-in procedures are also required to perform tail calls. The first argument passed to `apply` and to `call-with-current-continuation`, and the second argument passed to `call-with-values`, must be called via a tail call. Similarly, `eval` must evaluate its argument as if it were in tail position within the `eval` procedure.

In the following example the only tail call is the call to `f`. None of the calls to `g` or `h` are tail calls. The reference to `x` is in a tail context, but it is not a call and thus is not a tail call.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

Note: Implementations are allowed, but not required, to recognize that some non-tail calls, such as the call to `h` above, can be evaluated as though they were tail calls. In the example above, the `let` expression could be compiled as a tail call to `h`. (The possibility of `h` returning an unexpected number of values can be ignored, because in that case the effect of the `let` is explicitly unspecified and implementation-dependent.)

4. Expressions

Expression types are categorized as *primitive* or *derived.* Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive, but can instead be defined as macros. With the exception of `quasiquote`, whose macro definition is complex, the derived expressions are classified as library features. Suitable definitions are given in section 7.3.

4.1. Primitive expression types

4.1.1. Variable references

⟨variable⟩ syntax

An expression consisting of a variable (section 3.1) is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

```
(define x 28)
x                           ⟹  28
```

4.1.2. Literal expressions

(`quote` ⟨datum⟩) syntax
`'`⟨datum⟩ syntax
⟨constant⟩ syntax

(`quote` ⟨datum⟩) evaluates to ⟨datum⟩. ⟨Datum⟩ may be any external representation of a Scheme object (see section 3.3). This notation is used to include literal constants in Scheme code.

```
(quote a)                   ⟹  a
(quote #(a b c))            ⟹  #(a b c)
(quote (+ 1 2))             ⟹  (+ 1 2)
```

`(quote` ⟨datum⟩`)` may be abbreviated as `'`⟨datum⟩. The two notations are equivalent in all respects.

```
'a                          ⟹  a
'#(a b c)                   ⟹  #(a b c)
'()                         ⟹  ()
'(+ 1 2)                    ⟹  (+ 1 2)
'(quote a)                  ⟹  (quote a)
''a                         ⟹  (quote a)
```

Numerical constants, string constants, character constants, and boolean constants evaluate "to themselves"; they need not be quoted.

```
'"abc"                      ⟹  "abc"
"abc"                       ⟹  "abc"
'145932                     ⟹  145932
145932                      ⟹  145932
'#t                         ⟹  #t
#t                          ⟹  #t
```

As noted in section 3.4, it is an error to alter a constant (i.e. the value of a literal expression) using a mutation procedure like `set-car!` or `string-set!`.

4.1.3. Procedure calls

`(`⟨operator⟩ ⟨operand$_1$⟩ ...`)`   syntax

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments.

```
(+ 3 4)                     ⟹  7
((if #f + *) 3 4)           ⟹  12
```

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables `+` and `*`. New procedures are created by evaluating `lambda` expressions (see section 4.1.4).

Procedure calls may return any number of values (see `values` in section 6.4). With the exception of `values` the procedures available in the initial environment return one value or, for procedures such as `apply`, pass on the values returned by a call to one of their arguments.

Procedure calls are also called *combinations*.

Note: In contrast to other dialects of Lisp, the order of evaluation is unspecified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

Note: Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.

Note: In many dialects of Lisp, the empty combination, `()`, is a legitimate expression. In Scheme, combinations must have at least one subexpression, so `()` is not a syntactically valid expression.

4.1.4. Procedures

`(lambda` ⟨formals⟩ ⟨body⟩`)`   syntax

Syntax: ⟨Formals⟩ should be a formal arguments list as described below, and ⟨body⟩ should be a sequence of one or more expressions.

Semantics: A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the `lambda` expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and the expressions in the body of the `lambda` expression will be evaluated sequentially in the extended environment. The result(s) of the last expression in the body will be returned as the result(s) of the procedure call.

```
(lambda (x) (+ x x))        ⟹  a procedure
((lambda (x) (+ x x)) 4)    ⟹  8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)     ⟹  3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                    ⟹  10
```

⟨Formals⟩ should have one of the following forms:

- `(`⟨variable$_1$⟩ ...`)`: The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.
- ⟨variable⟩: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the ⟨variable⟩.
- `(`⟨variable$_1$⟩ ... ⟨variable$_n$⟩ `.` ⟨variable$_{n+1}$⟩`)`: If a space-delimited period precedes the last variable, then the procedure takes n or more arguments, where n is the number of formal arguments before the period (there must be at least one). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

It is an error for a ⟨variable⟩ to appear more than once in ⟨formals⟩.

```
((lambda x x) 3 4 5 6)          ⟹  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                       ⟹  (5 6)
```

Each procedure created as the result of evaluating a `lambda` expression is (conceptually) tagged with a storage location, in order to make `eqv?` and `eq?` work on procedures (see section 6.1).

4.1.5. Conditionals

`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)` syntax
`(if` ⟨test⟩ ⟨consequent⟩`)` syntax

Syntax: ⟨Test⟩, ⟨consequent⟩, and ⟨alternate⟩ may be arbitrary expressions.

Semantics: An `if` expression is evaluated as follows: first, ⟨test⟩ is evaluated. If it yields a true value (see section 6.3.1), then ⟨consequent⟩ is evaluated and its value(s) is(are) returned. Otherwise ⟨alternate⟩ is evaluated and its value(s) is(are) returned. If ⟨test⟩ yields a false value and no ⟨alternate⟩ is specified, then the result of the expression is unspecified.

```
(if (> 3 2) 'yes 'no)           ⟹  yes
(if (> 2 3) 'yes 'no)           ⟹  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                    ⟹  1
```

4.1.6. Assignments

`(set!` ⟨variable⟩ ⟨expression⟩`)` syntax

⟨Expression⟩ is evaluated, and the resulting value is stored in the location to which ⟨variable⟩ is bound. ⟨Variable⟩ must be bound either in some region enclosing the `set!` expression or at top level. The result of the `set!` expression is unspecified.

```
(define x 2)
(+ x 1)                         ⟹  3
(set! x 4)                      ⟹  unspecified
(+ x 1)                         ⟹  5
```

4.2. Derived expression types

The constructs in this section are hygienic, as discussed in section 4.3. For reference purposes, section 7.3 gives macro definitions that will convert most of the constructs described in this section into the primitive constructs described in the previous section.

4.2.1. Conditionals

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ `...)` library syntax

Syntax: Each ⟨clause⟩ should be of the form

`(`⟨test⟩ ⟨expression$_1$⟩ `...)`

where ⟨test⟩ is any expression. Alternatively, a ⟨clause⟩ may be of the form

`(`⟨test⟩ `=>` ⟨expression⟩`)`

The last ⟨clause⟩ may be an "else clause," which has the form

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ `...)`.

Semantics: A `cond` expression is evaluated by evaluating the ⟨test⟩ expressions of successive ⟨clause⟩s in order until one of them evaluates to a true value (see section 6.3.1). When a ⟨test⟩ evaluates to a true value, then the remaining ⟨expression⟩s in its ⟨clause⟩ are evaluated in order, and the result(s) of the last ⟨expression⟩ in the ⟨clause⟩ is(are) returned as the result(s) of the entire `cond` expression. If the selected ⟨clause⟩ contains only the ⟨test⟩ and no ⟨expression⟩s, then the value of the ⟨test⟩ is returned as the result. If the selected ⟨clause⟩ uses the `=>` alternate form, then the ⟨expression⟩ is evaluated. Its value must be a procedure that accepts one argument; this procedure is then called on the value of the ⟨test⟩ and the value(s) returned by this procedure is(are) returned by the `cond` expression. If all ⟨test⟩s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its ⟨expression⟩s are evaluated, and the value(s) of the last one is(are) returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))          ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))            ⟹  equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))                ⟹  2
```

`(case` ⟨key⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ `...)` library syntax

Syntax: ⟨Key⟩ may be any expression. Each ⟨clause⟩ should have the form

`((`⟨datum$_1$⟩ `...)` ⟨expression$_1$⟩ ⟨expression$_2$⟩ `...)`,

where each ⟨datum⟩ is an external representation of some object. All the ⟨datum⟩s must be distinct. The last ⟨clause⟩ may be an "else clause," which has the form

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ `...)`.

Semantics: A `case` expression is evaluated as follows. ⟨Key⟩ is evaluated and its result is compared against each ⟨datum⟩. If the result of evaluating ⟨key⟩ is equivalent (in the sense of `eqv?`; see section 6.1) to a ⟨datum⟩, then the expressions in the corresponding ⟨clause⟩ are evaluated from left to right and the result(s) of the last expression in the ⟨clause⟩ is(are) returned as the result(s) of the `case` expression. If the result of evaluating ⟨key⟩ is different from every ⟨datum⟩, then if there is an else clause its expressions are evaluated and the result(s) of the last is(are) the result(s) of the `case` expression; otherwise the result of the `case` expression is unspecified.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⟹  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                 ⟹  unspecified
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant))        ⟹  consonant
```

`(and` ⟨test$_1$⟩ `...)` library syntax

The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value (see section 6.3.1) is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then `#t` is returned.

```
(and (= 2 2) (> 2 1))       ⟹  #t
(and (= 2 2) (< 2 1))       ⟹  #f
(and 1 2 'c '(f g))         ⟹  (f g)
(and)                       ⟹  #t
```

`(or` ⟨test$_1$⟩ `...)` library syntax

The ⟨test⟩ expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see section 6.3.1) is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then `#f` is returned.

```
(or (= 2 2) (> 2 1))        ⟹  #t
(or (= 2 2) (< 2 1))        ⟹  #t
(or #f #f #f)               ⟹  #f
(or (memq 'b '(a b c))
    (/ 3 0))                ⟹  (b c)
```

4.2.2. Binding constructs

The three binding constructs `let`, `let*`, and `letrec` give Scheme a block structure, like Algol 60. The syntax of the three constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially; while in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

`(let` ⟨bindings⟩ ⟨body⟩`)` library syntax

Syntax: ⟨Bindings⟩ should have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

where each ⟨init⟩ is an expression, and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨init⟩s are evaluated in the current environment (in some unspecified order), the ⟨variable⟩s are bound to fresh locations holding the results, the ⟨body⟩ is evaluated in the extended environment, and the value(s) of the last expression of ⟨body⟩ is(are) returned. Each binding of a ⟨variable⟩ has ⟨body⟩ as its region.

```
(let ((x 2) (y 3))
  (* x y))                  ⟹  6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))               ⟹  35
```

See also named `let`, section 4.2.4.

`(let*` ⟨bindings⟩ ⟨body⟩`)` library syntax

Syntax: ⟨Bindings⟩ should have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

and ⟨body⟩ should be a sequence of one or more expressions.

Semantics: `Let*` is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a binding indicated by `(`⟨variable⟩ ⟨init⟩`)` is that part of the `let*` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))               ⟹  70
```

`(letrec` ⟨bindings⟩ ⟨body⟩`)` library syntax

Syntax: ⟨Bindings⟩ should have the form

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨variable⟩s are bound to fresh locations holding undefined values, the ⟨init⟩s are evaluated in the resulting environment (in some unspecified order), each ⟨variable⟩ is assigned to the result of the corresponding ⟨init⟩, the ⟨body⟩ is evaluated in the resulting environment, and the value(s) of the last expression in ⟨body⟩ is(are) returned. Each binding of a ⟨variable⟩ has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                            ⟹    #t
```

One restriction on letrec is very important: it must be possible to evaluate each ⟨init⟩ without assigning or referring to the value of any ⟨variable⟩. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of letrec, all the ⟨init⟩s are lambda expressions and the restriction is satisfied automatically.

4.2.3. Sequencing

(begin ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...)   library syntax

The ⟨expression⟩s are evaluated sequentially from left to right, and the value(s) of the last ⟨expression⟩ is(are) returned. This expression type is used to sequence side effects such as input and output.

```
(define x 0)

(begin (set! x 5)
       (+ x 1))             ⟹    6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))   ⟹    unspecified
           and prints  4 plus 1 equals 5
```

4.2.4. Iteration

(do ((⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩)   library syntax
     ...)
    (⟨test⟩ ⟨expression⟩ ...)
  ⟨command⟩ ...)

Do is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits after evaluating the ⟨expression⟩s.

Do expressions are evaluated as follows: The ⟨init⟩ expressions are evaluated (in some unspecified order), the ⟨variable⟩s are bound to fresh locations, the results of the ⟨init⟩ expressions are stored in the bindings of the ⟨variable⟩s, and then the iteration phase begins.

Each iteration begins by evaluating ⟨test⟩; if the result is false (see section 6.3.1), then the ⟨command⟩ expressions are evaluated in order for effect, the ⟨step⟩ expressions are evaluated in some unspecified order, the ⟨variable⟩s are bound to fresh locations, the results of the ⟨step⟩s are stored in the bindings of the ⟨variable⟩s, and the next iteration begins.

If ⟨test⟩ evaluates to a true value, then the ⟨expression⟩s are evaluated from left to right and the value(s) of the last ⟨expression⟩ is(are) returned. If no ⟨expression⟩s are present, then the value of the do expression is unspecified.

The region of the binding of a ⟨variable⟩ consists of the entire do expression except for the ⟨init⟩s. It is an error for a ⟨variable⟩ to appear more than once in the list of do variables.

A ⟨step⟩ may be omitted, in which case the effect is the same as if (⟨variable⟩ ⟨init⟩ ⟨variable⟩) had been written instead of (⟨variable⟩ ⟨init⟩).

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))    ⟹    #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))     ⟹    25
```

(let ⟨variable⟩ ⟨bindings⟩ ⟨body⟩)   library syntax

"Named let" is a variant on the syntax of let which provides a more general looping construct than do and may also be used to express recursions. It has the same syntax and semantics as ordinary let except that ⟨variable⟩ is bound within ⟨body⟩ to a procedure whose formal arguments are the bound variables and whose body is ⟨body⟩. Thus the execution of ⟨body⟩ may be repeated by invoking the procedure named by ⟨variable⟩.

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
          (loop (cdr numbers)
                nonneg
                (cons (car numbers) neg)))))
   ⟹  ((6 1 3) (-5 -2))
```

4.2.5. Delayed evaluation

(delay ⟨expression⟩) library syntax

The delay construct is used together with the procedure force to implement *lazy evaluation* or *call by need*. (delay ⟨expression⟩) returns an object called a *promise* which at some point in the future may be asked (by the force procedure) to evaluate ⟨expression⟩, and deliver the resulting value. The effect of ⟨expression⟩ returning multiple values is unspecified.

See the description of force (section 6.4) for a more complete description of delay.

4.2.6. Quasiquotation

(quasiquote ⟨qq template⟩) syntax
`⟨qq template⟩ syntax

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the ⟨qq template⟩, the result of evaluating `⟨qq template⟩ is equivalent to the result of evaluating '⟨qq template⟩. If a comma appears within the ⟨qq template⟩, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence. A comma at-sign should only appear within a list or vector ⟨qq template⟩.

```
`(list ,(+ 1 2) 4)          ⟹  (list 3 4)
(let ((name 'a)) `(list ,name ',name))
          ⟹  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
          ⟹  (a 3 4 5 6 b)
`(( foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
          ⟹  ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
          ⟹  #(10 5 2 4 3 8)
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ⟹  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
          ⟹  (a `(b ,x ,'y d) e)
```

The two notations `⟨qq template⟩ and (quasiquote ⟨qq template⟩) are identical in all respects. ,⟨expression⟩ is identical to (unquote ⟨expression⟩), and ,@⟨expression⟩ is identical to (unquote-splicing ⟨expression⟩). The external syntax generated by write for two-element lists whose car is one of these symbols may vary between implementations.

```
(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  `(list ,(+ 1 2) 4)
      i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

Unpredictable behavior can result if any of the symbols quasiquote, unquote, or unquote-splicing appear in positions within a ⟨qq template⟩ otherwise than as described above.

4.3. Macros

Scheme programs can define and use new derived expression types, called *macros*. Program-defined expression types have the syntax

(⟨keyword⟩ ⟨datum⟩ ...)

where ⟨keyword⟩ is an identifier that uniquely determines the expression type. This identifier is called the *syntactic keyword*, or simply *keyword*, of the macro. The number of the ⟨datum⟩s, and their syntax, depends on the expression type.

Each instance of a macro is called a *use* of the macro. The set of rules that specifies how a use of a macro is transcribed into a more primitive expression is called the *transformer* of the macro.

The macro definition facility consists of two parts:

- A set of expressions used to establish that certain identifiers are macro keywords, associate them with macro transformers, and control the scope within which a macro is defined, and
- a pattern language for specifying macro transformers.

The syntactic keyword of a macro may shadow variable bindings, and local variable bindings may shadow keyword bindings. All macros defined using the pattern language are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping [14, 15, 2, 7, 9]:

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers. Note that a `define` at top level may or may not introduce a binding; see section 5.2.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

4.3.1. Binding constructs for syntactic keywords

`Let-syntax` and `letrec-syntax` are analogous to `let` and `letrec`, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Syntactic keywords may also be bound at top level; see section 5.3.

`(let-syntax` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: ⟨Bindings⟩ should have the form

```
((⟨keyword⟩ ⟨transformer spec⟩) ...)
```

Each ⟨keyword⟩ is an identifier, each ⟨transformer spec⟩ is an instance of `syntax-rules`, and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨keyword⟩ to appear more than once in the list of keywords being bound.

Semantics: The ⟨body⟩ is expanded in the syntactic environment obtained by extending the syntactic environment of the `let-syntax` expression with macros whose keywords are the ⟨keyword⟩s, bound to the specified transformers. Each binding of a ⟨keyword⟩ has ⟨body⟩ as its region.

```
(let-syntax ((when (syntax-rules ()
                     ((when test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (when if (set! if ’now))
    if))                       ⟹  now

(let ((x ’outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x ’inner))
      (m))))                   ⟹  outer
```

`(letrec-syntax` ⟨bindings⟩ ⟨body⟩`)` syntax

Syntax: Same as for `let-syntax`.

Semantics: The ⟨body⟩ is expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` expression with macros whose keywords are the ⟨keyword⟩s, bound to the specified transformers. Each binding of a ⟨keyword⟩ has the ⟨bindings⟩ as well as the ⟨body⟩ within its region, so the transformers can transcribe expressions into uses of the macros introduced by the `letrec-syntax` expression.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))             ⟹  7
```

4.3.2. Pattern language

A ⟨transformer spec⟩ has the following form:

`(syntax-rules` ⟨literals⟩ ⟨syntax rule⟩ `...)`

Syntax: ⟨Literals⟩ is a list of identifiers and each ⟨syntax rule⟩ should be of the form

```
(⟨pattern⟩ ⟨template⟩)
```

The ⟨pattern⟩ in a ⟨syntax rule⟩ is a list ⟨pattern⟩ that begins with the keyword for the macro.

A ⟨pattern⟩ is either an identifier, a constant, or one of the following

```
(⟨pattern⟩ ...)
(⟨pattern⟩ ⟨pattern⟩ ... . ⟨pattern⟩)
(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩)
#(⟨pattern⟩ ...)
#(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩)
```

and a template is either an identifier, a constant, or one of the following

```
(⟨element⟩ ...)
(⟨element⟩ ⟨element⟩ ... . ⟨template⟩)
#(⟨element⟩ ...)
```

where an ⟨element⟩ is a ⟨template⟩ optionally followed by an ⟨ellipsis⟩ and an ⟨ellipsis⟩ is the identifier "`...`" (which cannot be used as an identifier in either a template or a pattern).

Semantics: An instance of `syntax-rules` produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the ⟨syntax rule⟩s, beginning with the leftmost ⟨syntax rule⟩. When a match is found, the macro use is transcribed hygienically according to the template.

An identifier that appears in the pattern of a ⟨syntax rule⟩ is a *pattern variable*, unless it is the keyword that begins the pattern, is listed in ⟨literals⟩, or is the identifier "`...`". Pattern variables match arbitrary input elements and are used to refer to elements of the input in the template. It is an error for the same pattern variable to appear more than once in a ⟨pattern⟩.

The keyword at the beginning of the pattern in a ⟨syntax rule⟩ is not involved in the matching and is not considered a pattern variable or literal identifier.

Rationale: The scope of the keyword is determined by the expression or syntax definition that binds it to the associated macro transformer. If the keyword were a pattern variable or literal identifier, then the template that follows the pattern would be within its scope regardless of whether the keyword were bound by `let-syntax` or by `letrec-syntax`.

Identifiers that appear in ⟨literals⟩ are interpreted as literal identifiers to be matched against corresponding subforms of the input. A subform in the input matches a literal identifier if and only if it is an identifier and either both its occurrence in the macro expression and its occurrence in the macro definition have the same lexical binding, or the two identifiers are equal and both have no lexical binding.

A subpattern followed by `...` can match zero or more elements of the input. It is an error for `...` to appear in ⟨literals⟩. Within a pattern the identifier `...` must follow the last element of a nonempty sequence of subpatterns.

More formally, an input form F matches a pattern P if and only if:

- P is a non-literal identifier; or
- P is a literal identifier and F is an identifier with the same binding; or
- P is a list `(`P_1 … P_n`)` and F is a list of n forms that match P_1 through P_n, respectively; or
- P is an improper list `(`P_1 P_2 … P_n `.` P_{n+1}`)` and F is a list or improper list of n or more forms that match P_1 through P_n, respectively, and whose nth "cdr" matches P_{n+1}; or
- P is of the form `(`P_1 … P_n P_{n+1} ⟨ellipsis⟩`)` where ⟨ellipsis⟩ is the identifier `...` and F is a proper list of at least n forms, the first n of which match P_1 through P_n, respectively, and each remaining element of F matches P_{n+1}; or
- P is a vector of the form `#(`P_1 … P_n`)` and F is a vector of n forms that match P_1 through P_n; or
- P is of the form `#(`P_1 … P_n P_{n+1} ⟨ellipsis⟩`)` where ⟨ellipsis⟩ is the identifier `...` and F is a vector of n or more forms the first n of which match P_1 through P_n, respectively, and each remaining element of F matches P_{n+1}; or
- P is a datum and F is equal to P in the sense of the `equal?` procedure.

It is an error to use a macro keyword, within the scope of its binding, in an expression that does not match any of the patterns.

When a macro use is transcribed according to the template of the matching ⟨syntax rule⟩, pattern variables that occur in the template are replaced by the subforms they match in the input. Pattern variables that occur in subpatterns followed by one or more instances of the identifier `...` are allowed only in subtemplates that are followed by as many instances of `...`. They are replaced in the output by all of the subforms they match in the input, distributed as indicated. It is an error if the output cannot be built up as specified.

Identifiers that appear in the template but are not pattern variables or the identifier `...` are inserted into the output as literal identifiers. If a literal identifier is inserted as a free identifier then it refers to the binding of that identifier within whose scope the instance of `syntax-rules` appears. If a literal identifier is inserted as a bound identifier then it is in effect renamed to prevent inadvertent captures of free identifiers.

As an example, if `let` and `cond` are defined as in section 7.3 then they are hygienic (as required) and the following is not an error.

```
(let ((=> #f))
  (cond (#t => 'ok)))          ⟹ ok
```

The macro transformer for `cond` recognizes `=>` as a local variable, and hence an expression, and not as the top-level identifier `=>`, which the macro transformer treats as a syntactic keyword. Thus the example expands into

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

instead of

```
(let ((=> #f))
  (let ((temp #t))
    (if temp ('ok temp))))
```

which would result in an invalid procedure call.

5. Program structure

5.1. Programs

A Scheme program consists of a sequence of expressions, definitions, and syntax definitions. Expressions are described in chapter 4; definitions and syntax definitions are the subject of the rest of the present chapter.

Programs are typically stored in files or entered interactively to a running Scheme system, although other paradigms are possible; questions of user interface lie outside the scope of this report. (Indeed, Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.)

Definitions and syntax definitions occurring at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top level environment or modify the value of existing top-level bindings. Expressions occurring at the top level of a program are interpreted imperatively; they are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

At the top level of a program `(begin` ⟨form$_1$⟩ ...`)` is equivalent to the sequence of expressions, definitions, and syntax definitions that form the body of the `begin`.

5.2. Definitions

Definitions are valid in some, but not all, contexts where expressions are allowed. They are valid only at the top level of a ⟨program⟩ and at the beginning of a ⟨body⟩.

A definition should have one of the following forms:

- `(define` ⟨variable⟩ ⟨expression⟩`)`
- `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`

  ⟨Formals⟩ should be either a sequence of zero or more variables, or a sequence of one or more variables followed by a space-delimited period and another variable (as in a lambda expression). This form is equivalent to

  ```
  (define ⟨variable⟩
    (lambda (⟨formals⟩) ⟨body⟩)).
  ```

- `(define (`⟨variable⟩ `.` ⟨formal⟩`)` ⟨body⟩`)`

  ⟨Formal⟩ should be a single variable. This form is equivalent to

  ```
  (define ⟨variable⟩
    (lambda ⟨formal⟩ ⟨body⟩)).
  ```

5.2.1. Top level definitions

At the top level of a program, a definition

```
(define ⟨variable⟩ ⟨expression⟩)
```

has essentially the same effect as the assignment expression

```
(set! ⟨variable⟩ ⟨expression⟩)
```

if ⟨variable⟩ is bound. If ⟨variable⟩ is not bound, however, then the definition will bind ⟨variable⟩ to a new location before performing the assignment, whereas it would be an error to perform a `set!` on an unbound variable.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                    ⟹  6
(define first car)
(first '(1 2))              ⟹  1
```

Some implementations of Scheme use an initial environment in which all possible variables are bound to locations, most of which contain undefined values. Top level definitions in such an implementation are truly equivalent to assignments.

5.2.2. Internal definitions

Definitions may occur at the beginning of a ⟨body⟩ (that is, the body of a `lambda`, `let`, `let*`, `letrec`, `let-syntax`, or `letrec-syntax` expression or that of a definition of an appropriate form). Such definitions are known as *internal definitions* as opposed to the top level definitions described above. The variable defined by an internal definition is local to the ⟨body⟩. That is, ⟨variable⟩ is bound rather than assigned, and the region of the binding is the entire ⟨body⟩. For example,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))            ⟹  45
```

A ⟨body⟩ containing internal definitions can always be converted into a completely equivalent `letrec` expression. For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
           (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

Just as for the equivalent `letrec` expression, it must be possible to evaluate each ⟨expression⟩ of every internal definition in a ⟨body⟩ without assigning or referring to the value of any ⟨variable⟩ being defined.

Wherever an internal definition may occur `(begin` ⟨definition$_1$⟩ ...`)` is equivalent to the sequence of definitions that form the body of the `begin`.

5.3. Syntax definitions

Syntax definitions are valid only at the top level of a ⟨program⟩. They have the following form:

(define-syntax ⟨keyword⟩ ⟨transformer spec⟩)

⟨Keyword⟩ is an identifier, and the ⟨transformer spec⟩ should be an instance of syntax-rules. The top-level syntactic environment is extended by binding the ⟨keyword⟩ to the specified transformer.

There is no define-syntax analogue of internal definitions.

Although macros may expand into definitions and syntax definitions in any context that permits them, it is an error for a definition or syntax definition to shadow a syntactic keyword whose meaning is needed to determine whether some form in the group of forms that contains the shadowing definition is in fact a definition, or, for internal definitions, is needed to determine the boundary between the group and the expressions that follow the group. For example, the following are errors:

```
(define define 3)

(begin (define begin list))

(let-syntax
  ((foo (syntax-rules ()
          ((foo (proc args ...) body ...)
           (define proc
             (lambda (args ...)
               body ...))))))
  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x)))
```

6. Standard procedures

This chapter describes Scheme's built-in procedures. The initial (or "top level") Scheme environment starts out with a number of variables bound to locations containing useful values, most of which are primitive procedures that manipulate data. For example, the variable abs is bound to (a location initially containing) a procedure of one argument that computes the absolute value of a number, and the variable + is bound to a procedure that computes sums. Built-in procedures that can easily be written in terms of other built-in procedures are identified as "library procedures".

A program may use a top-level definition to bind any variable. It may subsequently alter any such binding by an assignment (see 4.1.6). These operations do not modify the behavior of Scheme's built-in procedures. Altering any top-level binding that has not been introduced by a definition has an unspecified effect on the behavior of the built-in procedures.

6.1. Equivalence predicates

A *predicate* is a procedure that always returns a boolean value (#t or #f). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, eq? is the finest or most discriminating, and equal? is the coarsest. Eqv? is slightly less discriminating than eq?.

(eqv? obj_1 obj_2) procedure

The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj_1 and obj_2 should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of eqv? holds for all implementations of Scheme.

The eqv? procedure returns #t if:

- obj_1 and obj_2 are both #t or both #f.
- obj_1 and obj_2 are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                              ⟹  #t
```

> *Note:* This assumes that neither obj_1 nor obj_2 is an "uninterned symbol" as alluded to in section 6.3.3. This report does not presume to specify the behavior of eqv? on implementation-dependent extensions.

- obj_1 and obj_2 are both numbers, are numerically equal (see =, section 6.2), and are either both exact or both inexact.
- obj_1 and obj_2 are both characters and are the same character according to the char=? procedure (section 6.3.4).
- both obj_1 and obj_2 are the empty list.
- obj_1 and obj_2 are pairs, vectors, or strings that denote the same locations in the store (section 3.4).
- obj_1 and obj_2 are procedures whose location tags are equal (section 4.1.4).

The eqv? procedure returns #f if:

- obj_1 and obj_2 are of different types (section 3.2).

- one of obj_1 and obj_2 is #t but the other is #f.
- obj_1 and obj_2 are symbols but

```
    (string=? (symbol->string obj1)
              (symbol->string obj2))
                                  ⟹  #f
```

- one of obj_1 and obj_2 is an exact number but the other is an inexact number.
- obj_1 and obj_2 are numbers for which the = procedure returns #f.
- obj_1 and obj_2 are characters for which the char=? procedure returns #f.
- one of obj_1 and obj_2 is the empty list but the other is not.
- obj_1 and obj_2 are pairs, vectors, or strings that denote distinct locations.
- obj_1 and obj_2 are procedures that would behave differently (return different value(s) or have different side effects) for some arguments.

```
(eqv? 'a 'a)                  ⟹  #t
(eqv? 'a 'b)                  ⟹  #f
(eqv? 2 2)                    ⟹  #t
(eqv? '() '())                ⟹  #t
(eqv? 100000000 100000000)    ⟹  #t
(eqv? (cons 1 2) (cons 1 2))⟹  #f
(eqv? (lambda () 1)
      (lambda () 2))          ⟹  #f
(eqv? #f 'nil)                ⟹  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                 ⟹  #t
```

The following examples illustrate cases in which the above rules do not fully specify the behavior of eqv?. All that can be said about such cases is that the value returned by eqv? must be a boolean.

```
(eqv? "" "")                  ⟹  unspecified
(eqv? '#() '#())              ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))         ⟹  unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))         ⟹  unspecified
```

The next set of examples shows the use of eqv? with procedures that have local state. Gen-counter must return a distinct procedure every time, since each procedure has its own internal counter. Gen-loser, however, returns equivalent procedures each time, since the local state does not affect the value or side effects of the procedures.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                 ⟹  #t
(eqv? (gen-counter) (gen-counter))
                              ⟹  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                 ⟹  #t
(eqv? (gen-loser) (gen-loser))
                              ⟹  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                              ⟹  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
                              ⟹  #f
```

Since it is an error to modify constant objects (those returned by literal expressions), implementations are permitted, though not required, to share structure between constants where appropriate. Thus the value of eqv? on constants is sometimes implementation-dependent.

```
(eqv? '(a) '(a))              ⟹  unspecified
(eqv? "a" "a")                ⟹  unspecified
(eqv? '(b) (cdr '(a b)))      ⟹  unspecified
(let ((x '(a)))
  (eqv? x x))                 ⟹  #t
```

Rationale: The above definition of eqv? allows implementations latitude in their treatment of procedures and literals: implementations are free either to detect or to fail to detect that two procedures or two literals are equivalent to each other, and can decide whether or not to merge representations of equivalent objects by using the same pointer or bit pattern to represent both.

(eq? obj_1 obj_2) procedure

Eq? is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?.

Eq? and eqv? are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, and non-empty strings and vectors. Eq?'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when eqv? would also return true. Eq? may also behave differently from eqv? on empty vectors and empty strings.

```
(eq? 'a 'a)                     ⟹  #t
(eq? '(a) '(a))                 ⟹  unspecified
(eq? (list 'a) (list 'a))       ⟹  #f
(eq? "a" "a")                   ⟹  unspecified
(eq? "" "")                     ⟹  unspecified
(eq? '() '())                   ⟹  #t
(eq? 2 2)                       ⟹  unspecified
(eq? #\A #\A)                   ⟹  unspecified
(eq? car car)                   ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                    ⟹  unspecified
(let ((x '(a)))
  (eq? x x))                    ⟹  #t
(let ((x '#()))
  (eq? x x))                    ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                    ⟹  #t
```

Rationale: It will usually be possible to implement `eq?` much more efficiently than `eqv?`, for example, as a simple pointer comparison instead of as some more complicated operation. One reason is that it may not be possible to compute `eqv?` of two numbers in constant time, whereas `eq?` implemented as pointer comparison will always finish in constant time. `Eq?` may be used like `eqv?` in applications using procedures to implement objects with state since it obeys the same constraints as `eqv?`.

`(equal?` obj_1 obj_2`)` library procedure

`Equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers and symbols. A rule of thumb is that objects are generally `equal?` if they print the same. `Equal?` may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)                  ⟹  #t
(equal? '(a) '(a))              ⟹  #t
(equal? '(a (b) c)
        '(a (b) c))             ⟹  #t
(equal? "abc" "abc")            ⟹  #t
(equal? 2 2)                    ⟹  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))     ⟹  #t
(equal? (lambda (x) x)
        (lambda (y) y))         ⟹  unspecified
```

6.2. Numbers

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there was no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system [20] little effort was made to execute numerical code efficiently. This report recognizes the excellent work of the Common Lisp committee and accepts many of their recommendations. In some ways this report simplifies and generalizes their proposals in a manner consistent with the purposes of Scheme.

It is important to distinguish between the mathematical numbers, the Scheme numbers that attempt to model them, the machine representations used to implement the Scheme numbers, and notations used to write numbers. This report uses the types *number*, *complex*, *real*, *rational*, and *integer* to refer to both mathematical numbers and Scheme numbers. Machine representations such as fixed point and floating point are referred to by names such as *fixnum* and *flonum*.

6.2.1. Numerical types

Mathematically, numbers may be arranged into a tower of subtypes in which each level is a subset of the level above it:

number
complex
real
rational
integer

For example, 3 is an integer. Therefore 3 is also a rational, a real, and a complex. The same is true of the Scheme numbers that model 3. For Scheme numbers, these types are defined by the predicates `number?`, `complex?`, `real?`, `rational?`, and `integer?`.

There is no simple relationship between a number's type and its representation inside a computer. Although most implementations of Scheme will offer at least two different representations of 3, these different representations denote the same integer.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use fixnum, flonum, and perhaps other representations for numbers, this should not be apparent to a casual programmer writing simple programs.

It is necessary, however, to distinguish between numbers that are represented exactly and those that may not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

6.2.2. Exactness

Scheme numbers are either *exact* or *inexact*. A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is inexact if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus inexactness is a contagious property of a number.

If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equivalent. This is generally not true of computations involving inexact numbers since approximate methods such as floating point arithmetic may be used, but it is the duty of each implementation to make the result as close as practical to the mathematically ideal result.

Rational operations such as `+` should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction or it may silently coerce its result to an inexact value. See section 6.2.3.

With the exception of `inexact->exact`, the operations described in this section must generally return inexact results when given any inexact arguments. An operation may, however, return an exact result if it can prove that the value of the result is unaffected by the inexactness of its arguments. For example, multiplication of any number by an exact zero may produce an exact zero result, even if the other argument is inexact.

6.2.3. Implementation restrictions

Implementations of Scheme are not required to implement the whole tower of subtypes given in section 6.2.1, but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language. For example, an implementation in which all numbers are real may still be quite useful.

Implementations may also support only a limited range of numbers of any type, subject to the requirements of this section. The supported range for exact numbers of any type may be different from the supported range for inexact numbers of that type. For example, an implementation that uses flonums to represent all its inexact real numbers may support a practically unbounded range of exact integers and rationals while limiting the range of inexact reals (and therefore the range of inexact integers and rationals) to the dynamic range of the flonum format. Furthermore the gaps between the representable inexact integers and rationals are likely to be very large in such an implementation as the limits of this range are approached.

An implementation of Scheme must support exact integers throughout the range of numbers that may be used for indexes of lists, vectors, and strings or that may result from computing the length of a list, vector, or string. The `length`, `vector-length`, and `string-length` procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore any integer constant within the index range, if expressed by an exact integer syntax, will indeed be read as an exact integer, regardless of any implementation restrictions that may apply outside this range. Finally, the procedures listed below will always return an exact integer result provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer within the implementation:

```
+            -              *
quotient     remainder      modulo
max          min            abs
numerator    denominator    gcd
lcm          floor          ceiling
truncate     round          rationalize
expt
```

Implementations are encouraged, but not required, to support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the `/` procedure in such a way that they always return exact results when given exact arguments. If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number. Such a coercion may cause an error later.

An implementation may use floating point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that the IEEE 32-bit and 64-bit floating point standards be followed by implementations that use flonum representations, and that implementations using other representations should match or exceed the precision achievable using these floating point standards [12].

In particular, implementations that use flonum representations must follow these rules: A flonum result must be represented with at least as much precision as is used to express any of the inexact arguments to that operation. It is desirable (but not required) for potentially inexact operations such as `sqrt`, when applied to exact arguments, to produce exact answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If, however, an exact number is operated upon so as to produce an inexact result (as by `sqrt`), and if the result is represented as a flonum, then the most precise flonum format available must be used; but if the result is represented in some other way then the representation must have at least as much precision as the most precise flonum format available.

Although Scheme allows a variety of written notations for numbers, any particular implementation may support only some of them. For example, an implementation in which all numbers are real need not support the rectangular and polar notations for complex numbers. If an implementation encounters an exact numerical constant that it cannot represent as an exact number, then it may either report a violation of an implementation restriction or it may silently represent the constant by an inexact number.

6.2.4. Syntax of numerical constants

The syntax of the written representations for numbers is described formally in section 7.1.1. Note that case is not significant in numerical constants.

A number may be written in binary, octal, decimal, or hexadecimal by the use of a radix prefix. The radix prefixes are `#b` (binary), `#o` (octal), `#d` (decimal), and `#x` (hexadecimal). With no radix prefix, a number is assumed to be expressed in decimal.

A numerical constant may be specified to be either exact or inexact by a prefix. The prefixes are `#e` for exact, and `#i` for inexact. An exactness prefix may appear before or after any radix prefix that is used. If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a "`#`" character in the place of a digit, otherwise it is exact.

In systems with inexact numbers of varying precisions it may be useful to specify the precision of a constant. For this purpose, numerical constants may be written with an exponent marker that indicates the desired precision of the inexact representation. The letters `s`, `f`, `d`, and `l` specify the use of *short*, *single*, *double*, and *long* precision, respectively. (When fewer than four internal inexact representations exist, the four size specifications are mapped onto those available. For example, an implementation with two internal representations may map short and single together and long and double together.) In addition, the exponent marker `e` specifies the default precision for the implementation. The default precision has at least as much precision as *double*, but implementations may wish to allow this default to be set by the user.

```
3.14159265358979F0
        Round to single — 3.141593
0.6L0
        Extend to long — .600000000000000
```

6.2.5. Numerical operations

The reader is referred to section 1.3.3 for a summary of the naming conventions used to specify restrictions on the types of arguments to numerical routines. The examples used in this section assume that any numerical constant written using an exact notation is indeed represented as an exact number. Some examples also assume that certain numerical constants written using an inexact notation can be represented without loss of accuracy; the inexact constants were chosen so that this is likely to be true in implementations that use flonums to represent inexact numbers.

(number? *obj*) procedure
(complex? *obj*) procedure
(real? *obj*) procedure
(rational? *obj*) procedure
(integer? *obj*) procedure

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is of the named type, and otherwise they return `#f`. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

If z is an inexact complex number, then `(real?` z`)` is true if and only if `(zero? (imag-part` z`))` is true. If x is an inexact real number, then `(integer?` x`)` is true if and only if `(=` x `(round` x`))`.

```
(complex? 3+4i)          ⟹  #t
(complex? 3)             ⟹  #t
(real? 3)                ⟹  #t
(real? -2.5+0.0i)        ⟹  #t
(real? #e1e10)           ⟹  #t
(rational? 6/10)         ⟹  #t
(rational? 6/3)          ⟹  #t
(integer? 3+0i)          ⟹  #t
(integer? 3.0)           ⟹  #t
(integer? 8/4)           ⟹  #t
```

Note: The behavior of these type predicates on inexact numbers is unreliable, since any inaccuracy may affect the result.

Note: In many implementations the `rational?` procedure will be the same as `real?`, and the `complex?` procedure will be the same as `number?`, but unusual implementations may be able to represent some irrational numbers exactly or may extend the number system to support some kind of non-complex numbers.

(exact? *z*) procedure
(inexact? *z*) procedure

These numerical predicates provide tests for the exactness of a quantity. For any Scheme number, precisely one of these predicates is true.

(= z_1 z_2 z_3 . . .) procedure
(< x_1 x_2 x_3 . . .) procedure
(> x_1 x_2 x_3 . . .) procedure
(<= x_1 x_2 x_3 . . .) procedure
(>= x_1 x_2 x_3 . . .) procedure

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

These predicates are required to be transitive.

Note: The traditional implementations of these predicates in Lisp-like languages are not transitive.

Note: While it is not an error to compare inexact numbers using these predicates, the results may be unreliable because a small inaccuracy may affect the result; this is especially true of `=` and `zero?`. When in doubt, consult a numerical analyst.

`(zero?` z`)` library procedure

`(positive?` x`)` library procedure

`(negative?` x`)` library procedure

`(odd?` n`)` library procedure

`(even?` n`)` library procedure

These numerical predicates test a number for a particular property, returning `#t` or `#f`. See note above.

`(max` x_1 x_2 `...)` library procedure

`(min` x_1 x_2 `...)` library procedure

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)                ⟹  4    ; exact
(max 3.9 4)              ⟹  4.0  ; inexact
```

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If `min` or `max` is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.

`(+` z_1 `...)` procedure

`(*` z_1 `...)` procedure

These procedures return the sum or product of their arguments.

```
(+ 3 4)                  ⟹  7
(+ 3)                    ⟹  3
(+)                      ⟹  0
(* 4)                    ⟹  4
(*)                      ⟹  1
```

`(-` z_1 z_2`)` procedure

`(-` z`)` procedure

`(-` z_1 z_2 `...)` optional procedure

`(/` z_1 z_2`)` procedure

`(/` z`)` procedure

`(/` z_1 z_2 `...)` optional procedure

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

```
(- 3 4)                  ⟹  -1
(- 3 4 5)                ⟹  -6
(- 3)                    ⟹  -3
(/ 3 4 5)                ⟹  3/20
(/ 3)                    ⟹  1/3
```

`(abs` x`)` library procedure

`Abs` returns the absolute value of its argument.

```
(abs -7)                 ⟹  7
```

`(quotient` n_1 n_2`)` procedure

`(remainder` n_1 n_2`)` procedure

`(modulo` n_1 n_2`)` procedure

These procedures implement number-theoretic (integer) division. n_2 should be non-zero. All three procedures return integers. If n_1/n_2 is an integer:

```
    (quotient n1 n2)     ⟹ n1/n2
    (remainder n1 n2)    ⟹ 0
    (modulo n1 n2)       ⟹ 0
```

If n_1/n_2 is not an integer:

```
    (quotient n1 n2)     ⟹ nq
    (remainder n1 n2)    ⟹ nr
    (modulo n1 n2)       ⟹ nm
```

where n_q is n_1/n_2 rounded towards zero, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r and n_m differ from n_1 by a multiple of n_2, n_r has the same sign as n_1, and n_m has the same sign as n_2.

From this we can conclude that for integers n_1 and n_2 with n_2 not equal to 0,

```
     (= n1 (+ (* n2 (quotient n1 n2))
           (remainder n1 n2)))
                         ⟹  #t
```

provided all numbers involved in that computation are exact.

```
(modulo 13 4)            ⟹  1
(remainder 13 4)         ⟹  1

(modulo -13 4)           ⟹  3
(remainder -13 4)        ⟹  -1

(modulo 13 -4)           ⟹  -3
(remainder 13 -4)        ⟹  1

(modulo -13 -4)          ⟹  -1
(remainder -13 -4)       ⟹  -1

(remainder -13 -4.0)     ⟹  -1.0  ; inexact
```

(gcd n_1 ...) library procedure
(lcm n_1 ...) library procedure

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative.

```
(gcd 32 -36)            ⟹  4
(gcd)                   ⟹  0
(lcm 32 -36)            ⟹  288
(lcm 32.0 -36)          ⟹  288.0  ; inexact
(lcm)                   ⟹  1
```

(numerator q) procedure
(denominator q) procedure

These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

```
(numerator (/ 6 4))         ⟹  3
(denominator (/ 6 4))       ⟹  2
(denominator
  (exact->inexact (/ 6 4))) ⟹ 2.0
```

(floor x) procedure
(ceiling x) procedure
(truncate x) procedure
(round x) procedure

These procedures return integers. `Floor` returns the largest integer not larger than x. `Ceiling` returns the smallest integer not smaller than x. `Truncate` returns the integer closest to x whose absolute value is not larger than the absolute value of x. `Round` returns the closest integer to x, rounding to even when x is halfway between two integers.

Rationale: `Round` rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

Note: If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result should be passed to the `inexact->exact` procedure.

```
(floor -4.3)            ⟹  -5.0
(ceiling -4.3)          ⟹  -4.0
(truncate -4.3)         ⟹  -4.0
(round -4.3)            ⟹  -4.0

(floor 3.5)             ⟹  3.0
(ceiling 3.5)           ⟹  4.0
(truncate 3.5)          ⟹  3.0
(round 3.5)             ⟹  4.0  ; inexact

(round 7/2)             ⟹  4    ; exact
(round 7)               ⟹  7
```

(rationalize x y) library procedure

`Rationalize` returns the *simplest* rational number differing from x by no more than y. A rational number r_1 is *simpler* than another rational number r_2 if $r_1 = p_1/q_1$ and $r_2 = p_2/q_2$ (in lowest terms) and $|p_1| \leq |p_2|$ and $|q_1| \leq |q_2|$. Thus 3/5 is simpler than 4/7. Although not all rationals are comparable in this ordering (consider 2/7 and 3/5) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler 2/5 lies between 2/7 and 3/5). Note that 0 = 0/1 is the simplest rational of all.

```
(rationalize
  (inexact->exact .3) 1/10) ⟹ 1/3    ; exact
(rationalize .3 1/10)       ⟹ #i1/3  ; inexact
```

(exp z) procedure
(log z) procedure
(sin z) procedure
(cos z) procedure
(tan z) procedure
(asin z) procedure
(acos z) procedure
(atan z) procedure
(atan y x) procedure

These procedures are part of every implementation that supports general real numbers; they compute the usual transcendental functions. `Log` computes the natural logarithm of z (not the base ten logarithm). `Asin`, `acos`, and `atan` compute arcsine ($\sin^{-1}$), arccosine ($\cos^{-1}$), and arctangent ($\tan^{-1}$), respectively. The two-argument variant of `atan` computes `(angle (make-rectangular` x y`))` (see below), even in implementations that don't support general complex numbers.

In general, the mathematical functions log, arcsine, arccosine, and arctangent are multiply defined. The value of $\log z$ is defined to be the one whose imaginary part lies in the range from $-\pi$ (exclusive) to π (inclusive). $\log 0$ is undefined. With log defined this way, the values of $\sin^{-1} z$, $\cos^{-1} z$, and $\tan^{-1} z$ are according to the following formulæ:

$$\sin^{-1} z = -i \log(iz + \sqrt{1 - z^2})$$

$$\cos^{-1} z = \pi/2 - \sin^{-1} z$$

$$\tan^{-1} z = (\log(1 + iz) - \log(1 - iz))/(2i)$$

The above specification follows [27], which in turn cites [19]; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions. When it is possible these procedures produce a real result from a real argument.

`(sqrt` z`)` procedure

Returns the principal square root of z. The result will have either positive real part, or zero real part and non-negative imaginary part.

`(expt` z_1 z_2`)` procedure

Returns z_1 raised to the power z_2. For $z_1 \neq 0$

$$z_1{}^{z_2} = e^{z_2 \log z_1}$$

0^z is 1 if $z = 0$ and 0 otherwise.

`(make-rectangular` x_1 x_2`)` procedure
`(make-polar` x_3 x_4`)` procedure
`(real-part` z`)` procedure
`(imag-part` z`)` procedure
`(magnitude` z`)` procedure
`(angle` z`)` procedure

These procedures are part of every implementation that supports general complex numbers. Suppose x_1, x_2, x_3, and x_4 are real numbers and z is a complex number such that

$$z = x_1 + x_2 i = x_3 \cdot e^{i x_4}$$

Then

`(make-rectangular` x_1 x_2`)` $\Longrightarrow z$
`(make-polar` x_3 x_4`)` $\Longrightarrow z$
`(real-part` z`)` $\Longrightarrow x_1$
`(imag-part` z`)` $\Longrightarrow x_2$
`(magnitude` z`)` $\Longrightarrow |x_3|$
`(angle` z`)` $\Longrightarrow x_{angle}$

where $-\pi < x_{angle} \leq \pi$ with $x_{angle} = x_4 + 2\pi n$ for some integer n.

Rationale: `Magnitude` is the same as `abs` for a real argument, but `abs` must be present in all implementations, whereas `magnitude` need only be present in implementations that support general complex numbers.

`(exact->inexact` z`)` procedure
`(inexact->exact` z`)` procedure

`Exact->inexact` returns an inexact representation of z. The value returned is the inexact number that is numerically closest to the argument. If an exact argument has no reasonably close inexact equivalent, then a violation of an implementation restriction may be reported.

`Inexact->exact` returns an exact representation of z. The value returned is the exact number that is numerically closest to the argument. If an inexact argument has no reasonably close exact equivalent, then a violation of an implementation restriction may be reported.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range. See section 6.2.3.

6.2.6. Numerical input and output

`(number->string` z`)` procedure
`(number->string` z *radix*`)` procedure

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. The procedure `number->string` takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number
                                        radix)
                        radix)))
```

is true. It is an error if no possible result makes this expression true.

If z is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true [3, 5]; otherwise the format of the result is unspecified.

The result returned by `number->string` never contains an explicit radix prefix.

Note: The error case can occur only when z is not a complex number or is a complex number with a non-rational real or imaginary part.

Rationale: If z is an inexact number represented using flonums, and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and non-flonum representations.

`(string->number` *string*`)` procedure
`(string->number` *string radix*`)` procedure

Returns a number of the maximally precise representation expressed by the given *string*. *Radix* must be an exact integer, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g. `"#o177"`). If *radix* is not supplied, then the default radix is 10. If *string* is not a syntactically valid notation for a number, then `string->number` returns `#f`.

```
(string->number "100")        ⟹  100
(string->number "100" 16)     ⟹  256
(string->number "1e2")        ⟹  100.0
(string->number "15##")       ⟹  1500.0
```

Note: The domain of `string->number` may be restricted by implementations in the following ways. `String->number` is permitted to return `#f` whenever *string* contains an explicit radix prefix. If all numbers supported by an implementation are real, then `string->number` is permitted to return `#f` whenever *string* uses the polar or rectangular notations for complex numbers. If all numbers are integers, then `string->number` may return `#f` whenever the fractional notation is used. If all numbers are exact, then `string->number` may return `#f` whenever an exponent marker or explicit exactness prefix is used, or if a `#` appears in place of a digit. If all inexact numbers are integers, then `string->number` may return `#f` whenever a decimal point is used.

6.3. Other data types

This section describes operations on some of Scheme's non-numeric data types: booleans, pairs, lists, symbols, characters, strings and vectors.

6.3.1. Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. What really matters, though, are the objects that the Scheme conditional expressions (`if`, `cond`, `and`, `or`, `do`) treat as true or false. The phrase "a true value" (or sometimes just "true") means any object treated as true by the conditional expressions, and the phrase "a false value" (or "false") means any object treated as false by the conditional expressions.

Of all the standard Scheme values, only `#f` counts as false in conditional expressions. Except for `#f`, all standard Scheme values, including `#t`, pairs, the empty list, symbols, numbers, strings, vectors, and procedures, count as true.

Note: Programmers accustomed to other dialects of Lisp should be aware that Scheme distinguishes both `#f` and the empty list from the symbol `nil`.

Boolean constants evaluate to themselves, so they do not need to be quoted in programs.

```
#t                     ⟹  #t
#f                     ⟹  #f
'#f                    ⟹  #f
```

`(not` *obj*`)` library procedure

`Not` returns `#t` if *obj* is false, and returns `#f` otherwise.

```
(not #t)               ⟹  #f
(not 3)                ⟹  #f
(not (list 3))         ⟹  #f
(not #f)               ⟹  #t
(not '())              ⟹  #f
(not (list))           ⟹  #f
(not 'nil)             ⟹  #f
```

`(boolean?` *obj*`)` library procedure

`Boolean?` returns `#t` if *obj* is either `#t` or `#f` and returns `#f` otherwise.

```
(boolean? #f)          ⟹  #t
(boolean? 0)           ⟹  #f
(boolean? '())         ⟹  #f
```

6.3.2. Pairs and lists

A *pair* (sometimes called a *dotted pair*) is a record structure with two fields called the car and cdr fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. The car and cdr fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set X such that

- The empty list is in X.
- If *list* is in X, then any pair whose cdr field contains *list* is also in X.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type (it is not a pair); it has no elements and its length is zero.

Note: The above definitions imply that all lists have finite length and are terminated by the empty list.

The most general notation (external representation) for Scheme pairs is the "dotted" notation `(`c_1 `.` c_2`)` where c_1 is the value of the car field and c_2 is the value of the cdr field. For example `(4 . 5)` is a pair whose car is 4 and whose cdr is 5. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()` . For example,

```
(a b c d e)
```

and

```
(a . (b . (c . (d . (e . ())))))
```

are equivalent notations for a list of symbols.

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

```
(a b c . d)
```

is equivalent to

```
(a . (b . (c . d)))
```

Whether a given pair is a list depends upon what is stored in the cdr field. When the `set-cdr!` procedure is used, an object can be a list one moment and not the next:

```
(define x (list 'a 'b 'c))
(define y x)
y                           ⟹  (a b c)
(list? y)                   ⟹  #t
(set-cdr! x 4)              ⟹  unspecified
x                           ⟹  (a . 4)
(eqv? x y)                  ⟹  #t
y                           ⟹  (a . 4)
(list? y)                   ⟹  #f
(set-cdr! x x)              ⟹  unspecified
(list? x)                   ⟹  #f
```

Within literal expressions and representations of objects read by the `read` procedure, the forms `'`⟨datum⟩, `` ` ``⟨datum⟩, `,`⟨datum⟩, and `,@`⟨datum⟩ denote two-element lists whose first elements are the symbols `quote`, `quasiquote`, `unquote`, and `unquote-splicing`, respectively. The second element in each case is ⟨datum⟩. This convention is supported so that arbitrary Scheme programs may be represented as lists. That is, according to Scheme's grammar, every ⟨expression⟩ is also a ⟨datum⟩ (see section 7.1.2). Among other things, this permits the use of the `read` procedure to parse Scheme programs. See section 3.3.

`(pair?` *obj*`)` procedure

`Pair?` returns `#t` if *obj* is a pair, and otherwise returns `#f`.

```
(pair? '(a . b))            ⟹  #t
(pair? '(a b c))            ⟹  #t
(pair? '())                 ⟹  #f
(pair? '#(a b))             ⟹  #f
```

`(cons` *obj*$_1$ *obj*$_2$`)` procedure

Returns a newly allocated pair whose car is *obj*$_1$ and whose cdr is *obj*$_2$. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

```
(cons 'a '())               ⟹  (a)
(cons '(a) '(b c d))        ⟹  ((a) b c d)
(cons "a" '(b c))           ⟹  ("a" b c)
(cons 'a 3)                 ⟹  (a . 3)
(cons '(a b) 'c)            ⟹  ((a b) . c)
```

`(car` *pair*`)` procedure

Returns the contents of the car field of *pair*. Note that it is an error to take the car of the empty list.

```
(car '(a b c))              ⟹  a
(car '((a) b c d))          ⟹  (a)
(car '(1 . 2))              ⟹  1
(car '())                   ⟹  error
```

`(cdr` *pair*`)` procedure

Returns the contents of the cdr field of *pair*. Note that it is an error to take the cdr of the empty list.

```
(cdr '((a) b c d))          ⟹  (b c d)
(cdr '(1 . 2))              ⟹  2
(cdr '())                   ⟹  error
```

`(set-car!` *pair obj*`)` procedure

Stores *obj* in the car field of *pair*. The value returned by `set-car!` is unspecified.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)            ⟹  unspecified
(set-car! (g) 3)            ⟹  error
```

`(set-cdr!` *pair obj*`)` procedure

Stores *obj* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

`(caar` *pair*`)` library procedure
`(cadr` *pair*`)` library procedure
⋮ ⋮
`(cdddar` *pair*`)` library procedure
`(cddddr` *pair*`)` library procedure

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x))))).
```

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

`(null?` *obj*`)` library procedure

Returns `#t` if *obj* is the empty list, otherwise returns `#f`.

`(list?` *obj*`)` library procedure

Returns `#t` if *obj* is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list.

```
(list? '(a b c))     ⟹  #t
(list? '())          ⟹  #t
(list? '(a . b))     ⟹  #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))         ⟹  #f
```

(list *obj* ...)   library procedure

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)          ⟹  (a 7 c)
(list)                        ⟹  ()
```

(length *list*)   library procedure

Returns the length of *list*.

```
(length '(a b c))             ⟹  3
(length '(a (b) (c d e)))     ⟹  3
(length '())                  ⟹  0
```

(append *list* ...)   library procedure

Returns a list consisting of the elements of the first *list* followed by the elements of the other *list*s.

```
(append '(x) '(y))            ⟹  (x y)
(append '(a) '(b c d))        ⟹  (a b c d)
(append '(a (b)) '((c)))      ⟹  (a (b) (c))
```

The resulting list is always newly allocated, except that it shares structure with the last *list* argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))      ⟹  (a b c . d)
(append '() 'a)               ⟹  a
```

(reverse *list*)   library procedure

Returns a newly allocated list consisting of the elements of *list* in reverse order.

```
(reverse '(a b c))            ⟹  (c b a)
(reverse '(a (b c) d (e (f))))
          ⟹  ((e (f)) d (b c) a)
```

(list-tail *list k*)   library procedure

Returns the sublist of *list* obtained by omitting the first *k* elements. It is an error if *list* has fewer than *k* elements. List-tail could be defined by

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

(list-ref *list k*)   library procedure

Returns the *k*th element of *list*. (This is the same as the car of (list-tail *list k*).) It is an error if *list* has fewer than *k* elements.

```
(list-ref '(a b c d) 2)       ⟹  c
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))
          ⟹  c
```

(memq *obj list*)   library procedure
(memv *obj list*)   library procedure
(member *obj list*)   library procedure

These procedures return the first sublist of *list* whose car is *obj*, where the sublists of *list* are the non-empty lists returned by (list-tail *list k*) for *k* less than the length of *list*. If *obj* does not occur in *list*, then #f (not the empty list) is returned. Memq uses eq? to compare *obj* with the elements of *list*, while memv uses eqv? and member uses equal?.

```
(memq 'a '(a b c))            ⟹  (a b c)
(memq 'b '(a b c))            ⟹  (b c)
(memq 'a '(b c d))            ⟹  #f
(memq (list 'a) '(b (a) c))   ⟹  #f
(member (list 'a)
        '(b (a) c))           ⟹  ((a) c)
(memq 101 '(100 101 102))     ⟹  unspecified
(memv 101 '(100 101 102))     ⟹  (101 102)
```

(assq *obj alist*)   library procedure
(assv *obj alist*)   library procedure
(assoc *obj alist*)   library procedure

Alist (for "association list") must be a list of pairs. These procedures find the first pair in *alist* whose car field is *obj*, and returns that pair. If no pair in *alist* has *obj* as its car, then #f (not the empty list) is returned. Assq uses eq? to compare *obj* with the car fields of the pairs in *alist*, while assv uses eqv? and assoc uses equal?.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                   ⟹  (a 1)
(assq 'b e)                   ⟹  (b 2)
(assq 'd e)                   ⟹  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                              ⟹  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                              ⟹  ((a))
(assq 5 '((2 3) (5 7) (11 13)))
                              ⟹  unspecified
(assv 5 '((2 3) (5 7) (11 13)))
                              ⟹  (5 7)
```

Rationale: Although they are ordinarily used as predicates, memq, memv, member, assq, assv, and assoc do not have question marks in their names because they return useful values rather than just #t or #f.

6.3.3. Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eqv?`) if and only if their names are spelled the same way. This is exactly the property needed to represent identifiers in programs, and so most implementations of Scheme use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated values are used in Pascal.

The rules for writing a symbol are exactly the same as the rules for writing an identifier; see sections 2.1 and 7.1.1.

It is guaranteed that any symbol that has been returned as part of a literal expression, or read using the `read` procedure, and subsequently written out using the `write` procedure, will read back in as the identical symbol (in the sense of `eqv?`). The `string->symbol` procedure, however, can create symbols for which this write/read invariance may not hold because their names contain special characters or letters in the non-standard case.

Note: Some implementations of Scheme have a feature known as “slashification” in order to guarantee write/read invariance for all symbols, but historically the most important use of this feature has been to compensate for the lack of a string data type.

Some implementations also have “uninterned symbols”, which defeat write/read invariance even in implementations with slashification, and also generate exceptions to the rule that two symbols are the same if and only if their names are spelled the same.

`(symbol?` *obj*`)` procedure

Returns `#t` if *obj* is a symbol, otherwise returns `#f`.

```
(symbol? 'foo)            ⟹  #t
(symbol? (car '(a b)))    ⟹  #t
(symbol? "bar")           ⟹  #f
(symbol? 'nil)            ⟹  #t
(symbol? '())             ⟹  #f
(symbol? #f)              ⟹  #f
```

`(symbol->string` *symbol*`)` procedure

Returns the name of *symbol* as a string. If the symbol was part of an object returned as the value of a literal expression (section 4.1.2) or by a call to the `read` procedure, and its name contains alphabetic characters, then the string returned will contain characters in the implementation's preferred standard case—some implementations will prefer upper case, others lower case. If the symbol was returned by `string->symbol`, the case of characters in the string returned will be the same as the case in the string that was passed to `string->symbol`. It is an error to apply mutation procedures like `string-set!` to strings returned by this procedure.

The following examples assume that the implementation's standard case is lower case:

```
(symbol->string 'flying-fish)
                              ⟹  "flying-fish"
(symbol->string 'Martin)      ⟹  "martin"
(symbol->string
   (string->symbol "Malvina"))
                              ⟹  "Malvina"
```

`(string->symbol` *string*`)` procedure

Returns the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves. See `symbol->string`.

The following examples assume that the implementation's standard case is lower case:

```
(eq? 'mISSISSIppi 'mississippi)
          ⟹  #t
(string->symbol "mISSISSIppi")
          ⟹  the symbol with name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
          ⟹  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
          ⟹  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
          ⟹  #t
```

6.3.4. Characters

Characters are objects that represent printed characters such as letters and digits. Characters are written using the notation `#\`⟨character⟩ or `#\`⟨character name⟩. For example:

```
#\a          ; lower case letter
#\A          ; upper case letter
#\(          ; left parenthesis
#\           ; the space character
#\space      ; the preferred way to write a space
#\newline    ; the newline character
```

Case is significant in `#\`⟨character⟩, but not in `#\`⟨character name⟩. If ⟨character⟩ in `#\`⟨character⟩ is alphabetic, then the character following ⟨character⟩ must be a delimiter character such as a space or parenthesis. This rule resolves the ambiguous case where, for example, the sequence of characters "`#\space`" could be taken to be either a representation of the space character or a representation of the character "`#\s`" followed by a representation of the symbol "`pace`."

Characters written in the `#\` notation are self-evaluating. That is, they do not have to be quoted in programs.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have "`-ci`" (for "case insensitive") embedded in their names.

`(char?` *obj*`)` procedure

Returns `#t` if *obj* is a character, otherwise returns `#f`.

`(char=?` *char*$_1$ *char*$_2$`)` procedure
`(char<?` *char*$_1$ *char*$_2$`)` procedure
`(char>?` *char*$_1$ *char*$_2$`)` procedure
`(char<=?` *char*$_1$ *char*$_2$`)` procedure
`(char>=?` *char*$_1$ *char*$_2$`)` procedure

These procedures impose a total ordering on the set of characters. It is guaranteed that under this ordering:

- The upper case characters are in order. For example, `(char<? #\A #\B)` returns `#t`.
- The lower case characters are in order. For example, `(char<? #\a #\b)` returns `#t`.
- The digits are in order. For example, `(char<? #\0 #\9)` returns `#t`.
- Either all the digits precede all the upper case letters, or vice versa.
- Either all the digits precede all the lower case letters, or vice versa.

Some implementations may generalize these procedures to take more than two arguments, as with the corresponding numerical predicates.

`(char-ci=?` *char*$_1$ *char*$_2$`)` library procedure
`(char-ci<?` *char*$_1$ *char*$_2$`)` library procedure
`(char-ci>?` *char*$_1$ *char*$_2$`)` library procedure
`(char-ci<=?` *char*$_1$ *char*$_2$`)` library procedure
`(char-ci>=?` *char*$_1$ *char*$_2$`)` library procedure

These procedures are similar to `char=?` et cetera, but they treat upper case and lower case letters as the same. For example, `(char-ci=? #\A #\a)` returns `#t`. Some implementations may generalize these procedures to take more than two arguments, as with the corresponding numerical predicates.

`(char-alphabetic?` *char*`)` library procedure
`(char-numeric?` *char*`)` library procedure
`(char-whitespace?` *char*`)` library procedure
`(char-upper-case?` *letter*`)` library procedure
`(char-lower-case?` *letter*`)` library procedure

These procedures return `#t` if their arguments are alphabetic, numeric, whitespace, upper case, or lower case characters, respectively, otherwise they return `#f`. The following remarks, which are specific to the ASCII character set, are intended only as a guide: The alphabetic characters are the 52 upper and lower case letters. The numeric characters are the ten decimal digits. The whitespace characters are space, tab, line feed, form feed, and carriage return.

`(char->integer` *char*`)` procedure
`(integer->char` *n*`)` procedure

Given a character, `char->integer` returns an exact integer representation of the character. Given an exact integer that is the image of a character under `char->integer`, `integer->char` returns that character. These procedures implement order-preserving isomorphisms between the set of characters under the `char<=?` ordering and some subset of the integers under the `<=` ordering. That is, if

```
(char<=? a b) ⟹ #t  and  (<= x y) ⟹ #t
```

and *x* and *y* are in the domain of `integer->char`, then

```
(<= (char->integer a)
    (char->integer b))      ⟹  #t

(char<=? (integer->char x)
         (integer->char y)) ⟹  #t
```

`(char-upcase` *char*`)` library procedure
`(char-downcase` *char*`)` library procedure

These procedures return a character *char*$_2$ such that `(char-ci=?` *char* *char*$_2$`)`. In addition, if *char* is alphabetic, then the result of `char-upcase` is upper case and the result of `char-downcase` is lower case.

6.3.5. Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within doublequotes (`"`). A doublequote can be written inside a string only by escaping it with a backslash (\), as in

```
"The word \"recursion\" has many meanings."
```

A backslash can be written inside a string only by escaping it with another backslash. Scheme does not specify the effect of a backslash within a string that is not followed by a doublequote or backslash.

A string constant may continue from one line to the next, but the exact contents of such a string are unspecified.

The *length* of a string is the number of characters that it contains. This number is an exact, non-negative integer that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

In phrases such as "the characters of *string* beginning with index *start* and ending with index *end*," it is understood that the index *start* is inclusive and the index *end* is exclusive. Thus if *start* and *end* are the same index, a null substring is referred to, and if *start* is zero and *end* is the length of *string*, then the entire string is referred to.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The versions that ignore case have "`-ci`" (for "case insensitive") embedded in their names.

`(string?` *obj*`)` procedure

Returns `#t` if *obj* is a string, otherwise returns `#f`.

`(make-string` *k*`)` procedure
`(make-string` *k char*`)` procedure

`Make-string` returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

`(string` *char* ...`)` library procedure

Returns a newly allocated string composed of the arguments.

`(string-length` *string*`)` procedure

Returns the number of characters in the given *string*.

`(string-ref` *string k*`)` procedure

k must be a valid index of *string*. `String-ref` returns character *k* of *string* using zero-origin indexing.

`(string-set!` *string k char*`)` procedure

k must be a valid index of *string*. `String-set!` stores *char* in element *k* of *string* and returns an unspecified value.

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)      ⟹  unspecified
(string-set! (g) 0 #\?)      ⟹  error
(string-set! (symbol->string 'immutable)
             0
             #\?)            ⟹  error
```

`(string=?` *string*$_1$ *string*$_2$`)` library procedure
`(string-ci=?` *string*$_1$ *string*$_2$`)` library procedure

Returns `#t` if the two strings are the same length and contain the same characters in the same positions, otherwise returns `#f`. `String-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

`(string<?` *string*$_1$ *string*$_2$`)` library procedure
`(string>?` *string*$_1$ *string*$_2$`)` library procedure
`(string<=?` *string*$_1$ *string*$_2$`)` library procedure
`(string>=?` *string*$_1$ *string*$_2$`)` library procedure
`(string-ci<?` *string*$_1$ *string*$_2$`)` library procedure
`(string-ci>?` *string*$_1$ *string*$_2$`)` library procedure
`(string-ci<=?` *string*$_1$ *string*$_2$`)` library procedure
`(string-ci>=?` *string*$_1$ *string*$_2$`)` library procedure

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

Implementations may generalize these and the `string=?` and `string-ci=?` procedures to take more than two arguments, as with the corresponding numerical predicates.

`(substring` *string start end*`)` library procedure

String must be a string, and *start* and *end* must be exact integers satisfying

$$0 \le \textit{start} \le \textit{end} \le \texttt{(string-length } \textit{string}\texttt{)}.$$

`Substring` returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

`(string-append` *string* ...`)` library procedure

Returns a newly allocated string whose characters form the concatenation of the given strings.

`(string->list` *string*`)` library procedure
`(list->string` *list*`)` library procedure

`String->list` returns a newly allocated list of the characters that make up the given string. `List->string` returns a newly allocated string formed from the characters in the list *list*, which must be a list of characters. `String->list` and `list->string` are inverses so far as `equal?` is concerned.

`(string-copy` *string*`)` library procedure

Returns a newly allocated copy of the given *string*.

(`string-fill!` *string char*) library procedure

Stores *char* in every element of the given *string* and returns an unspecified value.

6.3.6. Vectors

Vectors are heterogenous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indexes* of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation `#(`*obj* ...`)`. For example, a vector of length 3 containing the number zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be written as following:

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector, not an expression evaluating to a vector. Like list constants, vector constants must be quoted:

```
'#(0 (2 2 2 2) "Anna")
          ⟹  #(0 (2 2 2 2) "Anna")
```

(`vector?` *obj*) procedure

Returns `#t` if *obj* is a vector, otherwise returns `#f`.

(`make-vector` *k*) procedure
(`make-vector` *k fill*) procedure

Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

(`vector` *obj* ...) library procedure

Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)           ⟹  #(a b c)
```

(`vector-length` *vector*) procedure

Returns the number of elements in *vector* as an exact integer.

(`vector-ref` *vector k*) procedure

k must be a valid index of *vector*. `Vector-ref` returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5)
          ⟹  8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
          ⟹  13
```

(`vector-set!` *vector k obj*) procedure

k must be a valid index of *vector*. `Vector-set!` stores *obj* in element *k* of *vector*. The value returned by `vector-set!` is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
          ⟹  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
          ⟹  error  ; constant vector
```

(`vector->list` *vector*) library procedure
(`list->vector` *list*) library procedure

`Vector->list` returns a newly allocated list of the objects contained in the elements of *vector*. `List->vector` returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '#(dah dah didah))
          ⟹  (dah dah didah)
(list->vector '(dididit dah))
          ⟹  #(dididit dah)
```

(`vector-fill!` *vector fill*) library procedure

Stores *fill* in every element of *vector*. The value returned by `vector-fill!` is unspecified.

6.4. Control features

This chapter describes various primitive procedures which control the flow of program execution in special ways. The `procedure?` predicate is also described here.

(`procedure?` *obj*) procedure

Returns `#t` if *obj* is a procedure, otherwise returns `#f`.

```
(procedure? car)             ⟹  #t
(procedure? 'car)            ⟹  #f
(procedure? (lambda (x) (* x x)))
                             ⟹  #t
(procedure? '(lambda (x) (* x x)))
                             ⟹  #f
(call-with-current-continuation procedure?)
                             ⟹  #t
```

(apply *proc* arg_1 … *args*) procedure

Proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list (append (list arg_1 …) *args*) as the actual arguments.

```
(apply + (list 3 4))           ⟹  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)       ⟹  30
```

(map *proc* $list_1$ $list_2$ …) library procedure

The *lists* must be lists, and *proc* must be a procedure taking as many arguments as there are *lists* and returning a single value. If more than one *list* is given, then they must all be the same length. Map applies *proc* element-wise to the elements of the *lists* and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *lists* is unspecified.

```
(map cadr '((a b) (d e) (g h)))
          ⟹  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
          ⟹  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))      ⟹  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))                ⟹  (1 2) or (2 1)
```

(for-each *proc* $list_1$ $list_2$ …) library procedure

The arguments to for-each are like the arguments to map, but for-each calls *proc* for its side effects rather than for its values. Unlike map, for-each is guaranteed to call *proc* on the elements of the *lists* in order from the first element(s) to the last, and the value returned by for-each is unspecified.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                           ⟹  #(0 1 4 9 16)
```

(force *promise*) library procedure

Forces the value of *promise* (see delay, section 4.2.5). If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned.

```
(force (delay (+ 1 2)))        ⟹  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                               ⟹  (3 3)

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail a-stream)))
                               ⟹  2
```

Force and delay are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                              ⟹  a promise
(force p)                      ⟹  6
p                              ⟹  a promise, still
(begin (set! x 10)
       (force p))              ⟹  6
```

Here is a possible implementation of delay and force. Promises are implemented here as procedures of no arguments, and force simply calls its argument:

```
(define force
  (lambda (object)
    (object)))
```

We define the expression

```
(delay ⟨expression⟩)
```

to have the same meaning as the procedure call

```
(make-promise (lambda () ⟨expression⟩))
```

as follows

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (make-promise (lambda () expression))))),
```

where make-promise is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result))))))))
```

Rationale: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of make-promise.

Various extensions to this semantics of delay and force are supported in some implementations:

- Calling force on an object that is not a promise may simply return the object.
- It may be the case that there is no means by which a promise can be operationally distinguished from its forced value. That is, expressions like the following may evaluate to either #t or to #f, depending on the implementation:

  ```
  (eqv? (delay 1) 1)            ⟹  unspecified
  (pair? (delay (cons 1 2)))    ⟹  unspecified
  ```

- Some implementations may implement "implicit forcing," where the value of a promise is forced by primitive procedures like cdr and +:

  ```
  (+ (delay (* 3 7)) 13)        ⟹  34
  ```

(call-with-current-continuation *proc*)   procedure

Proc must be a procedure of one argument. The procedure call-with-current-continuation packages up the current continuation (see the rationale below) as an "escape procedure" and passes it as an argument to *proc*. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. Calling the escape procedure may cause the invocation of *before* and *after* thunks installed using dynamic-wind.

The escape procedure accepts the same number of arguments as the continuation to the original call to call-with-current-continuation. Except for continuations created by the call-with-values procedure, all continuations take exactly one value. The effect of passing no value or more than one value to continuations that were not created by call-with-values is unspecified.

The escape procedure that is passed to *proc* has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

The following examples show only the most common ways in which call-with-current-continuation is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                      ⟹  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))      ⟹  4

(list-length '(a b . c))      ⟹  #f
```

Rationale:

A common use of call-with-current-continuation is for structured, non-local exits from loops or procedure bodies, but in fact call-with-current-continuation is extremely useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a *continuation* wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. Call-with-current-continuation allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [16] invented a general purpose escape operator called the J-operator. John Reynolds [24] described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the `catch` construct could be provided by a procedure instead of by a special syntactic construct, and the name `call-with-current-continuation` was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name `call/cc` instead.

(`values` *obj* ...) procedure

Delivers all of its arguments to its continuation. Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value. `Values` might be defined as follows:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

(`call-with-values` *producer consumer*) procedure

Calls its *producer* argument with no values and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                            ⟹  5

(call-with-values * -)      ⟹  -1
```

(`dynamic-wind` *before thunk after*) procedure

Calls *thunk* without arguments, returning the result(s) of this call. *Before* and *after* are called, also without arguments, as required by the following rules (note that in the absence of calls to continuations captured using `call-with-current-continuation` the three arguments are called once each, in order). *Before* is called whenever execution enters the dynamic extent of the call to *thunk* and *after* is called whenever it exits that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. In Scheme, because of `call-with-current-continuation`, the dynamic extent of a call may not be a single, connected time period. It is defined as follows:

- The dynamic extent is entered when execution of the body of the called procedure begins.
- The dynamic extent is also entered when execution is not within the dynamic extent and a continuation is invoked that was captured (using `call-with-current-continuation`) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *after*s from these two invocations of `dynamic-wind` are both to be called, then the *after* associated with the second (inner) call to `dynamic-wind` is called first.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *before*s from these two invocations of `dynamic-wind` are both to be called, then the *before* associated with the first (outer) call to `dynamic-wind` is called first.

If invoking a continuation requires calling the *before* from one call to `dynamic-wind` and the *after* from another, then the *after* is called first.

The effect of using a captured continuation to enter or exit the dynamic extent of a call to *before* or *after* is undefined.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

          ⟹ (connect talk1 disconnect
               connect talk2 disconnect)
```

6.5. `Eval`

`(eval` *expression environment-specifier*`)` procedure

Evaluates *expression* in the specified environment and returns its value. *Expression* must be a valid Scheme expression represented as data, and *environment-specifier* must be a value returned by one of the three procedures described below. Implementations may extend `eval` to allow non-expression programs (definitions) as the first argument and to allow other values as environments, with the restriction that `eval` is not allowed to create new bindings in the environments associated with `null-environment` or `scheme-report-environment`.

```
(eval '(* 7 3) (scheme-report-environment 5))
                                  ⟹  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                  ⟹  20
```

`(scheme-report-environment` *version*`)` procedure
`(null-environment` *version*`)` procedure

Version must be the exact integer `5`, corresponding to this revision of the Scheme report (the Revised^5 Report on Scheme). `Scheme-report-environment` returns a specifier for an environment that is empty except for all bindings defined in this report that are either required or both optional and supported by the implementation. `Null-environment` returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in this report that are either required or both optional and supported by the implementation.

Other values of *version* can be used to specify environments matching past revisions of this report, but their support is not required. An implementation will signal an error if *version* is neither `5` nor another value supported by the implementation.

The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (for example `car`) is unspecified. Thus the environments specified by `scheme-report-environment` may be immutable.

`(interaction-environment)` optional procedure

This procedure returns a specifier for the environment that contains implementation-defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

6.6. Input and output

6.6.1. Ports

Ports represent input and output devices. To Scheme, an input port is a Scheme object that can deliver characters upon command, while an output port is a Scheme object that can accept characters.

`(call-with-input-file` *string proc*`)` library procedure
`(call-with-output-file` *string proc*`)` library procedure

String should be a string naming a file, and *proc* should be a procedure that accepts one argument. For `call-with-input-file`, the file should already exist; for `call-with-output-file`, the effect is unspecified if the file already exists. These procedures call *proc* with one argument: the port obtained by opening the named file for input or output. If the file cannot be opened, an error is signalled. If *proc* returns, then the port is closed automatically and the value(s) yielded by the *proc* is(are) returned. If *proc* does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

Rationale: Because Scheme's escape procedures have unlimited extent, it is possible to escape from the current continuation but later to escape back in. If implementations were permitted to close the port on any escape from the current continuation, then it would be impossible to write portable code using both `call-with-current-continuation` and `call-with-input-file` or `call-with-output-file`.

`(input-port?` *obj*`)` procedure
`(output-port?` *obj*`)` procedure

Returns `#t` if *obj* is an input port or output port respectively, otherwise returns `#f`.

`(current-input-port)` procedure
`(current-output-port)` procedure

Returns the current default input or output port.

`(with-input-from-file` *string thunk*`)`
optional procedure
`(with-output-to-file` *string thunk*`)`
optional procedure

String should be a string naming a file, and *proc* should be a procedure of no arguments. For `with-input-from-file`, the file should already exist; for `with-output-to-file`, the effect is unspecified if the file already exists. The file is opened for input or output, an input or output port connected to it is made the default value returned by `current-input-port` or `current-output-port` (and is used by `(read)`, `(write` *obj*`)`, and so forth), and the *thunk* is called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. `With-input-from-file` and `with-output-to-file` return(s) the value(s) yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, their behavior is implementation dependent.

`(open-input-file` *filename*`)` procedure

Takes a string naming an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened, an error is signalled.

`(open-output-file` *filename*`)` procedure

Takes a string naming an output file to be created and returns an output port capable of writing characters to a new file by that name. If the file cannot be opened, an error is signalled. If a file with the given name already exists, the effect is unspecified.

`(close-input-port` *port*`)` procedure
`(close-output-port` *port*`)` procedure

Closes the file associated with *port*, rendering the *port* incapable of delivering or accepting characters. These routines have no effect if the file has already been closed. The value returned is unspecified.

6.6.2. Input

`(read)` library procedure
`(read` *port*`)` library procedure

`Read` converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal ⟨datum⟩ (see sections 7.1.2 and 6.3.2). `Read` returns the next object parsable from the given input *port*, updating *port* to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned. The port remains open, and further attempts to read will also return an end of file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The *port* argument may be omitted, in which case it defaults to the value returned by `current-input-port`. It is an error to read from a closed port.

`(read-char)` procedure
`(read-char` *port*`)` procedure

Returns the next character available from the input *port*, updating the *port* to point to the following character. If no more characters are available, an end of file object is returned. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

`(peek-char)` procedure
`(peek-char` *port*`)` procedure

Returns the next character available from the input *port*, *without* updating the *port* to point to the following character. If no more characters are available, an end of file object is returned. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

Note: The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` with the same *port*. The only difference is that the very next call to `read-char` or `peek-char` on that *port* will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

`(eof-object?` *obj*`)` procedure

Returns `#t` if *obj* is an end of file object, otherwise returns `#f`. The precise set of end of file objects will vary among implementations, but in any case no end of file object will ever be an object that can be read in using `read`.

`(char-ready?)` procedure
`(char-ready?` *port*`)` procedure

Returns `#t` if a character is ready on the input *port* and returns `#f` otherwise. If `char-ready` returns `#t` then the next `read-char` operation on the given *port* is guaranteed not to hang. If the *port* is at end of file then `char-ready?` returns `#t`. *Port* may be omitted, in which case it defaults to the value returned by `current-input-port`.

Rationale: `Char-ready?` exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must ensure that characters whose existence has been asserted by `char-ready?` cannot be rubbed out. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

6.6.3. Output

`(write` *obj*`)` library procedure
`(write` *obj port*`)` library procedure

Writes a written representation of *obj* to the given *port*. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the `#\` notation. `Write` returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

`(display` *obj*`)` library procedure
`(display` *obj port*`)` library procedure

Writes a representation of *obj* to the given *port*. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by `write-char` instead of by `write`. `Display` returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

Rationale: `Write` is intended for producing machine-readable output and `display` is for producing human-readable output. Implementations that allow "slashification" within symbols will probably want `write` but not `display` to slashify funny characters in symbols.

`(newline)` library procedure
`(newline` *port*`)` library procedure

Writes an end of line to *port*. Exactly how this is done differs from one operating system to another. Returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

`(write-char` *char*`)` procedure
`(write-char` *char port*`)` procedure

Writes the character *char* (not an external representation of the character) to the given *port* and returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `current-output-port`.

6.6.4. System interface

Questions of system interface generally fall outside of the domain of this report. However, the following operations are important enough to deserve description here.

`(load` *filename*`)` optional procedure

Filename should be a string naming an existing file containing Scheme source code. The `load` procedure reads expressions and definitions from the file and evaluates them sequentially. It is unspecified whether the results of the expressions are printed. The `load` procedure does not affect the values returned by `current-input-port` and `current-output-port`. `Load` returns an unspecified value.

Rationale: For portability, `load` must operate on source files. Its operation on other kinds of files necessarily varies among implementations.

`(transcript-on` *filename*`)` optional procedure
`(transcript-off)` optional procedure

Filename must be a string naming an output file to be created. The effect of `transcript-on` is to open the named file for output, and to cause a transcript of subsequent interaction between the user and the Scheme system to be written to the file. The transcript is ended by a call to `transcript-off`, which closes the transcript file. Only one transcript may be in progress at any time, though some implementations may relax this restriction. The values returned by these procedures are unspecified.

7. Formal syntax and semantics

This chapter provides formal descriptions of what has already been described informally in previous chapters of this report.

7.1. Formal syntax

This section provides a formal syntax for Scheme written in an extended BNF.

All spaces in the grammar are for legibility. Case is insignificant; for example, `#x1A` and `#X1a` are equivalent. ⟨empty⟩ stands for the empty string.

The following extensions to BNF are used to make the description more concise: ⟨thing⟩* means zero or more occurrences of ⟨thing⟩; and ⟨thing⟩$^+$ means at least one ⟨thing⟩.

7.1.1. Lexical structure

This section describes how individual tokens (identifiers, numbers, etc.) are formed from sequences of characters. The following sections describe how expressions and programs are formed from sequences of tokens.

⟨Intertoken space⟩ may occur on either side of any token, but not within a token.

Tokens which require implicit termination (identifiers, numbers, characters, and dot) may be terminated by any ⟨delimiter⟩, but not necessarily by anything else.

The following five characters are reserved for future extensions to the language: `[ ] { } |`

⟨token⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
  | ⟨character⟩ | ⟨string⟩
  | `(` | `)` | `#(` | `'` | `` ` `` | `,` | `,@` | `.`
⟨delimiter⟩ ⟶ ⟨whitespace⟩ | `(` | `)` | `"` | `;`
⟨whitespace⟩ ⟶ ⟨space or newline⟩
⟨comment⟩ ⟶ `;` ⟨all subsequent characters up to a line break⟩
⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩
⟨intertoken space⟩ ⟶ ⟨atmosphere⟩*

⟨identifier⟩ ⟶ ⟨initial⟩ ⟨subsequent⟩*
  | ⟨peculiar identifier⟩
⟨initial⟩ ⟶ ⟨letter⟩ | ⟨special initial⟩
⟨letter⟩ ⟶ `a` | `b` | `c` | `...` | `z`

⟨special initial⟩ ⟶ `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=`
  | `>` | `?` | `^` | `_` | `~`
⟨subsequent⟩ ⟶ ⟨initial⟩ | ⟨digit⟩
  | ⟨special subsequent⟩
⟨digit⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨special subsequent⟩ ⟶ `+` | `-` | `.` | `@`
⟨peculiar identifier⟩ ⟶ `+` | `-` | `...`

⟨syntactic keyword⟩ ⟶ ⟨expression keyword⟩
  | `else` | `=>` | `define`
  | `unquote` | `unquote-splicing`
⟨expression keyword⟩ ⟶ `quote` | `lambda` | `if`
  | `set!` | `begin` | `cond` | `and` | `or` | `case`
  | `let` | `let*` | `letrec` | `do` | `delay`
  | `quasiquote`

⟨variable⟩ ⟶ ⟨any ⟨identifier⟩ that isn't
  also a ⟨syntactic keyword⟩⟩

⟨boolean⟩ ⟶ `#t` | `#f`
⟨character⟩ ⟶ `#\` ⟨any character⟩
  | `#\` ⟨character name⟩
⟨character name⟩ ⟶ `space` | `newline`

⟨string⟩ ⟶ `"` ⟨string element⟩* `"`
⟨string element⟩ ⟶ ⟨any character other than `"` or `\`⟩
  | `\"` | `\\`

⟨number⟩ ⟶ ⟨num 2⟩| ⟨num 8⟩
  | ⟨num 10⟩| ⟨num 16⟩

The following rules for ⟨num R⟩, ⟨complex R⟩, ⟨real R⟩, ⟨ureal R⟩, ⟨uinteger R⟩, and ⟨prefix R⟩ should be replicated for $R = 2, 8, 10,$ and 16. There are no rules for ⟨decimal 2⟩, ⟨decimal 8⟩, and ⟨decimal 16⟩, which means that numbers containing decimal points or exponents must be in decimal radix.

⟨num R⟩ ⟶ ⟨prefix R⟩ ⟨complex R⟩
⟨complex R⟩ ⟶ ⟨real R⟩ | ⟨real R⟩ `@` ⟨real R⟩
  | ⟨real R⟩ `+` ⟨ureal R⟩ `i` | ⟨real R⟩ `-` ⟨ureal R⟩ `i`
  | ⟨real R⟩ `+ i` | ⟨real R⟩ `- i`
  | `+` ⟨ureal R⟩ `i` | `-` ⟨ureal R⟩ `i` | `+ i` | `- i`
⟨real R⟩ ⟶ ⟨sign⟩ ⟨ureal R⟩
⟨ureal R⟩ ⟶ ⟨uinteger R⟩
  | ⟨uinteger R⟩ `/` ⟨uinteger R⟩
  | ⟨decimal R⟩
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
  | `.` ⟨digit 10⟩$^+$ `#`* ⟨suffix⟩
  | ⟨digit 10⟩$^+$ `.` ⟨digit 10⟩* `#`* ⟨suffix⟩
  | ⟨digit 10⟩$^+$ `#`$^+$ `.` `#`* ⟨suffix⟩
⟨uinteger R⟩ ⟶ ⟨digit R⟩$^+$ `#`*
⟨prefix R⟩ ⟶ ⟨radix R⟩ ⟨exactness⟩
  | ⟨exactness⟩ ⟨radix R⟩

⟨suffix⟩ ⟶ ⟨empty⟩
  | ⟨exponent marker⟩ ⟨sign⟩ ⟨digit 10⟩$^+$
⟨exponent marker⟩ ⟶ `e` | `s` | `f` | `d` | `l`
⟨sign⟩ ⟶ ⟨empty⟩ | `+` | `-`
⟨exactness⟩ ⟶ ⟨empty⟩ | `#i` | `#e`
⟨radix 2⟩ ⟶ `#b`
⟨radix 8⟩ ⟶ `#o`
⟨radix 10⟩ ⟶ ⟨empty⟩ | `#d`

```
⟨radix 16⟩ ⟶ #x
⟨digit 2⟩ ⟶ 0 | 1
⟨digit 8⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨digit 10⟩ | a | b | c | d | e | f
```

7.1.2. External representations

⟨Datum⟩ is what the read procedure (section 6.6.2) successfully parses. Note that any string that parses as an ⟨expression⟩ will also parse as a ⟨datum⟩.

```
⟨datum⟩ ⟶ ⟨simple datum⟩ | ⟨compound datum⟩
⟨simple datum⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
     | ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩
⟨symbol⟩ ⟶ ⟨identifier⟩
⟨compound datum⟩ ⟶ ⟨list⟩ | ⟨vector⟩
⟨list⟩ ⟶ (⟨datum⟩*) | (⟨datum⟩+ . ⟨datum⟩)
     | ⟨abbreviation⟩
⟨abbreviation⟩ ⟶ ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ ⟶ ' | ` | , | ,@
⟨vector⟩ ⟶ #(⟨datum⟩*)
```

7.1.3. Expressions

```
⟨expression⟩ ⟶ ⟨variable⟩
     | ⟨literal⟩
     | ⟨procedure call⟩
     | ⟨lambda expression⟩
     | ⟨conditional⟩
     | ⟨assignment⟩
     | ⟨derived expression⟩
     | ⟨macro use⟩
     | ⟨macro block⟩

⟨literal⟩ ⟶ ⟨quotation⟩ | ⟨self-evaluating⟩
⟨self-evaluating⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
     | ⟨character⟩ | ⟨string⟩
⟨quotation⟩ ⟶ '⟨datum⟩ | (quote ⟨datum⟩)
⟨procedure call⟩ ⟶ (⟨operator⟩ ⟨operand⟩*)
⟨operator⟩ ⟶ ⟨expression⟩
⟨operand⟩ ⟶ ⟨expression⟩

⟨lambda expression⟩ ⟶ (lambda ⟨formals⟩ ⟨body⟩)
⟨formals⟩ ⟶ (⟨variable⟩*) | ⟨variable⟩
     | (⟨variable⟩+ . ⟨variable⟩)
⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩
⟨command⟩ ⟶ ⟨expression⟩

⟨conditional⟩ ⟶ (if ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩)
⟨test⟩ ⟶ ⟨expression⟩
⟨consequent⟩ ⟶ ⟨expression⟩
⟨alternate⟩ ⟶ ⟨expression⟩ | ⟨empty⟩

⟨assignment⟩ ⟶ (set! ⟨variable⟩ ⟨expression⟩)

⟨derived expression⟩ ⟶
       (cond ⟨cond clause⟩+)
     | (cond ⟨cond clause⟩* (else ⟨sequence⟩))
     | (case ⟨expression⟩
         ⟨case clause⟩+)
     | (case ⟨expression⟩
         ⟨case clause⟩*
         (else ⟨sequence⟩))
     | (and ⟨test⟩*)
     | (or ⟨test⟩*)
     | (let (⟨binding spec⟩*) ⟨body⟩)
     | (let ⟨variable⟩ (⟨binding spec⟩*) ⟨body⟩)
     | (let* (⟨binding spec⟩*) ⟨body⟩)
     | (letrec (⟨binding spec⟩*) ⟨body⟩)
     | (begin ⟨sequence⟩)
     | (do (⟨iteration spec⟩*)
           (⟨test⟩ ⟨do result⟩)
         ⟨command⟩*)
     | (delay ⟨expression⟩)
     | ⟨quasiquotation⟩

⟨cond clause⟩ ⟶ (⟨test⟩ ⟨sequence⟩)
     | (⟨test⟩)
     | (⟨test⟩ => ⟨recipient⟩)
⟨recipient⟩ ⟶ ⟨expression⟩
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨sequence⟩)
⟨binding spec⟩ ⟶ (⟨variable⟩ ⟨expression⟩)
⟨iteration spec⟩ ⟶ (⟨variable⟩ ⟨init⟩ ⟨step⟩)
     | (⟨variable⟩ ⟨init⟩)
⟨init⟩ ⟶ ⟨expression⟩
⟨step⟩ ⟶ ⟨expression⟩
⟨do result⟩ ⟶ ⟨sequence⟩ | ⟨empty⟩

⟨macro use⟩ ⟶ (⟨keyword⟩ ⟨datum⟩*)
⟨keyword⟩ ⟶ ⟨identifier⟩

⟨macro block⟩ ⟶
     (let-syntax (⟨syntax spec⟩*) ⟨body⟩)
     | (letrec-syntax (⟨syntax spec⟩*) ⟨body⟩)
⟨syntax spec⟩ ⟶ (⟨keyword⟩ ⟨transformer spec⟩)
```

7.1.4. Quasiquotations

The following grammar for quasiquote expressions is not context-free. It is presented as a recipe for generating an infinite number of production rules. Imagine a copy of the following rules for $D = 1, 2, 3, \ldots$. D keeps track of the nesting depth.

```
⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨qq template 0⟩ ⟶ ⟨expression⟩
```

⟨quasiquotation D⟩ ⟶ `⟨qq template D⟩
  | `(quasiquote` ⟨qq template D⟩`)`
⟨qq template D⟩ ⟶ ⟨simple datum⟩
  | ⟨list qq template D⟩
  | ⟨vector qq template D⟩
  | ⟨unquotation D⟩
⟨list qq template D⟩ ⟶ `(`⟨qq template or splice D⟩`*)`
  | `(`⟨qq template or splice D⟩$^+$ `.` ⟨qq template D⟩`)`
  | `'`⟨qq template D⟩
  | ⟨quasiquotation $D+1$⟩
⟨vector qq template D⟩ ⟶ `#(`⟨qq template or splice D⟩`*)`
⟨unquotation D⟩ ⟶ `,`⟨qq template $D-1$⟩
  | `(unquote` ⟨qq template $D-1$⟩`)`
⟨qq template or splice D⟩ ⟶ ⟨qq template D⟩
  | ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶ `,@`⟨qq template $D-1$⟩
  | `(unquote-splicing` ⟨qq template $D-1$⟩`)`

In ⟨quasiquotation⟩s, a ⟨list qq template D⟩ can sometimes be confused with either an ⟨unquotation D⟩ or a ⟨splicing unquotation D⟩. The interpretation as an ⟨unquotation⟩ or ⟨splicing unquotation D⟩ takes precedence.

7.1.5. Transformers

⟨transformer spec⟩ ⟶
  `(syntax-rules (`⟨identifier⟩`*)` ⟨syntax rule⟩`*)`
⟨syntax rule⟩ ⟶ `(`⟨pattern⟩ ⟨template⟩`)`
⟨pattern⟩ ⟶ ⟨pattern identifier⟩
  | `(`⟨pattern⟩`*)`
  | `(`⟨pattern⟩$^+$ `.` ⟨pattern⟩`)`
  | `(`⟨pattern⟩`*` ⟨pattern⟩ ⟨ellipsis⟩`)`
  | `#(`⟨pattern⟩`*)`
  | `#(`⟨pattern⟩`*` ⟨pattern⟩ ⟨ellipsis⟩`)`
  | ⟨pattern datum⟩
⟨pattern datum⟩ ⟶ ⟨string⟩
  | ⟨character⟩
  | ⟨boolean⟩
  | ⟨number⟩
⟨template⟩ ⟶ ⟨pattern identifier⟩
  | `(`⟨template element⟩`*)`
  | `(`⟨template element⟩$^+$ `.` ⟨template⟩`)`
  | `#(`⟨template element⟩`*)`
  | ⟨template datum⟩
⟨template element⟩ ⟶ ⟨template⟩
  | ⟨template⟩ ⟨ellipsis⟩
⟨template datum⟩ ⟶ ⟨pattern datum⟩
⟨pattern identifier⟩ ⟶ ⟨any identifier except `...`⟩
⟨ellipsis⟩ ⟶ ⟨the identifier `...`⟩

7.1.6. Programs and definitions

⟨program⟩ ⟶ ⟨command or definition⟩*

⟨command or definition⟩ ⟶ ⟨command⟩
  | ⟨definition⟩
  | ⟨syntax definition⟩
  | `(begin` ⟨command or definition⟩$^+$`)`
⟨definition⟩ ⟶ `(define` ⟨variable⟩ ⟨expression⟩`)`
  | `(define (`⟨variable⟩ ⟨def formals⟩`)` ⟨body⟩`)`
  | `(begin` ⟨definition⟩`*)`
⟨def formals⟩ ⟶ ⟨variable⟩*
  | ⟨variable⟩* `.` ⟨variable⟩
⟨syntax definition⟩ ⟶
  `(define-syntax` ⟨keyword⟩ ⟨transformer spec⟩`)`

7.2. Formal semantics

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [29]; the notation is summarized below:

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth member of the sequence s (1-based)
$\#s$	length of sequence s
$s \S t$	concatenation of sequences s and t
$s \dagger k$	drop the first k members of sequence s
$t \rightarrow a, b$	McCarthy conditional "if t then a else b"
$\rho[x/i]$	substitution "ρ with x for i"
x in $\mathtt{D}$	injection of x into domain $\mathtt{D}$
$x \mid \mathtt{D}$	projection of x to domain $\mathtt{D}$

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of procedure calls and multiple return values.

The boolean flag associated with pairs, vectors, and strings will be true for mutable objects and false for immutable objects.

The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations *permute* and *unpermute*, which must be inverses, to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

The storage allocator *new* is implementation-dependent, but it must obey the following axiom: if $\mathit{new}\,\sigma \in \mathtt{L}$, then $\sigma\,(\mathit{new}\,\sigma \mid \mathtt{L}) \downarrow 2 = \mathit{false}$.

The definition of $\mathcal{K}$ is omitted because an accurate definition of $\mathcal{K}$ would complicate the semantics without being very interesting.

If P is a program in which all variables are defined before being referenced or assigned, then the meaning of P is

$$\mathcal{E}[\![\texttt{((lambda (I*) P') }\langle\text{undefined}\rangle\texttt{ ...)}]\!]$$

where I^* is the sequence of variables defined in P, P′ is the sequence of expressions obtained by replacing every definition in P by an assignment, ⟨undefined⟩ is an expression that evaluates to *undefined*, and $\mathcal{E}$ is the semantic function that assigns meaning to expressions.

7.2.1. Abstract syntax

$\mathrm{K} \in \mathrm{Con}$	constants, including quotations
$\mathrm{I} \in \mathrm{Ide}$	identifiers (variables)
$\mathrm{E} \in \mathrm{Exp}$	expressions
$\Gamma \in \mathrm{Com} = \mathrm{Exp}$	commands

$$\begin{array}{lll}
\mathrm{Exp} & \longrightarrow & \mathrm{K} \mid \mathrm{I} \mid \mathtt{(}\mathrm{E}_0\ \mathrm{E}^*\mathtt{)} \\
& \mid & \mathtt{(lambda\ (}\mathrm{I}^*\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)} \\
& \mid & \mathtt{(lambda\ (}\mathrm{I}^*\ \mathtt{.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)} \\
& \mid & \mathtt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0\mathtt{)} \\
& \mid & \mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\mathtt{)} \mid \mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\mathtt{)} \\
& \mid & \mathtt{(set!}\ \mathrm{I}\ \mathrm{E}\mathtt{)}
\end{array}$$

7.2.2. Domain equations

$\alpha \in \mathtt{L}$		locations
$\nu \in \mathtt{N}$		natural numbers
$\mathtt{T}$	$= \{\mathit{false},\ \mathit{true}\}$	booleans
$\mathtt{Q}$		symbols
$\mathtt{H}$		characters
$\mathtt{R}$		numbers
$\mathtt{E}_\mathrm{p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	pairs
$\mathtt{E}_\mathrm{v}$	$= \mathtt{L}^* \times \mathtt{T}$	vectors
$\mathtt{E}_\mathrm{s}$	$= \mathtt{L}^* \times \mathtt{T}$	strings
$\mathtt{M}$	$= \{\mathit{false},\ \mathit{true},\ \mathit{null},\ \mathit{undefined},\ \mathit{unspecified}\}$	miscellaneous
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	procedure values
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E}_\mathrm{p} + \mathtt{E}_\mathrm{v} + \mathtt{E}_\mathrm{s} + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	stores
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	environments
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	command continuations
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	expression continuations
$\mathtt{A}$		answers
$\mathtt{X}$		errors

7.2.3. Semantic functions

$$\begin{array}{rl}
\mathcal{K} : & \mathrm{Con} \to \mathtt{E} \\
\mathcal{E} : & \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{E}^* : & \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{C} : & \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}
\end{array}$$

Definition of $\mathcal{K}$ deliberately omitted.

$$\mathcal{E}\llbracket\mathrm{K}\rrbracket = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}\llbracket\mathrm{K}\rrbracket)\,\kappa$$

$$\begin{array}{l}
\mathcal{E}\llbracket\mathrm{I}\rrbracket = \lambda\rho\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\
\qquad\qquad\qquad (\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \to \\
\qquad\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“undefined variable”}, \\
\qquad\qquad\qquad\qquad\quad \mathit{send}\,\epsilon\,\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}\llbracket\mathtt{(}\mathrm{E}_0\ \mathrm{E}^*\mathtt{)}\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle\mathrm{E}_0\rangle\ \S\ \mathrm{E}^*)) \\
\qquad\qquad \rho \\
\qquad\qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa) \\
\qquad\qquad\qquad (\mathit{unpermute}\,\epsilon^*)))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}\llbracket\mathtt{(lambda\ (}\mathrm{I}^*\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\quad \mathit{send}\,(\langle\mathit{new}\,\sigma \mid \mathtt{L}, \\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
\qquad\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}\llbracket\Gamma^*\rrbracket\rho'(\mathcal{E}\llbracket\mathrm{E}_0\rrbracket\rho'\kappa')) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*)) \\
\qquad\qquad\qquad\qquad\qquad \epsilon^*, \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}\rangle \\
\qquad\qquad\quad \text{in } \mathtt{E}) \\
\qquad\qquad \kappa \\
\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad\quad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}$$

$$\begin{array}{l}
\mathcal{E}\llbracket\mathtt{(lambda\ (}\mathrm{I}^*\ \mathtt{.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\quad \mathit{send}\,(\langle\mathit{new}\,\sigma \mid \mathtt{L}, \\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to \\
\qquad\qquad\qquad\qquad \mathit{tievalsrest} \\
\qquad\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}\llbracket\Gamma^*\rrbracket\rho'(\mathcal{E}\llbracket\mathrm{E}_0\rrbracket\rho'\kappa')) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,(\mathrm{I}^*\ \S\ \langle\mathrm{I}\rangle)\,\alpha^*)) \\
\qquad\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\qquad\quad (\#\mathrm{I}^*), \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“too few arguments”}\rangle\ \text{in } \mathtt{E}) \\
\qquad\qquad \kappa \\
\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad\quad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}$$

$$\mathcal{E}\llbracket\mathtt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}\rrbracket = \mathcal{E}\llbracket\mathtt{(lambda\ (.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E}_0\mathtt{)}\rrbracket$$

$$\begin{array}{l}
\mathcal{E}\llbracket\mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\mathtt{)}\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}\llbracket\mathrm{E}_0\rrbracket\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}\llbracket\mathrm{E}_1\rrbracket\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}\llbracket\mathrm{E}_2\rrbracket\rho\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}\llbracket\mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\mathtt{)}\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}\llbracket\mathrm{E}_0\rrbracket\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}\llbracket\mathrm{E}_1\rrbracket\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{send}\ \mathit{unspecified}\,\kappa))
\end{array}$$

Here and elsewhere, any expressed value other than *undefined* may be used in place of *unspecified.*

$$\begin{array}{l}
\mathcal{E}\llbracket\mathtt{(set!}\ \mathrm{I}\ \mathrm{E}\mathtt{)}\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}\llbracket\mathrm{E}\rrbracket\,\rho\,(\mathit{single}(\lambda\epsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \epsilon \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\,\kappa)))
\end{array}$$

$$\mathcal{E}^*\llbracket\,\rrbracket = \lambda\rho\kappa\,.\,\kappa\langle\,\rangle$$

$$\begin{array}{l}
\mathcal{E}^*\llbracket\mathrm{E}_0\ \mathrm{E}^*\rrbracket = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}\llbracket\mathrm{E}_0\rrbracket\,\rho\,(\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*\llbracket\mathrm{E}^*\rrbracket\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle\ \S\ \epsilon^*))))
\end{array}$$

$$\mathcal{C}\llbracket\,\rrbracket = \lambda\rho\theta\,.\,\theta$$

$$\mathcal{C}\llbracket\Gamma_0\ \Gamma^*\rrbracket = \lambda\rho\theta\,.\,\mathcal{E}\llbracket\Gamma_0\rrbracket\,\rho\,(\lambda\epsilon^*\,.\,\mathcal{C}\llbracket\Gamma^*\rrbracket\rho\theta)$$

7.2.4. Auxiliary functions

$$\begin{array}{l}
\mathit{lookup} : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}\\
\mathit{lookup} = \lambda\rho\mathrm{I} \,.\, \rho\mathrm{I}
\end{array}$$

$$\begin{array}{l}
\mathit{extends} : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}\\
\mathit{extends} =\\
\quad \lambda\rho\mathrm{I}^*\alpha^* \,.\, \#\mathrm{I}^* = 0 \to \rho,\\
\qquad\qquad\quad \mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\ (\mathrm{I}^* \dagger 1)\ (\alpha^* \dagger 1)
\end{array}$$

$$\mathit{wrong} : \mathtt{X} \to \mathtt{C} \qquad \text{[implementation-dependent]}$$

$$\begin{array}{l}
\mathit{send} : \mathtt{E} \to \mathtt{K} \to \mathtt{C}\\
\mathit{send} = \lambda\epsilon\kappa \,.\, \kappa\langle\epsilon\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{single} : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}\\
\mathit{single} =\\
\quad \lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),\\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of return values”}
\end{array}$$

$$\mathit{new} : \mathtt{S} \to (\mathtt{L} + \{\mathit{error}\}) \qquad \text{[implementation-dependent]}$$

$$\begin{array}{l}
\mathit{hold} : \mathtt{L} \to \mathtt{K} \to \mathtt{C}\\
\mathit{hold} = \lambda\alpha\kappa\sigma \,.\, \mathit{send}\,(\sigma\alpha \downarrow 1)\kappa\sigma
\end{array}$$

$$\begin{array}{l}
\mathit{assign} : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}\\
\mathit{assign} = \lambda\alpha\epsilon\theta\sigma \,.\, \theta(\mathit{update}\,\alpha\epsilon\sigma)
\end{array}$$

$$\begin{array}{l}
\mathit{update} : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}\\
\mathit{update} = \lambda\alpha\epsilon\sigma \,.\, \sigma[\langle\epsilon, \mathit{true}\rangle/\alpha]
\end{array}$$

$$\begin{array}{l}
\mathit{tievals} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}\\
\mathit{tievals} =\\
\quad \lambda\psi\epsilon^*\sigma \,.\, \#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma,\\
\qquad\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \mathit{tievals}\,(\lambda\alpha^* \,.\, \psi(\langle\mathit{new}\,\sigma \mid \mathtt{L}\rangle \mathbin{\S} \alpha^*))\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\epsilon^* \dagger 1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma),\\
\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma
\end{array}$$

$$\begin{array}{l}
\mathit{tievalsrest} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C}\\
\mathit{tievalsrest} =\\
\quad \lambda\psi\epsilon^*\nu \,.\, \mathit{list}\,(\mathit{dropfirst}\,\epsilon^*\nu)\\
\qquad\qquad\qquad (\mathit{single}(\lambda\epsilon \,.\, \mathit{tievals}\,\psi\ ((\mathit{takefirst}\,\epsilon^*\nu) \mathbin{\S} \langle\epsilon\rangle)))
\end{array}$$

$$\mathit{dropfirst} = \lambda ln \,.\, n = 0 \to l, \mathit{dropfirst}\,(l \dagger 1)(n - 1)$$

$$\mathit{takefirst} = \lambda ln \,.\, n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle \mathbin{\S} (\mathit{takefirst}\,(l \dagger 1)(n - 1))$$

$$\begin{array}{l}
\mathit{truish} : \mathtt{E} \to \mathtt{T}\\
\mathit{truish} = \lambda\epsilon \,.\, \epsilon = \mathit{false} \to \mathit{false}, \mathit{true}
\end{array}$$

$$\mathit{permute} : \mathrm{Exp}^* \to \mathrm{Exp}^* \qquad \text{[implementation-dependent]}$$

$$\mathit{unpermute} : \mathtt{E}^* \to \mathtt{E}^* \qquad \text{[inverse of } \mathit{permute}\text{]}$$

$$\begin{array}{l}
\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{applicate} =\\
\quad \lambda\epsilon\epsilon^*\kappa \,.\, \epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa, \mathit{wrong}\ \text{“bad procedure”}
\end{array}$$

$$\begin{array}{l}
\mathit{onearg} : (\mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})\\
\mathit{onearg} =\\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa,\\
\qquad\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}$$

$$\begin{array}{l}
\mathit{twoarg} : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})\\
\mathit{twoarg} =\\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa,\\
\qquad\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}$$

$$\begin{array}{l}
\mathit{list} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{list} =\\
\quad \lambda\epsilon^*\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa,\\
\qquad\qquad\quad \mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{array}$$

$$\begin{array}{l}
\mathit{cons} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{cons} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \to\\
\qquad\qquad\qquad\qquad\qquad (\lambda\sigma' \,.\, \mathit{new}\,\sigma' \in \mathtt{L} \to\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{send}\,(\langle\mathit{new}\,\sigma \mid \mathtt{L}, \mathit{new}\,\sigma' \mid \mathtt{L}, \mathit{true}\rangle\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } \mathtt{E})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \kappa\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'),\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma')\\
\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})\epsilon_1\sigma),\\
\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma)
\end{array}$$

$$\begin{array}{l}
\mathit{less} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{less} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
\qquad\qquad\qquad\qquad\quad \mathit{send}\,(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“non-numeric argument to }\texttt{<}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{add} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{add} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
\qquad\qquad\qquad\qquad\quad \mathit{send}\,((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R}) \text{ in } \mathtt{E})\kappa,\\
\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“non-numeric argument to }\texttt{+}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{car} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{car} =\\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E}_\mathrm{p} \to \mathit{hold}\,(\epsilon \mid \mathtt{E}_\mathrm{p} \downarrow 1)\kappa,\\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“non-pair argument to }\texttt{car}\text{”})
\end{array}$$

$$\mathit{cdr} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad \text{[similar to } \mathit{car}\text{]}$$

$$\begin{array}{l}
\mathit{setcar} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{setcar} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{E}_\mathrm{p} \to\\
\qquad\qquad\qquad\quad (\epsilon_1 \mid \mathtt{E}_\mathrm{p} \downarrow 3) \to \mathit{assign}\,(\epsilon_1 \mid \mathtt{E}_\mathrm{p} \downarrow 1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \epsilon_2\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\,\kappa),\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“immutable argument to }\texttt{set-car!}\text{”},\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“non-pair argument to }\texttt{set-car!}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{eqv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{eqv} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to\\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad\quad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to\\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad\quad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to\\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad\quad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
\qquad\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad\quad (\epsilon_1 \in \mathtt{E}_\mathrm{p} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{p}) \to\\
\qquad\qquad\qquad\qquad \mathit{send}\,((\lambda p_1 p_2 \,.\, ((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{false})\\
\qquad\qquad\qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E}_\mathrm{p})\\
\qquad\qquad\qquad\qquad\qquad (\epsilon_2 \mid \mathtt{E}_\mathrm{p}))\\
\qquad\qquad\qquad\qquad\quad \kappa,
\end{array}$$

$$\begin{array}{l}
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E}_{\mathrm{v}} \wedge \epsilon_2 \in \mathtt{E}_{\mathrm{v}}) \rightarrow \ldots, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E}_{\mathrm{s}} \wedge \epsilon_2 \in \mathtt{E}_{\mathrm{s}}) \rightarrow \ldots, \\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \rightarrow \\
\qquad\qquad\qquad\quad \mathit{send}\,((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \rightarrow \mathit{true}, \mathit{false}) \\
\qquad\qquad\qquad\qquad\quad \kappa, \\
\qquad\qquad\qquad\quad \mathit{send}\ \mathit{false}\,\kappa)
\end{array}$$

$$\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\
\mathit{apply} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\epsilon_1 \in \mathtt{F} \rightarrow \mathit{valueslist}\,\langle\epsilon_2\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_1\epsilon^*\kappa), \\
\qquad\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument to }\mathtt{apply}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{valueslist} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\
\mathit{valueslist} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E}_{\mathrm{p}} \rightarrow \\
\qquad\qquad\qquad\quad \mathit{cdr}\langle\epsilon\rangle \\
\qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,\mathit{valueslist} \\
\qquad\qquad\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\qquad\qquad\quad (\lambda\epsilon^*\,.\,\mathit{car}\langle\epsilon\rangle(\mathit{single}(\lambda\epsilon\,.\,\kappa(\langle\epsilon\rangle \mathbin{\S} \epsilon^*))))), \\
\qquad\qquad\qquad \epsilon = \mathit{null} \rightarrow \kappa\langle\,\rangle, \\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“non-list argument to }\mathtt{values\text{-}list}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \qquad [\mathtt{call\text{-}with\text{-}current\text{-}continuation}] \\
\mathit{cwcc} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{F} \rightarrow \\
\qquad\qquad\qquad\quad (\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathit{applicate}\,\epsilon \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle\langle\mathit{new}\,\sigma \mid \mathtt{L}, \lambda\epsilon^*\kappa'\,.\,\kappa\epsilon^*\rangle \text{ in } \mathtt{E}\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unspecified} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \sigma), \\
\qquad\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma), \\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“bad procedure argument”})
\end{array}$$

$$\begin{array}{l}
\mathit{values} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\
\mathit{values} = \lambda\epsilon^*\kappa\,.\,\kappa\epsilon^*
\end{array}$$

$$\begin{array}{l}
\mathit{cwv} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \qquad [\mathtt{call\text{-}with\text{-}values}] \\
\mathit{cwv} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\mathit{applicate}\,\epsilon_1\langle\,\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_2\,\epsilon^*))
\end{array}$$

7.3. Derived expression types

This section gives macro definitions for the derived expression types in terms of the primitive expression types (literal, variable, call, `lambda`, `if`, `set!`). See section 6.4 for a possible definition of `delay`.

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))


(define-syntax case
  (syntax-rules (else)
    ((case (key ...)
       clauses ...)
     (let ((atom-key (key ...)))
       (case atom-key clauses ...)))
    ((case key
       (else result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
       ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((case key
       ((atoms ...) result1 result2 ...)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)
         (case key clause clauses ...)))))


(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))


(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))


(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
        tag)
       val ...))))


    (define-syntax let*
      (syntax-rules ()
        ((let* () body1 body2 ...)
         (let () body1 body2 ...))
        ((let* ((name1 val1) (name2 val2) ...)
           body1 body2 ...)
         (let ((name1 val1))
           (let* ((name2 val2) ...)
             body1 body2 ...)))))
```

The following letrec macro uses the symbol <undefined> in place of an expression which returns something that when stored in a location makes it an error to try to obtain the value stored in the location (no such expression is defined in Scheme). A trick is used to generate the temporary names needed to avoid specifying the order in which the values are evaluated. This could also be accomplished by using an auxiliary macro.

```
    (define-syntax letrec
      (syntax-rules ()
        ((letrec ((var1 init1) ...) body ...)
         (letrec "generate_temp_names"
           (var1 ...)
           ()
           ((var1 init1) ...)
           body ...))
        ((letrec "generate_temp_names"
           ()
           (temp1 ...)
           ((var1 init1) ...)
           body ...)
         (let ((var1 <undefined>) ...)
           (let ((temp1 init1) ...)
             (set! var1 temp1)
             ...
             body ...)))
        ((letrec "generate_temp_names"
           (x y ...)
           (temp ...)
           ((var1 init1) ...)
           body ...)
         (letrec "generate_temp_names"
           (y ...)
           (newtemp temp ...)
           ((var1 init1) ...)
           body ...))))


    (define-syntax begin
      (syntax-rules ()
        ((begin exp ...)
         ((lambda () exp ...)))))
```

The following alternative expansion for begin does not make use of the ability to write more than one expression in the body of a lambda expression. In any case, note that these rules apply only if the body of the begin contains no definitions.

```
    (define-syntax begin
      (syntax-rules ()
        ((begin exp)
         exp)
        ((begin exp1 exp2 ...)
         (let ((x exp1))
           (begin exp2 ...)))))
```

The following definition of do uses a trick to expand the variable clauses. As with letrec above, an auxiliary macro would also work. The expression (if #f #f) is used to obtain an unspecific value.

```
    (define-syntax do
      (syntax-rules ()
        ((do ((var init step ...) ...)
             (test expr ...)
             command ...)
         (letrec
           ((loop
             (lambda (var ...)
               (if test
                   (begin
                     (if #f #f)
                     expr ...)
                   (begin
                     command
                     ...
                     (loop (do "step" var step ...)
                           ...))))))
           (loop init ...)))
        ((do "step" x)
         x)
        ((do "step" x y)
         y)))
```

NOTES

Language changes

This section enumerates the changes that have been made to Scheme since the "Revised^4 report" [6] was published.

- The report is now a superset of the IEEE standard for Scheme [13]: implementations that conform to the report will also conform to the standard. This required the following changes:
  - The empty list is now required to count as true.
  - The classification of features as essential or inessential has been removed. There are now three classes of built-in procedures: primitive, library, and optional. The optional procedures are `load`, `with-input-from-file`, `with-output-to-file`, `transcript-on`, `transcript-off`, and `interaction-environment`, and `-` and `/` with more than two arguments. None of these are in the IEEE standard.
  - Programs are allowed to redefine built-in procedures. Doing so will not change the behavior of other built-in procedures.
- *Port* has been added to the list of disjoint types.
- The macro appendix has been removed. High-level macros are now part of the main body of the report. The rewrite rules for derived expressions have been replaced with macro definitions. There are no reserved identifiers.
- `Syntax-rules` now allows vector patterns.
- Multiple-value returns, `eval`, and `dynamic-wind` have been added.
- The calls that are required to be implemented in a properly tail-recursive fashion are defined explicitly.
- '`@`' can be used within identifiers. '`|`' is reserved for possible future extensions.

ADDITIONAL MATERIAL

The Internet Scheme Repository at

`http://www.cs.indiana.edu/scheme-repository/`

contains an extensive Scheme bibliography, as well as papers, programs, implementations, and other material related to Scheme.

EXAMPLE

`Integrate-system` integrates the system

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

The parameter `system-derivative` is a function that takes a system state (a vector of values for the state variables $y_1, \ldots, y_n$) and produces a system derivative (the values $y'_1, \ldots, y'_n$). The parameter `initial-state` provides an initial system state, and `h` is an initial guess for the length of the integration step.

The value returned by `integrate-system` is an infinite stream of system states.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`Runge-Kutta-4` takes a function, `f`, that produces a system derivative from a system state. `Runge-Kutta-4` produces a function that takes a system state and produces a new system state.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3))))))))
```

```
(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))
```

```
(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

`Map-streams` is analogous to `map`: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

The following illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[3] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116.

[4] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[5] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[6] William Clinger and Jonathan Rees, editors. The revised^4 report on the algorithmic language Scheme. In *ACM Lisp Pointers* 4(3), pages 1–55, 1991.

[7] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[8] William Clinger. Proper Tail Recursion and Space Efficiency. To appear in *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, June 1998.

[9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4):295–326, 1993.

[10] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [11].

[11] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[12] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[13] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[14] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[15] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161.

[16] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, February 1965.

[17] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[18] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[19] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[20] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[21] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

[22] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[23] Jonathan Rees and William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[24] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[25] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[26] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[27] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[28] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[29] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[30] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.

Medical Devices: The Therac-25*

Nancy Leveson
University of Washington

1 Introduction

Between June 1985 and January 1987, a computer-controlled radiation therapy machine, called the Therac-25, massively overdosed six people. These accidents have been described as the worst in the 35-year history of medical accelerators [6].

A detailed accident investigation, drawn from publicly available documents, can be found in Leveson and Turner [4]. The following account is taken from this report and includes both the factors involved in the overdoses themselves and the attempts by the users, manufacturers, and governments to deal with them. Because this accident was never officially investigated, some information on the Therac-25 software development, management, and quality control procedures are not available. What is included below has been gleaned from law suits and depositions, government records, and copies of correspondence and other material obtained from the U.S. Food and Drug Administration (FDA), which regulates these devices.

2 Background

Medical linear accelerators (linacs) accelerate electrons to create high-energy beams that can destroy tumors with minimal impact on the surrounding

*This appendix is taken from Nancy Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995. Copyright 1995. All rights reserved.

healthy tissue. Relatively shallow tissue is treated with the accelerated electrons; to reach deeper tissue, the electron beam is converted into X-ray photons.

In the early 1970s, Atomic Energy of Canada Limited (AECL)[1] and a French company called CGR went into business together building linear accelerators. The products of this cooperation were (1) the Therac-6, a 6 million electron volt (MeV) accelerator capable of producing X-rays only and later (2) the Therac-20, a 20 MeV, dual-mode (X-rays or electrons) accelerator. Both were versions of older CGR machines, the Neptune and Sagittaire, respectively, which were augmented with computer control using a DEC PDP-11 minicomputer. We know that some of the old Therac-6 software routines were reused in the Therac-20 and that CGR developed the initial software.

Software functionality was limited in both machines: The computer merely added convenience to the existing hardware, which was capable of standing alone. Industry-standard hardware safety features and interlocks in the underlying machines were retained.

The business relationship between AECL and CGR faltered after the Therac-20 effort. Citing competitive pressures, the two companies did not renew their cooperative agreement when scheduled in 1981.

In the mid-1970s, AECL had developed a radical new "double pass" concept for electron acceleration. A double-pass accelerator needs much less space to develop comparable energy levels because it folds the long physical mechanism required to accelerate the electrons, and it is more economical to produce. Using this double-pass concept, AECL designed the Therac-25, a dual-mode linear accelerator that can deliver either photons at 25 MeV or electrons at various energy levels.

Compared with the Therac-20, the Therac-25 is notably more compact, more versatile, and arguably easier to use. The higher energy takes advantage of the phenomenon of *depth dose*: As the energy increases, the depth in the body at which maximum dose build-up occurs also increases, sparing the tissue above the target area. Economic advantages also come into play for the customer, since only one machine is required for both treatment modalities

[1] AECL was an arms-length entity, called a crown corporation, of the Canadian government. Since the time of the incidents related in this paper, AECL Medical, a division of AECL, was privatized and is now called Theratronics International, Ltd. Currently, the primary business of AECL is the design and installation of nuclear reactors.

(electrons and photons).

Several features of the Therac-25 are important in understanding the accidents. First, like the Therac-6 and the Therac-20, the Therac-25 is controlled by a PDP-11 computer. However, AECL designed the Therac-25 to take advantage of computer control from the outset; they did not build on a stand-alone machine. The Therac-6 and Therac-20 had been designed around machines that already had histories of clinical use without computer control.

In addition, the Therac-25 software has more responsibility for maintaining safety than the software in the previous machines. The Therac-20 has independent protective circuits for monitoring the electron-beam scanning plus mechanical interlocks for policing the machine and ensuring safe operation. The Therac-25 relies more on software for these functions. AECL took advantage of the computer's abilities to control and monitor the hardware and decided not to duplicate all the existing hardware safety mechanisms and interlocks.

Some software for the machines was interrelated or reused. In a letter to a Therac-25 user, the AECL quality assurance manager said, "The same Therac-6 package was used by the AECL software people when they started the Therac-25 software. The Therac-20 and Therac-25 software programs were done independently starting from a common base" [4]. The reuse of Therac-6 design features or modules may explain some of the problematic aspects of the Therac-25 software design. The quality assurance manager was apparently unaware that some Therac-20 routines were also used in the Therac-25; this was discovered after a bug related to one of the Therac-25 accidents was found in the Therac-20 software.

AECL produced the first hardwired prototype of the Therac-25 in 1976, and the completely computer-controlled commercial version was available in late 1982.

Turntable Positioning. The Therac-25 turntable design plays an important role in the accidents. The upper turntable (see Figure 1) rotates accessory equipment into the beam path to produce two therapeutic modes: electron mode and photon mode. A third position (called the field light position) involves no beam at all, but rather is used to facilitate correct positioning of the patient. Because the accessories appropriate to each mode

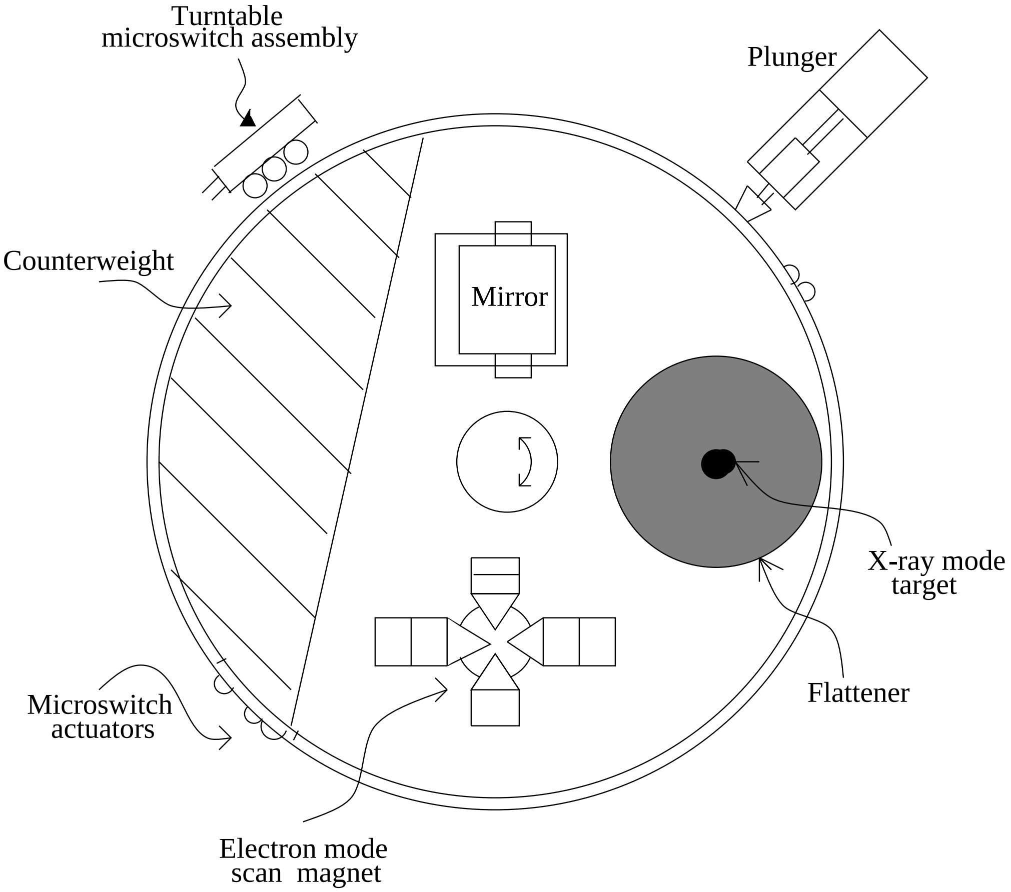


Figure 1: Upper turntable assembly.

are physically attached to the turntable, proper operation of the Therac-25 is heavily dependent on the turntable position, which is monitored by three microswitches.

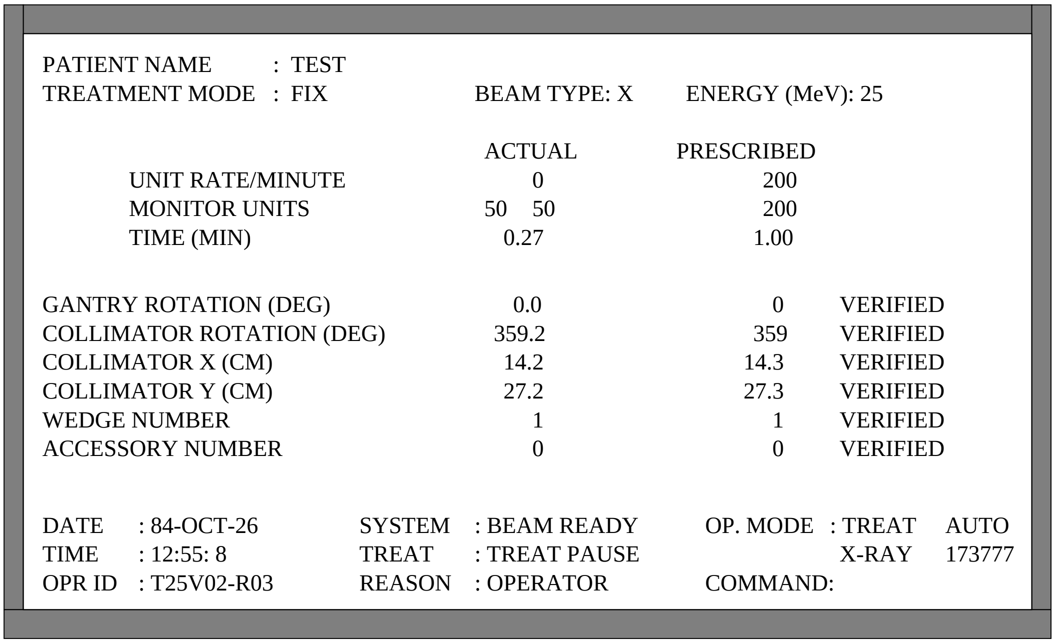
The raw, highly concentrated accelerator beam is dangerous to living tissue. In electron therapy, the computer controls the beam energy (from 5 to 25 MeV) and current, while scanning magnets are used to spread the beam to a safe, therapeutic concentration. These scanning magnets are mounted on the turntable and moved into proper position by the computer. Similarly, an ion chamber to measure electrons is mounted on the turntable and also moved into position by the computer. In addition, operator-mounted electron trimmers can be used to shape the beam if necessary.

For X-ray (or photon) therapy, only one energy level is available: 25 MeV. Much greater electron-beam current is required for X-ray mode (some 100 times greater than that for electron therapy) [6] to produce comparable output. Such a high dose-rate capability is required because a "beam flattener" is used to produce a uniform treatment field. This flattener, which resembles an inverted ice cream cone, is a very efficient attenuator; thus, to get a reasonable treatment dose rate out of the flattener, a very high input dose rate is required. If the machine should produce a photon beam with the beam flattener not in position, a high output dose to the patient results. This is the basic hazard of dual-mode machines: If the turntable is in the wrong position, the beam flattener will not be in place.

In the Therac-25, the computer is responsible for positioning the turntable (and for checking the turntable position) so that a target, flattening filter, and X-ray ion chamber are directly in the beam path. With the target in place, electron bombardment produces X-rays. The X-ray beam is shaped by the flattening filter and measured by the X-ray ion chamber.

No accelerator beam is expected in the third or field light turntable position. A stainless steel mirror is placed in the beam path and a light simulates the beam. This lets the operator see precisely where the beam will strike the patient and make necessary adjustments before treatment starts. There is no ion chamber in place at this turntable position, since no beam is expected.

Traditionally, electromechanical interlocks have been used on these types of equipment to ensure safety — in this case, to ensure that the turntable and attached equipment are in the correct position when treatment is started. In the Therac-25, software checks were substituted for many of the traditional hardware interlocks.



```
PATIENT NAME        :  TEST
TREATMENT MODE   :  FIX                  BEAM TYPE: X       ENERGY (MeV): 25

                                           ACTUAL          PRESCRIBED
        UNIT RATE/MINUTE                      0                  200
        MONITOR UNITS                    50    50                200
        TIME (MIN)                          0.27                1.00

GANTRY ROTATION (DEG)                      0.0                  0     VERIFIED
COLLIMATOR ROTATION (DEG)                359.2                359     VERIFIED
COLLIMATOR X (CM)                         14.2               14.3     VERIFIED
COLLIMATOR Y (CM)                         27.2               27.3     VERIFIED
WEDGE NUMBER                                 1                  1     VERIFIED
ACCESSORY NUMBER                             0                  0     VERIFIED


DATE     : 84-OCT-26          SYSTEM    : BEAM READY        OP. MODE  : TREAT    AUTO
TIME     : 12:55: 8           TREAT     : TREAT PAUSE                   X-RAY    173777
OPR ID   : T25V02-R03         REASON    : OPERATOR          COMMAND:
```

Figure 2: Operator interface screen layout.

The Operator Interface. The description of the operator interface here applies to the version of the software used during the accidents. Changes made as a result of an FDA recall are described later.

The Therac-25 operator controls the machine through a DEC VT100 terminal. In the general case, the operator positions the patient on the treatment table, manually sets the treatment field sizes and gantry rotation, and attaches accessories to the machine. Leaving the treatment room, the operator returns to the console to enter the patient identification, treatment prescription (including mode or beam type, energy level, dose, dose rate, and time), field sizing, gantry rotation, and accessory data. The system then compares the manually set values with those entered at the console. If they match, a *verified* message is displayed and treatment is permitted. If they do not match, treatment is not allowed to proceed until the mismatch is corrected. Figure 2 shows the screen layout.

When the system was first built, operators complained that it took too long to enter the treatment plan. In response, AECL modified the software before the first unit was installed: Instead of reentering the data at the keyboard, operators could simply use a carriage return to copy the treatment site data [5]. A quick series of carriage returns would thus complete the data entry. This modification was to figure in several of the accidents.

The Therac-25 could shut down in two ways after it detected an error condition. One was a *treatment suspend*, which required a complete machine reset to restart. The other, not so serious, was a *treatment pause*, which only required a single key command to restart the machine. If a *treatment pause* occurred, the operator could press the Ⓟ key to "proceed" and resume treatment quickly and conveniently. The previous treatment parameters remained in effect, and no reset was required. This feature could be invoked a maximum of five times before the machine automatically suspended treatment and required the operator to perform a system reset.

Error messages provided to the operator were cryptic, and some merely consisted of the word MALFUNCTION followed by a number from 1 to 64 denoting an analog/digital channel number. According to an FDA memorandum written after one accident:

> The operator's manual supplied with the machine does not explain nor even address the malfunction codes. The Maintance [sic] Manual lists the various malfunction numbers but gives no explanation. The materials provided give no indication that these malfunctions could place a patient at risk.
>
> The program does not advise the operator if a situation exists wherein the ion chambers used to monitor the patient are saturated, thus are beyond the measurement limits of the instrument. This software package does not appear to contain a safety system to prevent parameters being entered and intermixed that would result in excessive radiation being delivered to the patient under treatment.

An operator involved in one of the accidents testified that she had become insensitive to machine malfunctions. Malfunction messages were commonplace and most did not involve patient safety. Service technicians would fix the problems or the hospital physicist would realign the machine and make it operable again. She said,

> "It was not out of the ordinary for something to stop the machine.... It would often give a low dose rate in which you would turn the machine back on. ... They would give messages of low dose rate, V-tilt, H-tilt, and other things; I can't remember all the reasons it would stop, but there was a lot of them."

A radiation therapist at another clinic reported that an average of 40 dose-rate malfunctions, attributed to underdoses, occurred on some days.

The operator further testified that during instruction she had been taught that there were "so many safety mechanisms" that she understood it was virtually impossible to overdose a patient.

Hazard Analysis. In March 1983, AECL performed a safety analysis on the Therac-25. This analysis was in the form of a fault tree and apparently excluded the software. According to the final report, the analysis made several assumptions about the computer and its software:

> 1. Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions on teletherapy units. Any residual software errors are not included in the analysis.
> 2. Program software does not degrade due to wear, fatigue, or reproduction process.
> 3. Computer execution errors are caused by faulty hardware components and by "soft" (random) errors induced by alpha particles and electromagnetic noise.

The fault tree resulting from this analysis does appear to include computer failure, although apparently, judging from the basic assumptions above, it considers hardware failures only. For example, in one OR gate leading to the event of getting the wrong energy, a box contains "Computer selects wrong energy," and a probability of 10^{-11} is assigned to this event. For "Computer selects wrong mode," a probability of 4×10^{-9} is given. The report provides no justification of either number.

3 Events

Eleven Therac-25s were installed: five in the United States and six in Canada. Six accidents occurred between 1985 and 1987, when the machine was finally recalled to make extensive design changes. These changes include adding hardware safeguards against software errors.

Related problems were found in the Therac-20 software, but they were not recognized until after the Therac-25 accidents because the Therac-20 includes hardware safety interlocks. Thus, no injuries resulted.

3.1 Kennestone Regional Oncology Center, June 1985

Details of this accident in Marietta, Georgia, are sketchy because it was never investigated. There was no admission that the injury was caused by the Therac-25 until long after the occurrence, despite claims by the patient that she had been injured during treatment, the obvious and severe radiation burns the patient suffered, and the suspicions of the radiation physicist involved.

After undergoing a lumpectomy to remove a malignant breast tumor, a 61-year-old woman was receiving follow-up radiation treatment to nearby lymph nodes on a Therac-25 at the Kennestone facility in Marietta. The Therac-25 had been operating at Kennestone for about six months; other Therac-25s had been operating, apparently without incident, since 1983.

On June 3, 1985, the patient was set up for a 10 MeV electron treatment to the clavicle area. When the machine turned on, she felt a "tremendous force of heat...this red-hot sensation." When the technician came in, she said, "You burned me." The technician replied that that was impossible. Although there were no marks on the patient at the time, the treatment area felt "warm to the touch."

It is unclear exactly when AECL learned about this incident. Tim Still, the Kennestone physicist, said that he contacted AECL to ask if the Therac-25 could operate in electron mode without scanning to spread the beam. Three days later the engineers at AECL called the physicist back to explain that improper scanning was not possible.

In an August 19, 1986 letter from AECL to the FDA, the AECL quality assurance manager said, "In March of 1986 AECL received a lawsuit from the patient involved...This incident was never reported to AECL prior to this date, although some rather odd questions had been posed by Tim Still, the hospital physicist." The physicist at a hospital in Tyler, Texas, where a later accident occurred, reported, "According to Tim Still, the patient filed suit in October 1985 listing the hospital, manufacturer and service organization responsible for the machine. AECL was notified informally about the suit by the hospital, and AECL received official notification of a law suit in November 1985."

Because of the lawsuit (filed November 13, 1985), some AECL administrators must have known about the Marietta accident—although no investigation occurred at this time. FDA memos point to the lack of a mechanism in AECL to follow up reports of suspected accidents [4].

The patient went home, but shortly afterward she developed a reddening and swelling in the center of the treatment area. Her pain had increased to the point that her shoulder "froze," and she experienced spasms. She was admitted to a hospital in Atlanta, but her oncologists continued to send her to Kennestone for Therac-25 treatments. Clinical explanation was sought for the reddening of the skin, which at first her oncologist attributed to her disease or to normal treatment reaction.

About two weeks later, the Kennestone physicist noticed that the patient had a matching reddening on her back as though a burn had gone right through her body, and the swollen area had begun to slough off layers of skin. Her shoulder was immobile, and she was apparently in great pain. It was now obvious that she had a radiation burn, but the hospital and her doctors could provide no satisfactory explanation.

The Kennestone physicist later estimated that the patient received one or two doses of radiation in the 15,000 to 20,000 rad (radiation absorbed dose) range. He did not believe her injury could have been caused by less than 8,000 rads. To understand the magnitude of this, consider that typical single therapeutic doses are in the 200 rad range. Doses of 1,000 rads can be fatal if delivered to the whole body; in fact, 500 rads is the accepted figure for whole-body radiation that will cause death in 50 percent of the cases. The consequences of an overdose to a smaller part of the body depend on the tissue's radio-sensitivity. The director of radiation oncology at the Kennestone facility explained their confusion about the accident as due to the fact that they had never seen an overtreatment of that magnitude before [7].

Eventually, the patient's breast had to be removed because of the radiation burns. Her shoulder and arm were paralyzed, and she was in constant pain. She had suffered a serious radiation burn, but the manufacturer and operators of the machine refused to believe that it could have been caused by the Therac-25. The treatment prescription printout feature of the computer was disabled at the time of the accident, so there was no hardcopy of the treatment data. The lawsuit was eventually settled out of court.

From what we can determine, the accident was not reported to the FDA until *after* further accidents in 1986. The reporting requirements for medical device incidents at that time applied only to equipment manufacturers and importers, not users. The regulations required that manufacturers and importers report deaths, serious injuries, or malfunctions that could result in those consequences, but health-care professionals and institutions were not required to report incidents to manufacturers. The comptroller general of the U.S. Government Accounting Office (GAO), in testimony before Congress on November 6, 1989, expressed great concern about the viability of the incident-reporting regulations in preventing or spotting medical device problems. According to a 1990 GAO study, the FDA knew of less than 1 percent of deaths, serious injuries, or equipment malfunctions that occurred in hospitals [2]. The law was amended in 1990 to require health-care facilities to report incidents to the manufacturer and to the FDA.

At this point, the other Therac-25 users were also unaware that anything untoward had occurred and did not learn about any problems with the machine until after subsequent accidents. Even then, most of their information came through personal communication among themselves.

3.2 Ontario Cancer Foundation, July 1985

The second in this series of accidents occurred about seven weeks after the Kennestone patient was overdosed. At that time, the Therac-25 at the Ontario Cancer Foundation in Hamilton, Ontario (Canada), had been in use for more than six months. On July 26, 1985, a forty-year-old patient came to the clinic for her twenty-fourth Therac-25 treatment for carcinoma of the cervix. The operator activated the machine, but the Therac shut down after five seconds with an HTILT error message. The Therac-25's console display read NO DOSE and indicated a TREATMENT PAUSE.

Since the machine did not suspend and the control display indicated no dose was delivered to the patient, the operator went ahead with a second attempt at treatment by pressing the Ⓟ key (the *proceed* command), expecting the machine to deliver the proper dose this time. This was standard operating procedure, and Therac-25 operators had become accustomed to frequent malfunctions that had no untoward consequences for the patient. Again, the machine shut down in the same manner. The operator repeated this process four times after the original attempt—the display showing NO DOSE delivered to the patient each time. After the fifth pause, the machine went into treatment suspend, and a hospital service technician was called. The technician found nothing wrong with the machine. According to a Therac-25 operator, this scenario also was not unusual.

After the treatment, the patient complained of a burning sensation, described as an "electric tingling shock" to the treatment area in her hip. Six other patients were treated later that day without incident. She came back for further treatment on July 29 and complained of burning, hip pain, and excessive swelling in the region of treatment. The patient was hospitalized for the condition on July 30, and the machine was taken out of service.

AECL was informed of the apparent radiation injury and sent a service engineer to investigate. The U.S. FDA, the then Canadian Radiation Protection Bureau (RPB),[2] and users were informed that there was a problem, although the users claim that they were never informed that a patient injury had occurred. Users were told that they should visually confirm the proper turntable alignment until further notice (which occurred three months later).

The patient died on November 3, 1985, of an extremely virulent cancer. An autopsy revealed the cause of death as the cancer, but it was noted that had she not died, a total hip replacement would have been necessary as a result of the radiation overexposure. An AECL technician later estimated the patient had received between 13,000 and 17,000 rads.

3.2.1 Manufacturer's Response

AECL could not reproduce the malfunction that had occurred, but suspected a transient failure in the microswitch used to determine the turntable position. During the investigation of the accident, AECL hardwired the error conditions they assumed were necessary for the malfunction and, as a result, found some turntable positioning design weaknesses and potential mechanical problems.

[2] On April 1, 1986, the Radiation Protection Bureau and the Bureau of Medical Devices were merged to form the Bureau of Radiation and Medical Devices (BRMD).

The computer senses and controls turntable position by reading a 3-bit signal about the status of three microswitches in the turntable switch assembly. Essentially, AECL determined that a 1-bit error in the microswitch codes (which could be caused by a single open-circuit fault on the switch lines) could produce an ambiguous position message to the computer. The problem was exacerbated by the design of the mechanism that extends a plunger to lock the turntable when it is in one of the three cardinal positions: The plunger could be extended when the turntable was way out of position, thus giving a second false position indication. AECL devised a method to indicate turntable position that tolerated a 1-bit error so that the code would still unambiguously reveal correct position with any one microswitch failure.

In addition, AECL altered the software so that the computer checked for "in transit" status of the switches to keep further track of the switch operation and turntable position and to give additional assurance that the switches were working and the turntable was moving.

As a result of these improvements, AECL claimed in its report and correspondence with hospitals that "analysis of the hazard rate of the new solution indicates an improvement over the old system by at least *5 orders of magnitude* [emphasis added]." However, in its final incident report to the FDA, AECL concluded that they "cannot be firm on the exact cause of the accident but can only suspect . . . ," which underscored their inability to determine the cause of the accident with any certainty. The AECL quality assurance manager testified that they could not reproduce the switch malfunction and that testing of the microswitch was "inconclusive." The similarity of the errant behavior and the patient injuries in this accident and a later one in Yakima, Washington, provide good reason to believe that the Hamilton overdose was probably related to software error rather than to a microswitch failure.

3.2.2 Government and User Response

The Hamilton accident resulted in a voluntary recall by AECL, and the FDA termed it a Class II recall. Class II means "a situation in which the use of, or exposure to, a violative product may cause temporary or medically reversible adverse health consequences or where the probability of serious adverse health consequences is remote." The FDA audited AECL's subsequent modifications, and after the modifications were made, the users were told they could return to normal operating procedures.

As a result of the Hamilton accident, the head of advanced X-ray systems in the Canadian RPB, Gordon Symonds, wrote a report that analyzed the design and performance characteristics of the Therac-25 with respect to radiation safety. Besides citing the flawed microswitch, the report faulted both hardware and software components of the Therac's design. It concluded with a list of four modifications to the Therac-25 necessary for compliance with Canada's Radiation Emitting Devices (RED) Act. The RED law, enacted in 1971, gives government officials power to ensure the safety of radiation-emitting devices.

The modifications specified in the Symonds report included redesigning the microswitch and changing the way the computer handled malfunction conditions. In particular, treatment was to be terminated in the event of a dose-rate malfunction, giving a treatment "suspend." This change would have removed the option to proceed simply by pressing the Ⓟ key. The report also made recommendations regarding collimator test procedures and message and command formats. A November 8, 1985 letter, signed by the director of the Canadian RPB, asked that AECL make changes to the Therac-25 based on the Symond's report "to be in compliance with the RED act."

Although, as noted above, AECL did make the microswitch changes, they did not comply with the directive to change the malfunction pause behavior into treatment suspends, instead reducing the maximum number of retries from five to three. According to Symonds, the deficiencies outlined in the RPB letter of November 8 were still pending when the next accident happened five months later.

Immediately after the Hamilton accident, the Ontario Cancer Foundation hired an independent consultant to investigate. He concluded in a September 1985 report that an independent system (beside the computer) was needed to verify the turntable position and suggested the use of a potentiometer. The RPB wrote a letter to AECL in November 1985 requesting that AECL install such an independent interlock on the Therac-25. Also, in January 1986, AECL received a letter from the attorney representing the Hamilton clinic. The letter said that there had been continuing problems with the turntable, including four incidents at Hamilton, and requested the installation of an independent system (potentiometer) to verify the turntable position. AECL did not comply: No independent interlock was installed by AECL on the Therac-25s at this time. The Hamilton Clinic, however, decided to install one themselves on their machine.

3.3 Yakima Valley Memorial Hospital, December 1985

In this accident, as in the Kennestone overdose, machine malfunction was not acknowledged until after later accidents were understood.

The Therac-25 at Yakima, Washington, had been modified by AECL in September 1985 in response to the overdose at Hamilton. During December 1985, a woman treated with the Therac-25 developed erythema (excessive reddening of the skin) in a parallel striped pattern on her right hip. Despite this, she continued to be treated by the Therac-25, as the cause of her reaction was not determined to be abnormal until January 1986. On January 6, her treatments were completed.

The staff monitored the skin reaction closely and attempted to find possible causes. The open slots in the blocking trays in the Therac-25 could have produced such a striped pattern, but by the time the skin reaction was determined to be abnormal, the blocking trays had been discarded, so the blocking arrangement and tray striping orientation could not be reproduced. A reaction to chemotherapy was ruled out because that should have produced reactions at the other treatment sites and would not have produced stripes. When the doctors discovered that the woman slept with a heating pad, they thought maybe the burn pattern had been caused by the parallel wires that deliver the heat in such pads. The staff X-rayed the heating pad but discovered that the wire pattern did not correspond to the erythema pattern on the patient's hip.

The hospital staff sent a letter to AECL on January 31, and they also spoke on the phone with the AECL technical support supervisor. On February 24, the AECL technical support supervisor sent a written response to the director of radiation therapy at Yakima saying, "After careful consideration we are of the opinion that this damage could not have been produced by any malfunction of the Therac-25 or by any operator error." The letter goes on to support this opinion by listing two pages of technical reasons why an overdose by the Therac-25 was impossible, along with the additional argument that there have "apparently been no other instances of similar damage to this or other patients." The letter ends, "In closing, I wish to advise that this matter has been brought to the attention of our Hazards Committee as is normal practice."

The hospital staff eventually ascribed the patient's skin reaction to "cause unknown." In a report written on this first Yakima incident after another

Yakima overdose a year later, the medical physicist involved wrote:

> At that time, we did not believe that [the patient] was overdosed because the manufacturer had installed additional hardware and software safety devices to the accelerator.
>
> In a letter from the manufacturer dated 16-Sep-85, it is stated that "Analysis of the hazard rate resulting from these modifications indicates an improvement of at least five orders of magnitude"! With such an improvement in safety (10,000,000%) we did not believe that there could have been any accelerator malfunction. These modifications to the accelerator were completed on 5,6-Sep-85.

Even with fairly sophisticated physics support, the hospital staff, as users, did not have the ability to investigate the possibility of machine malfunction further. They were not aware of any other incidents and, in fact, were told that there had been none, so there was no reason for them to pursue the matter. No further investigation of this incident was done by the manufacturer or by any government agencies (who did not know about it).

About a year later (February 1987), after the second Yakima overdose led the hospital staff to suspect that this first injury had been due to a Therac-25 fault, the staff investigated and found that the first overdose victim had a chronic skin ulcer, tissue necrosis (death) under the skin, and was in continual pain. The damage was surgically repaired, skin grafts were made, and the symptoms relieved. The patient is alive today with minor disability and some scarring related to the overdose. The hospital staff concluded that the dose accidentally delivered in the first accident must have been much lower than in the second, as the reaction was significantly less intense and necrosis did not develop until six or eight months after exposure. Some other factors related to the place on the body where the overdose occurred also kept her from having more significant problems.

3.4 East Texas Cancer Center, March 1986

More is known about the Tyler, Texas, accidents than the others because of the diligence of the Tyler hospital physicist, Fritz Hager, without whose efforts the understanding of the software problems may have been delayed even further.

The Therac-25 had been at the East Texas Cancer Center (ETCC) for two years before the first serious accident, and more than 500 patients had been treated. On March 21, 1986, a male patient came into ETCC for his ninth treatment on the Therac-25, one of a series prescribed as followup to the removal of a tumor from his back.

This treatment was to be a 22 MeV electron beam treatment of 180 rads on the upper back and a little to the left of his spine, for a total of 6,000 rads over six and a half weeks. He was taken into the treatment room and placed face down on the treatment table. The operator then left the treatment room, closed the door, and sat at the control terminal.

The operator had held this job for some time, and her typing efficiency had increased with experience. She could quickly enter prescription data and change it conveniently with the Therac's editing features. She entered the patient's prescription data quickly, then noticed that she had typed "x" (for X-ray) when she had intended "e" (for electron) mode. This was a common mistake as most of the treatments involved X-rays, and she had gotten used to typing this. The mistake was easy to fix; she merely used the ⇧ key to edit the mode entry.

Because the other parameters she had entered were correct, she hit the return key several times and left their values unchanged. She reached the bottom of the screen, where it was indicated that the parameters had been VERIFIED and the terminal displayed BEAM READY, as expected. She hit the one-key command, Ⓑ for *beam on*, to begin the treatment. After a moment, the machine shut down and the console displayed the message MALFUNCTION 54. The machine also displayed a TREATMENT PAUSE, indicating a problem of low priority. The sheet on the side of the machine explained that this malfunction was a "dose input 2" error. The ETCC did not have any other information available in its instruction manual or other Therac-25 documentation to explain the meaning of MALFUNCTION 54. An AECL technician later testified that "dose input 2" meant that a dose had been delivered that was either too high or too low. The messages had been expected to be used only during internal company development.

The machine showed a substantial underdose on its dose monitor display—6 monitor units delivered whereas the operator had requested 202 monitor units. She was accustomed to the quirks of the machine, which would frequently stop or delay treatment; in the past, the only consequences had been inconvenience. She immediately took the normal action when the machine merely paused, which was to hit the Ⓟ key to proceed with the treatment. The machine promptly shut down with the same MALFUNCTION 54 error and the same underdose shown by the dosimetry.

The operator was isolated from the patient, since the machine apparatus was inside a shielded room of its own. The only way that the operator could be alerted to patient difficulty was through audio and video monitors. On this day, the video display was unplugged and the audio monitor was broken.

After the first attempt to treat him, the patient said that he felt as if he had received an electric shock or that someone had poured hot coffee on his back: He felt a thump and heat and heard a buzzing sound from the equipment. Since this was his ninth treatment, he knew that this was not normal. He began to get up from the treatment table to go for help. It was at this moment that the operator hit the Ⓟ key to proceed with the treatment. The patient said that he felt like his arm was being shocked by electricity and that his hand was leaving his body. He went to the treatment room door and pounded on it. The operator was shocked and immediately opened the door for him. He appeared visibly shaken and upset.

The patient was immediately examined by a physician, who observed intense reddening of the treatment area, but suspected nothing more serious than electric shock. The patient was discharged and sent home with instructions to return if he suffered any further reactions. The hospital physicist was called in, and he found the machine calibration within specifications. The meaning of the malfunction message was not understood. The machine was then used to treat patients for the rest of the day.

In actuality, but unknown to anyone at that time, the patient had received a massive overdose, concentrated in the center of the treatment location. After-the-fact simulations of the accident revealed possible doses of 16,500 to 25,000 rads in less than 1 second over an area of about 1 cm.

Over the weeks following the accident, the patient continued to have pain in his neck and shoulder. He lost the function of his left arm and had periodic bouts of nausea and vomiting. He was eventually hospitalized for radiation-induced myelitis of the cervical cord causing paralysis of his left arm and both legs, left vocal cord paralysis (which left him unable to speak), neurogenic bowel and bladder, and paralysis of the left diaphragm. He also had a lesion on his left lung and recurrent herpes simplex skin infections. He died from complications of the overdose five months after the accident.

3.4.1 User and Manufacturer Response

The Therac-25 was shut down for testing the day after this accident. One local AECL engineer and one from the home office in Canada came to ETCC to investigate. They spent a day running the machine through tests, but could not reproduce a Malfunction 54. The AECL engineer from the home office reportedly explained that it was not possible for the Therac-25 to overdose a patient. The ETCC physicist claims that he asked AECL at this time if there were any other reports of radiation overexposure and that AECL personnel (including the quality assurance manager) told him that AECL knew of no accidents involving radiation overexposure by the Therac-25. This seems odd since AECL was surely at least aware of the Hamilton accident that had occurred seven months before and the Yakima accident, and, even by their account, learned of the Georgia law suit around this time (which had been filed four months earlier). The AECL engineers then suggested that an electrical problem might have caused the problem.

The electric shock theory was checked out thoroughly by an independent engineering firm. The final report indicated that there was no electrical grounding problem in the machine, and it did not appear capable of giving a patient an electrical shock. The ETCC physicist checked the calibration of the Therac-25 and found it to be satisfactory. He put the machine back into service on April 7, 1986, convinced that it was performing properly.

3.5 East Texas Cancer Center, April 1986

Three weeks later, on April 11, 1986, another male patient was scheduled to receive an electron treatment at ETCC for a skin cancer on the side of his face. The prescription was for 10 MeV. The same technician who had treated the first Tyler accident victim prepared this patient for treatment. Much of what follows is from the operator's deposition.

As with her former patient, she entered the prescription data and then noticed an error in the mode. Again she used the edit ↑ key to change the mode from X-ray to electron. After she finished editing, she pressed the RETURN key several times to place the cursor on the bottom of the screen. She saw the BEAM READY message displayed and turned the beam on.

Within a few seconds the machine shut down, making a loud noise audible via the (now working) intercom. The display showed MALFUNCTION 54 again. The operator rushed into the treatment room, hearing her patient moaning for help. He began to remove the tape that had held his head in position and said something was wrong. She asked him what he felt, and he replied, "fire" on the side of his face. She immediately went to the hospital physicist and told him that another patient appeared to have been burned. Asked by the physicist to described what had happened, the patient explained that something had hit him on the side of the face, he saw a flash of light, and he heard a sizzling sound reminiscent of frying eggs. He was very agitated and asked, "What happened to me, what happened to me?"

This patient died from the overdose on May 1, 1986, three weeks after the accident. He had disorientation, which progressed to coma, fever to 104°F, and neurological damage. An autopsy showed an acute high-dose radiation injury to the right temporal lobe of the brain and the brain stem.

3.5.1 User and Manufacturer Response

After this second Tyler accident, the ETCC physicist immediately took the machine out of service and called AECL to alert them to this second apparent overexposure. The physicist then began a careful investigation of his own. He worked with the operator, who remembered exactly what she had done on this occasion. After a great deal of effort, they were eventually able to elicit the MALFUNCTION 54 message. They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace (as is natural for someone who has repeated the procedure a large number of times), the overdose occurred. It took some practice before the physicist could repeat the procedure rapidly enough to elicit the MALFUNCTION 54 message at will.

The next day, an engineer from AECL called and said that he could not reproduce the error. After the ETCC physicist explained that the procedure had to be performed quite rapidly, AECL could finally produce a similar malfunction on its own machine. Two days after the accident, AECL said it had measured the dosage (at the center of the field) to be 25,000 rads. An AECL engineer explained that the frying sound heard by the patients was the ion chambers being saturated.

In one law suit that resulted from the Tyler accidents, the AECL quality control manager testified that a "cursor up" problem had been found in the service (maintenance) mode at other clinics in February or March of 1985 and also in the summer of 1985. Both times, AECL thought that the software problems had been fixed. There is no way to determine whether there is any relationship between these problems and the Tyler accidents.

3.5.2 Related Therac-20 Problems

The software for both the Therac-25 and Therac-20 "evolved" from the Therac-6 software. Additional functions had to be added because the Therac-20 (and Therac-25) operate in both X-ray and electron mode, while the Therac-6 has only X-ray mode. CGR modified the software for the Therac-20 to handle the dual modes. When the Therac-25 development began, AECL engineers adapted the software from the Therac-6, but they also borrowed software routines from the Therac-20 to handle electron mode, which was allowed under their cooperative agreements.

After the second Tyler, Texas, accident, a physicist at the University of Chicago Joint Center for Radiation Therapy heard about the Therac-25 software problem and decided to find out whether the same thing could happen with the Therac-20. At first, the physicist was unable to reproduce the error on his machine, but two months later he found the link.

The Therac-20 at the University of Chicago is used to teach students in a radiation therapy school conducted by the center. The center's physicist, Frank Borger, noticed that whenever a new class of students started using the Therac-20, fuses and breakers on the machine tripped, shutting down the unit. These failures, which had been occurring ever since the school had acquired the machine, might happen three times a week while new students operated the machine and then disappear for months. Borger determined that new students make many different types of mistakes and use "creative methods" of editing parameters on the console. Through experimentation, he found that certain editing sequences correlated with blown fuses and determined that the same computer bug (as in the Therac-25 software) was responsible. The physicist notified the FDA, which notified Therac-20 users [3].

The software error is just a nuisance on the Therac-20 because this machine has independent hardware protective circuits for monitoring the electron beam scanning. The protective circuits do not allow the beam to turn on, so there is no danger of radiation exposure to a patient. While the Therac-20 relies on mechanical interlocks for monitoring the machine, the

Therac-25 relies largely on software.

3.5.3 The Software "Bug"

A lesson to be learned from the Therac-25 story is that focusing on particular software "bugs" is not the way to make a safe system. Virtually all complex software can be made to behave in an unexpected fashion under some conditions. The basic mistakes here involved poor software engineering practices and building a machine that relies on the software for safe operation. Furthermore, the particular coding error is not as important as the general unsafe design of the software overall. Examining the part of the code blamed for the Tyler accidents is instructive, however, in demonstrating the overall software design flaws. First the software design is described and then the errors believed to be involved in the Tyler accidents and perhaps others.

Therac-25 Software Development and Design. AECL claims proprietary rights to its software design. However, from voluminous documentation regarding the accidents, the repairs, and the eventual design changes, we can build a rough picture of it.

The software is responsible for monitoring the machine status, accepting input about the treatment desired, and setting the machine up for this treatment. It turns the beam on in response to an operator command (assuming that certain operational checks on the status of the physical machine are satisfied) and also turns the beam off when treatment is completed, when an operator commands it, or when a malfunction is detected. The operator can print out hardcopy versions of the CRT display or machine setup parameters.

The treatment unit has an interlock system designed to remove power to the unit when there is a hardware malfunction. The computer monitors this interlock system and provides diagnostic messages. Depending on the fault, the computer either prevents a treatment from being started or, if the treatment is in progress, creates a pause or a suspension of the treatment.

There are two basic operational modes: treatment mode and service mode. Treatment mode controls the normal treatment process. In service mode, the unit can be operated with some of the operational and treatment interlocks bypassed, and additional operational commands and characteristics may be selected. Service mode is entered only through the use of a password at the service keyboard.

The manufacturer describes the Therac-25 software as having a standalone, real-time treatment operating system. The system does not use a standard operating system or executive. Rather, the real-time executive was written especially for the Therac-25 and runs on a 32K PDP-11/23. Cycles are allocated to the critical and noncritical tasks using a preemptive scheduler.

The software, written in PDP-11 assembly language, has four major components: stored data, a scheduler, a set of critical and noncritical tasks, and interrupt services. The stored data includes calibration parameters for the accelerator setup as well as patient-treatment data. The interrupt routines include

- A clock interrupt service routine
- A scanning interrupt service routine
- Traps (for software overflow and computer hardware generated interrupts)
- Power up (initiated at power up to initialize the system and pass control to the scheduler)
- Treatment console screen interrupt handler
- Treatment console keyboard interrupt handler
- Service printer interrupt handler
- Service keyboard interrupt handler

The scheduler controls the sequencing of all noninterrupt events and coordinates all concurrent processes. Tasks are initiated every 0.1 second, with the critical tasks executed first and the noncritical tasks executed in any remaining cycle time. Critical tasks include the following:

- The treatment monitor (Treat) directs and monitors patient setup and treatment via eight operating phases. These are called as subroutines, depending on the value of the Tphase control variable. Following the execution of a particular subroutine, Treat reschedules itself. Treat interacts with the keyboard processing task, which handles operator console communication. The prescription data is cross-checked and verified by other tasks (such as keyboard processor or parameter setup sensor) that inform the treatment task of the verification status via shared variables.

- The servo task controls gun emission, dose rate (pulse repetition frequency), symmetry (beam steering), and machine motions. The servo task also sets up the machine parameters and monitors the beam-tilt-error and the flatness-error interlocks.
- The housekeeper task takes care of system status interlocks and limit checks and displays appropriate messages on the CRT display. It decodes some information and checks the setup verification.

Noncritical tasks include

- Checksum processor (scheduled to run periodically)
- Treatment console keyboard processor (scheduled to run only if it is called by other tasks or by keyboard interrupts). This task acts as the communication interface between the other software and the operator.
- Treatment console screen processor (run periodically). This task lays out appropriate record formats for either CRT displays or hard copies.
- Service keyboard processor (run on demand). This task arbitrates non-treatment-related communication between the therapy system and the operator.
- Snapshot (run periodically by the scheduler). Snapshot captures preselected parameter values and is called by the treatment task at the end of a treatment.
- Hand control processor (run periodically).
- Calibration processor. This task is responsible for a package of tasks that let the operator examine and change system setup parameters and interlock limits.

It is clear from the AECL documentation on the modifications that the software allows concurrent access to shared memory, that there is no real synchronization aside from data that are stored in shared variables, and that the "test" and "set" for such variables are not indivisible operations. Race conditions resulting from this implementation of multitasking played an important part in the accidents.

Specific Design Errors. The following explanation of the specific software problems found at this time is taken from the description AECL provided to the FDA, but clarified somewhat. The description leaves some unanswered questions, but it is the best that can be done with the information available.

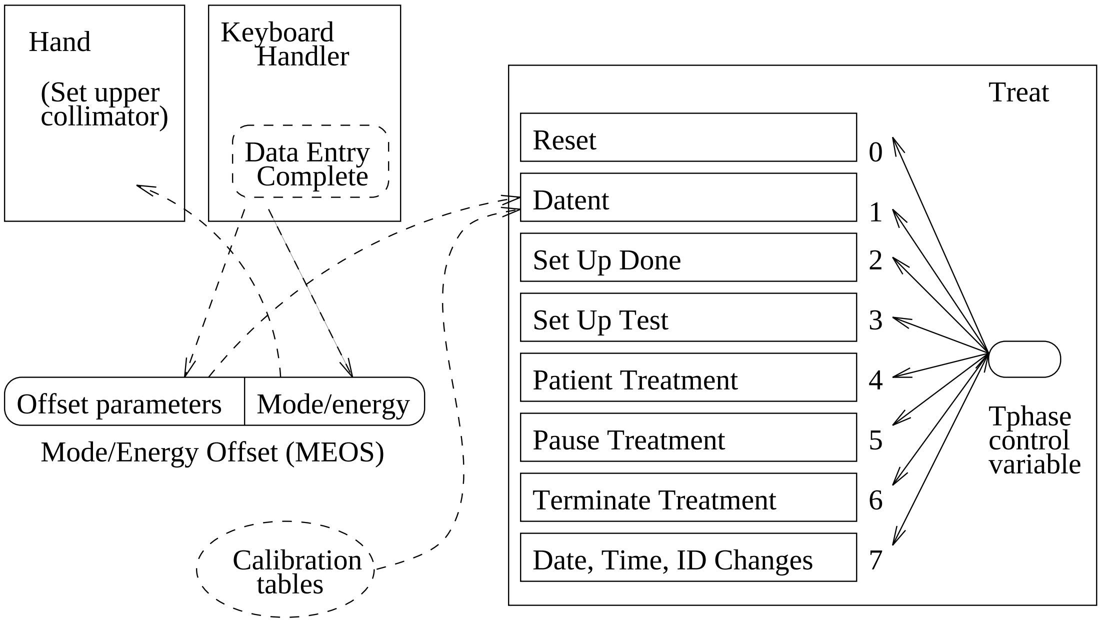


Figure 3: Tasks and subroutines in the code blamed for the Tyler accidents.

The treatment monitor task (Treat) controls the various phases of treatment by executing its eight subroutines. The treatment phase indicator variable (Tphase) is used to determine which subroutine should be executed (Figure 3). Following the execution of a particular subroutine, Treat reschedules itself.

One of Treat's subroutines, called Datent (data entry), communicates with the keyboard handler task (a task that runs concurrently with Treat) via a shared variable (Data Entry Complete flag) to determine whether the prescription data has been entered. The keyboard handler recognizes the completion of data entry and changes the Data Entry Complete variable to denote this. Once this variable is set, the Datent subroutine detects the variable's change in status and changes the value of Tphase from 1 (Datent) to 3 (Set Up Test). In this case, the Datent subroutine exits back to the Treat subroutine, which will reschedule itself and begin execution of the Set Up Test subroutine. If the Data Entry Complete variable has not been set, Datent leaves the value of Tphase unchanged and exits back to Treat's mainline. Treat will then reschedule itself, essentially rescheduling the Datent subroutine.

The command line at the lower right-hand corner of the screen (see Figure 2) is the cursor's normal position when the operator has completed all the necessary changes to the prescription. Prescription editing is signified by moving the cursor off the command line. As the program was originally designed, the Data Entry Complete variable by itself is not sufficient because it does not ensure that the cursor is located on the command line; under the right circumstances, the data entry phase can be exited before all edit changes are made on the screen.

The keyboard handler parses the mode and energy level specified by the operator and places an encoded result in another shared variable, the 2-byte Mode/Energy Offset variable (MEOS). The low-order byte of this variable is used by another task (Hand) to set the collimator/turntable to the proper position for the selected mode and energy. The high-order byte of the MEOS variable is used by Datent to set several operating parameters.

Initially, the data-entry process forces the operator to enter the mode and energy except when the photon mode is selected, in which case the energy defaults to 25 MeV. The operator can later edit the mode and energy separately. If the keyboard handler sets the Data Entry Complete flag before the operator changes the data in MEOS, Datent will not detect the changes because it has already exited and will not be reentered again. The upper collimator (turntable), on the other hand, is set to the position dictated by the low-order byte of MEOS by another concurrently running task (Hand) and can therefore be inconsistent with the parameters set in accordance with the information in the high-order byte. The software appears to contain no checks to detect such an incompatibility.

The first thing Datent does when it is entered is to check whether the keyboard handler has set the mode and energy in MEOS. If so, it uses the high-order byte to index into a table of preset operating parameters and places them in the digital-to-analog output table. The contents of this output table are transferred to the digital-to-analog converter during the next clock cycle. Once the parameters are all set, Datent calls the subroutine Magnet, which sets the bending magnets. The following shows a simplified pseudocode description of relevant parts of the software:

```
Datent:
        if mode/energy specified then
            begin
                calculate table index
                repeat
                    fetch parameter
                    output parameter
                    point to next parameter
                until all parameters set
                call Magnet
                if mode/energy changed then return
            end
        if data entry is complete then set Tphase to 3
        if data entry is not complete then
            if reset command entered then set Tphase to 0
        return

Magnet:
        Set bending magnet flag
        repeat
            Set next magnet
            call Ptime
            if mode/energy has changed, then exit
        until all magnets are set
        return

Ptime:
        repeat
            if bending magnet flag is set then
                if editing taking place then
                    if mode/energy has changed then exit
        until hysteresis delay has expired
        Clear bending magnet flag
        return
```

Setting the bending magnets takes about eight seconds. Magnet calls a subroutine called Ptime to introduce a time delay. Since several magnets need to be set, Ptime is entered and exited several times. A flag to indicate that the bending magnets are being set is initialized upon entry to the Magnet subroutine and cleared at the end of Ptime. Furthermore, Ptime checks a shared variable, set by the keyboard handler, that indicates the presence of any editing requests. If there are edits, then Ptime clears the bending magnet variable and exits to Magnet, which then exits to Datent. But the edit change variable is checked by Ptime only if the bending magnet flag is set. Because Ptime clears it during its first execution, any edits performed during each succeeding pass through Ptime will not be recognized. Thus, an edit change of the mode or energy, although reflected on the operator's screen and the mode/energy offset variable, will not be sensed by Datent so it can index the appropriate calibration tables for the machine parameters.

Recall that the Tyler error occurred when the operator made an entry indicating the mode and energy, went to the command line, then moved the cursor up to change the mode or energy and returned to the command line all within eight seconds. Because the magnet setting takes about eight seconds and Magnet does not recognize edits after the first execution of Ptime, the editing had been completed by the return to Datent, which never detected that it had occurred. Part of the problem was fixed after the accident by clearing the bending magnet variable at the end of Magnet (after *all* the magnets have been set) instead of at the end of Ptime.

But this is not the only problem. Upon exit from the Magnet subroutine, the data entry subroutine (Datent) checks the Data Entry Complete variable. If it indicates that data entry is complete, Datent sets Tphase to 3 and Datent is not entered again. If it is not set, Datent leaves Tphase unchanged, which means it will eventually be rescheduled. But the Data Entry Complete variable only indicates that the cursor has been down to the command line, not that it is still there. A potential race condition is set up. To fix this, AECL introduced another shared variable controlled by the keyboard handler task that indicates the cursor is not positioned on the command line. If this variable is set, then prescription entry is still in progress and the value of Tphase is left unchanged.

3.5.4 The Government and User Response

The FDA does not approve each new medical device on the market: All medical devices go through a classification process that determines the level of FDA approval necessary. Medical accelerators follow a procedure called pre-market notification before commercial distribution. In this process, the firm must establish that the product is substantially equivalent in safety and effectiveness to a product already on the market. If that cannot be done to the FDA's satisfaction, a pre-market approval is required. For the Therac-25, the FDA required only a pre-market notification. After the Therac-25 accidents, new procedures for approval of software-controlled devices were adopted.

The agency is basically reactive to problems and requires manufacturers to report serious ones. Once a problem is identified in a radiation-emitting product, the FDA is responsible for approving the corrective action plan (CAP).

The first reports of the Tyler incidents came to the FDA from the State of Texas Health Department, and this triggered FDA action. The FDA investigation was well under way when AECL produced a medical device report to discuss the details of the radiation overexposures at Tyler. The FDA declared the Therac-25 defective under the Radiation Control for Health and Safety Act and ordered the firm to notify all purchasers, investigate the problem, determine a solution, and submit a corrective action plan for FDA approval.

The final CAP consisted of more than twenty changes to the system hardware and software, plus modifications to the system documentation and manuals. Some of these changes were unrelated to the specific accidents, but were improvements to the general safety of the machine. The full CAP implementation, including an extensive safety analysis, was not complete until more than two years after the Tyler accidents.

AECL made their accident report to the FDA on April 15, 1986. On that same date, AECL sent out a letter to each Therac user recommending a temporary "fix" to the machine that would allow continued clinical use. The letter (shown in its complete form) stated:

> SUBJECT: CHANGE IN OPERATING PROCEDURES FOR THE THERAC 25 LINEAR ACCELERATOR
>
> Effective immediately, and until further notice, the key used for moving the cursor back through the prescription sequence (i.e., cursor 'UP' inscribed with an upward pointing arrow) must not be used for editing or any other purpose.

> To avoid accidental use of this key, the key cap must be removed and the switch contacts fixed in the open position with electrical tape or other insulating material. For assistance with the latter you should contact your local AECL service representative.
>
> Disabling this key means that if any prescription data entered is incorrect then a 'R' reset command must be used and the whole prescription reentered.
>
> For those users of the Multiport option it also means that editing of dose rate, dose and time will not be possible between ports.

On May 2, 1986, the FDA declared the Therac defective, demanded a CAP, and required renotification of all the Therac customers. In the letter from the FDA to AECL, the Director of Compliance, Center for Devices and Radiological Health, wrote:

> We have reviewed [AECL's] April 15 letter to purchasers and have concluded that it does not satisfy the requirements for notification to purchasers of a defect in an electronic product. Specifically, it does not describe the defect nor the hazards associated with it. The letter does not provide any reason for disabling the cursor key and the tone is not commensurate with the urgency for doing so. In fact, the letter implies the inconvenience to operators outweighs the need to disable the key. We request that you immediately renotify purchasers.

AECL promptly made a new notice to users and also requested an extension to produce a CAP. The FDA granted this request.

About this time, the Therac-25 users created a user's group and held their first meeting at the annual conference of the American Association of Physicists in Medicine. At the meeting, users discussed the Tyler accident and heard an AECL representative present the company's plans for responding to it. AECL promised to send a letter to all users detailing the CAP.

Several users described additional hardware safety features that they had added to their own machines to provide additional protection. An interlock (that checked gun current values), which the Vancouver clinic had previously added to their Therac-25, was labeled as redundant by AECL; the users disagreed. There were further discussions of poor design and other problems that caused a 10- to 30-percent underdosing in both modes.

The meeting notes said

> There was a general complaint by all users present about the lack of information propagation. The users were not happy about receiving incomplete information. The AECL representative countered by stating that AECL does not wish to spread rumors and that AECL has no policy to 'keep things quiet'. The consensus among the users was that an improvement was necessary.

After the first user's group meeting, there were two user's group newsletters. The first, dated fall 1986, contained letters from Tim Still, the Kennestone physicist, who complained about what he considered to be eight major problems he had experienced with the Therac-25. These problems included poor screen-refresh subroutines that leave trash and erroneous information on the operator console and some tape-loading problems upon startup that he discovered involved the use of "phantom tables" to trigger the interlock system in the event of a load failure instead of using a checksum. He asked the question, "Is programming safety relying too much on the software interlock routines?" The second user's group newsletter, in December 1986, further discussed the implications of the phantom table problem.

AECL produced its first CAP on June 13, 1986. The FDA asked for changes and additional information about the software, including a software test plan. AECL responded on September 26 with several documents describing the software and its modifications but no test plan. They explained how the Therac-25 software evolved from the Therac-6 software and stated that "no single test plan and report exists for the software since both hardware and software were tested and exercised separately and together over many years." AECL concluded that the current CAP improved "machine safety by many orders of magnitude and virtually eliminates the possibility of lethal doses as delivered in the Tyler incident."

An FDA internal memo dated October 20 commented on these AECL submissions, raising several concerns:

> Unfortunately, the AECL response also seems to point out an apparent lack of documentation on software specifications and a software test plan.

> . . . concerns include the question of previous knowledge of problems by AECL, the apparent paucity of software quality assurance at the manufacturing facility, and possible warnings and information dissemination to others of the generic type problems.
>
> . . . As mentioned in my first review, there is some confusion on whether the manufacturer should have been aware of the software problems prior to the ARO's [Accidental Radiation Overdoses] in Texas. AECL had received official notification of a law suit in November 1985 from a patient claiming accidental over-exposure from a Therac-25 in Marietta, Georgia.. . . If knowledge of these software deficiencies were known beforehand, what would be the FDA's posture in this case?
>
> . . . The materials submitted by the manufacturer have not been in sufficient detail and clarity to ensure an adequate software quality assurance program currently exists. For example, a response has not been provided with respect to the software part of the CAP to the CDRH's [FDA Center for Devices and Radiological Health] request for documentation on the revised requirements and specifications for the new software. In addition, an analysis has not been provided, as requested, on the interaction with other portions of the software to demonstrate the corrected software does not adversely affect other software functions.
>
> The July 23 letter from the CDRH requested a documented test plan including several specific pieces of information identified in the letter. This request has been ignored up to this point by the manufacturer. Considering the ramifications of the current software problem, changes in software QA attitudes are needed at AECL.

AECL also planned to retain the malfunction codes, but the FDA required better warnings for the operators. Furthermore, AECL had not planned on any quality assurance testing to ensure exact copying of software, but the FDA insisted on it. The FDA further requested assurances that rigorous testing would become a standard part of AECL's software modification procedures.

> We also expressed our concern that you did not intend to perform the protocol to future modifications to software. We believe that

> the rigorous testing must be performed each time a modification is made in order to ensure the modification does not adversely affect the safety of the system.

AECL was also asked to draw up an installation test plan to ensure that both hardware and software changes perform as designed when installed.

AECL submitted CAP Revision 2 and supporting documentation on December 22, 1986. They changed the CAP to have dose malfunctions suspend treatment and included a plan for meaningful error messages and highlighted dose error messages. They also expanded their diagrams of software modifications and expanded their test plan to cover hardware and software.

3.6 Yakima Valley Memorial Hospital, January 1987

On Saturday, January 17, 1987, the second patient of the day was to be treated for a carcinoma. This patient was to receive two film verification exposures of 4 and 3 rads plus a 79-rad photon treatment (for a total exposure of 86 rads.)

Film was placed under the patient and 4 rads were administered. After the machine paused to open the collimator jaws further, the second exposure of 3 rads was administered. The machine paused again.

The operator entered the treatment room to remove the film and verify the patient's precise position. He used the hand control in the treatment room to rotate the turntable to the field light position, which allowed him to check the alignment of the machine with respect to the patient's body in order to verify proper beam position. He then either pressed the *set* button on the hand control or left the room and typed a set command at the console to return the turntable to the proper position for treatment; there is some confusion as to exactly what transpired. When he left the room, he forgot to remove the film from underneath the patient. The console displayed "beam ready," and the operator hit the Ⓑ key to turn the beam on.

The beam came on, but the console displayed no dose or dose rate. After five or six seconds, the unit shut down with a pause and displayed a message. The message "may have disappeared quickly"; the operator was unclear on this point. However, since the machine merely paused, he was able to push the Ⓟ key to proceed with treatment.

The machine paused again, this time displaying FLATNESS on the reason line. The operator heard the patient say something over the intercom, but could not understand him. He went into the room to speak with the patient, who reported "feeling a burning sensation" in the chest. The console displayed only the total dose of the two film exposures (7 rads) and nothing more.

Later in the day, the patient developed a skin burn over the entire treatment area. Four days later, the redness developed a striped pattern matching the slots in the blocking tray. The striped pattern was similar to the burn a year earlier at this same hospital, which had first been ascribed to a heating pad and later officially labeled by the hospital as "cause unknown."

AECL began an investigation, and users were told to confirm the turntable position visually before turning on the beam. All tests run by the AECL engineers indicated that the machine was working perfectly. From the information that had been gathered to that point, it was suspected that the electron beam had come on when the turntable was in the field light position. But the investigators could not reproduce the fault condition.

On the following Thursday, AECL sent in an engineer from Ottawa to investigate. The hospital physicist had, in the meantime, run some tests himself. He placed a film in the Therac's beam and then ran two exposures of X-ray parameters with the turntable in field light position. The film appeared to match the film that was left (by mistake) under the patient during the accident.

After a week of checking the hardware, AECL determined that the "incorrect machine operation was probably not caused by hardware alone." After checking the software, AECL engineers discovered a flaw (described below) that could explain the erroneous behavior. The coding problems explaining this accident are completely different from those associated with the Tyler accidents.

Preliminary dose measurements by AECL indicated that the dose delivered under these conditions—that is, when the turntable is in the field light position—is on the order of 4,000 to 5,000 rads. After two attempts, the patient could have received 8,000 to 10,000 instead of the 86 rads prescribed. AECL again called users on January 26 (nine days after the accident) and gave them detailed instructions on how to avoid this problem. In an FDA internal report on the accident, the AECL quality assurance manager investigating the problem is quoted as saying that the software and hardware changes to be retrofitted following the Tyler accident nine months earlier (but which had not yet been installed) would have prevented the Yakima accident.

The patient died in April from complications related to the overdose. He had a terminal form of cancer, but a lawsuit was initiated by his survivors alleging that he died sooner than he would have and endured unnecessary pain and suffering due to the radiation overdose. The suit, like all the others, was settled out of court.

3.6.1 The Yakima Software "Bug"

The software problem for the second Yakima accident is fairly well-established and different from that implicated in the Tyler accidents. There is no way to determine what particular software design errors were related to the Kennestone, Hamilton, and first Yakima accidents. Given the unsafe programming practices exhibited in the code, unknown race conditions or errors could have been responsible for them. There is speculation, however, that the Hamilton accident was the same as this second Yakima overdose. In a report of a conference call on January 26, 1987, between the AECL quality assurance manager and Ed Miller of the FDA discussing the Yakima accident, Miller notes

> This situation probably occurred in the Hamilton, Ontario accident a couple of years ago. It was not discovered at that time and the cause was attributed to intermittent interlock failure. The subsequent recall of the multiple microswitch logic network did not really solve the problem.

The second Yakima accident was again attributed to a type of race condition in the software — this one allowed the device to be activated in an error setting (a "failure" of a software interlock). The Tyler accidents were related to problems in the data-entry routines that allowed the code to proceed to Set Up Test before the full prescription had been entered and acted upon. The Yakima accident involved problems encountered later in the logic after the treatment monitor Treat reaches Set Up Test.

The Therac-25's field light feature allows very precise positioning of the patient for treatment. The operator can control the machine right at the treatment site using a small hand control that offers certain limited functions for patient setup, including setting gantry, collimator, and table motions.

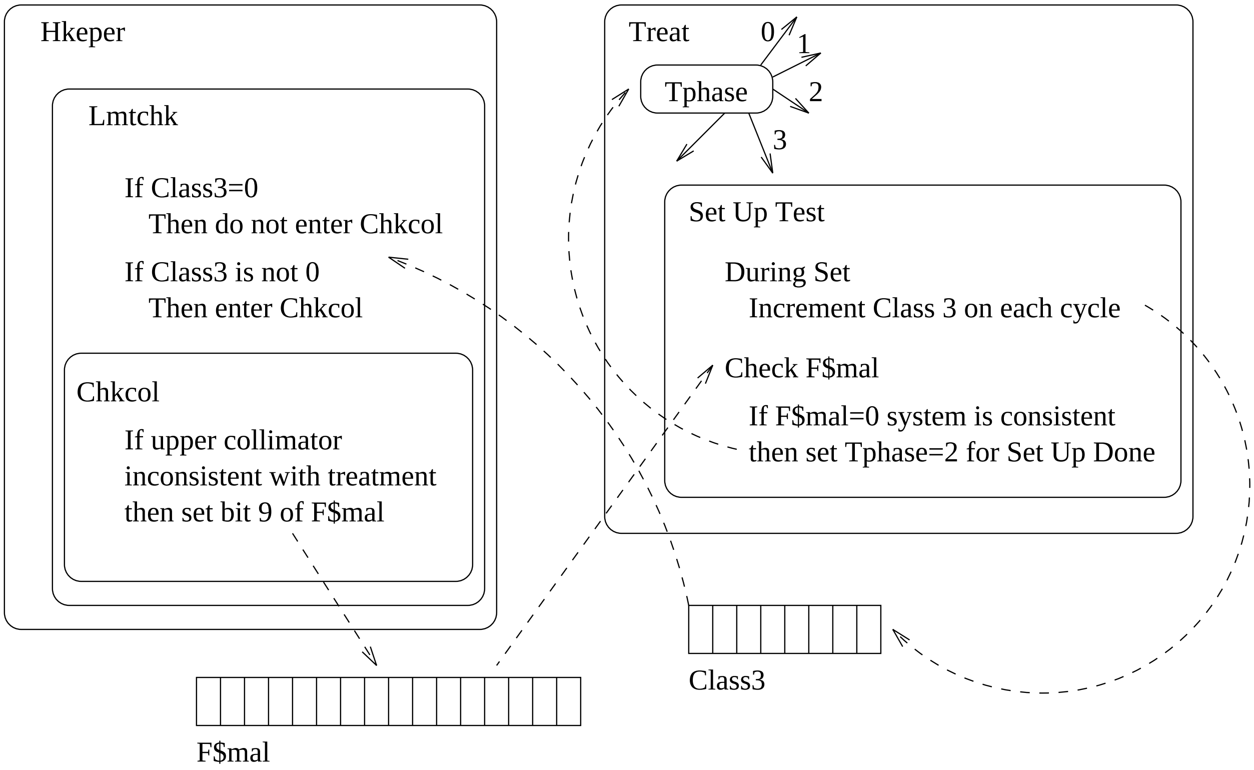


Figure 4: The Yakima software flaw.

Normally, the operator enters all the prescription data at the console (outside the treatment room) before the final setup of all machine parameters is completed in the treatment room. This gives rise to an UNVERIFIED condition at the console. The operator then completes patient setup in the treatment room, and all relevant parameters now VERIFY. The console displays a message to PRESS SET BUTTON while the turntable is in the field light position. The operator now presses the *set* button on the hand control or types "set" at the console. That should set the collimator to the proper position for treatment.

In the software, after the prescription is entered and verified by the Datent routine, the control variable Tphase is changed so that the Set Up Test routine is entered (Figure 4). Every pass through the Set Up Test routine increments the upper collimator position check, a shared variable called Class3. If Class3 is nonzero, there is an inconsistency and treatment should not proceed. A zero value for Class3 indicates that the relevant parameters are consistent with treatment, and the software does not inhibit the beam.

After setting the Class3 variable, Set Up Test next checks for any malfunctions in the system by checking another shared variable (set by a routine that actually handles the interlock checking) called F$mal to see if it has a nonzero value. A nonzero value in F$mal indicates that the machine is not ready for treatment, and the Set Up Test subroutine is rescheduled. When F$mal is zero (indicating that everything is ready for treatment), the Set Up Test subroutine sets the Tphase variable equal to 2, which results in next scheduling the Set Up Done subroutine and the treatment is allowed to continue.

The actual interlock checking is performed by a concurrent Housekeeper task (Hkeper). The upper collimator position check is performed by a subroutine of Hkeper called Lmtchk (analog-to-digital limit checking). Lmtchk first checks the Class3 variable. If Class3 contains a non-zero value, Lmtchk calls the Check Collimator (Chkcol) subroutine. If Class3 contains zero, Chkcol is bypassed and the upper collimator position check is not performed. The Chkcol subroutine sets or resets bit 9 of the F$mal shared variable, depending on the position of the upper collimator—which in turn is checked by the Set Up Test subroutine of Treat to decide whether to reschedule itself or to proceed to Set Up Done.

During machine setup, Set Up Test will be executed several hundred times because it reschedules itself waiting for other events to occur. In the code, the Class3 variable is incremented by one in each pass through Set Up Test. Since the Class3 variable is one byte, it can only contain a maximum value of 255 decimal. Thus, on every 256th pass through the Set Up Test code, the variable will overflow and have a zero value. That means that on every 256th pass through Set Up Test, the upper collimator will not be checked and an upper collimator fault will not be detected.

The overexposure occurred when the operator hit the “set” button at the precise moment that Class3 rolled over to zero. Thus, Chkcol was not executed and F$mal was not set to indicate that the upper collimator was still in the field-light position. The software turned on the full 25 MeV without the target in place and without scanning. A highly concentrated electron beam resulted, which was scattered and deflected by the stainless steel mirror that was in the path.

The technical "fix" implemented for this particular software flaw is described by AECL as simple: the program is changed so that the Class3 variable is set to some fixed nonzero value each time through Set Up Test instead of being incremented.

3.6.2 Manufacturer, Government, and User Response

On February 3, 1987, after interaction with the FDA and others, including the user's group, AECL announced to its customers

1. A new software release to correct both the Tyler and Yakima software problems
2. A hardware single-pulse shutdown circuit
3. A turntable potentiometer to independently monitor turntable position
4. A hardware turntable interlock circuit

The second item, a hardware single-pulse shutdown circuit, essentially acts as a hardware interlock to prevent overdosing by detecting an unsafe level of radiation and halting beam output after one pulse of high energy and current. This interlock effectively provides an independent way to protect against a wide range of potential hardware failures and software errors. The third item, a turntable potentiometer, was the safety device recommended by several groups after the Hamilton accident.

After the second Yakima accident, the FDA became concerned that the use of the Therac-25 during the CAP process, even with AECL's interim operating instructions, involved too much risk to patients. The FDA concluded that the accidents demonstrated that the software alone could not be relied upon to assure safe operation of the machine. In a February 18, 1987, internal FDA memorandum, the Director of the Division of Radiological Products wrote:

> It is impossible for CDRH to find all potential failure modes and conditions of the software. AECL has indicated the "simple software fix" will correct the turntable position problem displayed at Yakima. We have not yet had the opportunity to evaluate that modification. Even if it does, based upon past history, I am not

> convinced that there are not other software glitches that could result in serious injury.
>
> . . . We are in the position of saying that the proposed CAP can reasonably be expected to correct the deficiencies for which they were developed (Tyler). We cannot say that we are reasonable [sic] confident about the safety of the entire system to prevent or minimize exposure from other fault conditions.

On February 6, 1987, Ed Miller of the FDA called Pavel Dvorak of Canada's Health and Welfare to advise him that the FDA would recommend that all Therac-25s be shutdown until permanent modifications could be made. According to Miller's notes on the phone call, Dvorak agreed and indicated that Health and Welfare would coordinate their actions with the FDA.

AECL responded on April 13 with an update on the Therac CAP status and a schedule of the nine action items pressed by the users at a user's group meeting in March. This unique and highly productive meeting provided an unusual opportunity to involve the users in the CAP evaluation process. It brought together all concerned parties in one place and at one time so that a course of action could be decided upon and approved as quickly as possible. The attendees included representatives from

- The manufacturer (AECL)
- All users, including their technical and legal staffs
- The FDA and the Canadian Bureau of Radiation and Medical Devices
- the Canadian Atomic Energy Control Board
- the Province of Ontario
- the Radiation Regulations Committee of the Canadian Association of Physicists

According to Gordon Symonds, from the Canadian BRMD, this meeting was very important to the resolution of the problems, since the regulators, users, and manufacturer arrived at a consensus in one day.

At this second user's meeting, the participants carefully reviewed all the six known major Therac-25 accidents to that date and discussed the elements of the CAP along with possible additional modifications. They came up with a prioritized list of modifications they wanted included in the CAP and expressed concerns about the lack of independent evaluation of the software and the lack of a hardcopy audit trail to assist in diagnosing faults.

The AECL representative, who was the quality assurance manager, responded that tests had been done on the CAP changes, but that the tests were not documented and that independent evaluation of the software "might not be possible." He claimed that two outside experts had reviewed the software, but he could not provide their names. In response to user requests for a hard copy audit trail and access to source code, he explained that memory limitations would not permit including such options and that source code would not be made available to users.

On May 1, AECL issued CAP Revision 4 as a result of the FDA comments and the user's meeting input. The FDA response on May 26 approved the CAP subject to submission of the final test plan results and an independent safety analysis, distribution of the draft revised manual to customers, and completion of the CAP by June 30, 1987. The FDA concluded by rating this a Class I recall: a recall in which there is a reasonable probability that the use of, or exposure to, a violative product will cause serious adverse health consequences or death [1].

AECL sent more supporting documentation to the FDA on June 5, 1987, including the CAP test plan, a draft operator's manual, and the draft of the new safety analysis. This time the analysis included the software in the fault trees but used a "generic failure rate" of 10^{-4} for software events. This number was justified as being based on the historical performance of the Therac-25 software. The final report on the safety analysis states that many of the fault trees had a computer malfunction as a causative event, and the outcome for quantification was therefore dependent on the failure rate chosen for the software. Assuming that all software errors are equally likely seems rather strange.

A close inspection of the code was also conducted during this safety analysis to "obtain more information on which to base decisions." An outside consultant performed the inspection, which included a detailed examination of the implementation of each function, a search for coding errors, and a qualitative assessment of the software's reliability. No information is provided in the final safety report about whether any particular methodology or tools were used in the software inspection or whether someone just read the code looking for errors.

AECL planned a fifth revision of the CAP to include the testing and final safety analysis results. Referring to the test plan at this, the final stage of the CAP process, an FDA reviewer said,

> Amazingly, the test data presented to show that the software changes to handle the edit problems in the Therac-25 are appropriate prove the exact opposite result. A review of the data table in the test results indicates that the final beam type and energy (edit change) has no effect on the initial beam type and energy. I can only assume that either the fix is not right or the data was entered incorrectly. The manufacturer should be admonished for this error. Where is the QC [Quality Control] review for the test program? AECL must: (1) clarify this situation, (2) change the test protocol to prevent this type of error from occurring, and (3) set up appropriate QC control on data review.

A further FDA memo indicated:

> [The AECL quality assurance manager] could not give an explanation and will check into the circumstances. He subsequently called back and verified that the technician completed the form incorrectly. Correct operation was witnessed by himself and others. They will repeat and send us the correct data sheet.

At the American Association of Physicists in Medicine meeting in July 1987, a third user's meeting was held. The AECL representative described the status of the latest CAP and explained that the FDA had given verbal approval and that he expected full implementation by the end of August 1987. He went on to review and comment on the prioritized concerns of the last meeting. Three of the user-requested hardware changes had been included in the CAP. Changes to tape load error messages and checksums on the load data would wait until after the CAP was done. Software documentation was described as a lower priority task that needed definition and would not be available to the FDA in any form for over a year.

On July 6, 1987, AECL sent a letter to all users to update them on the FDA's verbal approval of the CAP and to delineate how AECL would proceed. Finally, on July 21, 1987, AECL issued the final and fifth CAP revision. The major features of the final CAP are these:

- All interruptions related to the dosimetry system will go to a treatment suspend, not a treatment pause. Operators will not be allowed to restart the machine without reentering all parameters.
- A software single-pulse shutdown will be added.
- An independent hardware single-pulse shutdown will be added.
- Monitoring logic for turntable position will be improved to ensure that the turntable is in one of the three legal positions.
- A potentiometer will be added to the turntable. The output is used to monitor exact turntable location and provide a visible position signal to the operator.
- Interlocking with the 270-degree bending magnet will be added to ensure that the target and beam flattener are in position if the X-ray mode is selected.
- Beam-on will be prevented if the turntable is in the field light or any intermediate position.
- Cryptic malfunction messages will be replaced with meaningful messages and highlighted dose-rate messages.
- Editing keys will be limited to *cursor up*, *backspace*, and *return*. All other keys will be inoperative.
- A motion-enable footswitch (a type of deadman switch) will be added. The operator will be required to hold this switch closed during movement of certain parts of the machine to prevent unwanted motions when the operator is not in control.
- Twenty three other changes will be made to the software to improve its operation and reliability, including disabling of unused keys, changing the operation of the *set* and *reset* commands, preventing copying of the control program on site, changing the way various detected hardware faults are handled, eliminating errors in the software that were detected during the review process, adding several additional software interlocks, disallowing changes in the service mode while a treatment is in progress, and adding meaningful error messages.
- The known software problems associated with the Tyler and Yakima accidents will be fixed.
- The manuals will be fixed to reflect the changes.

Figure 5 shows a typical Therac-25 installation after the CAP changes were made.

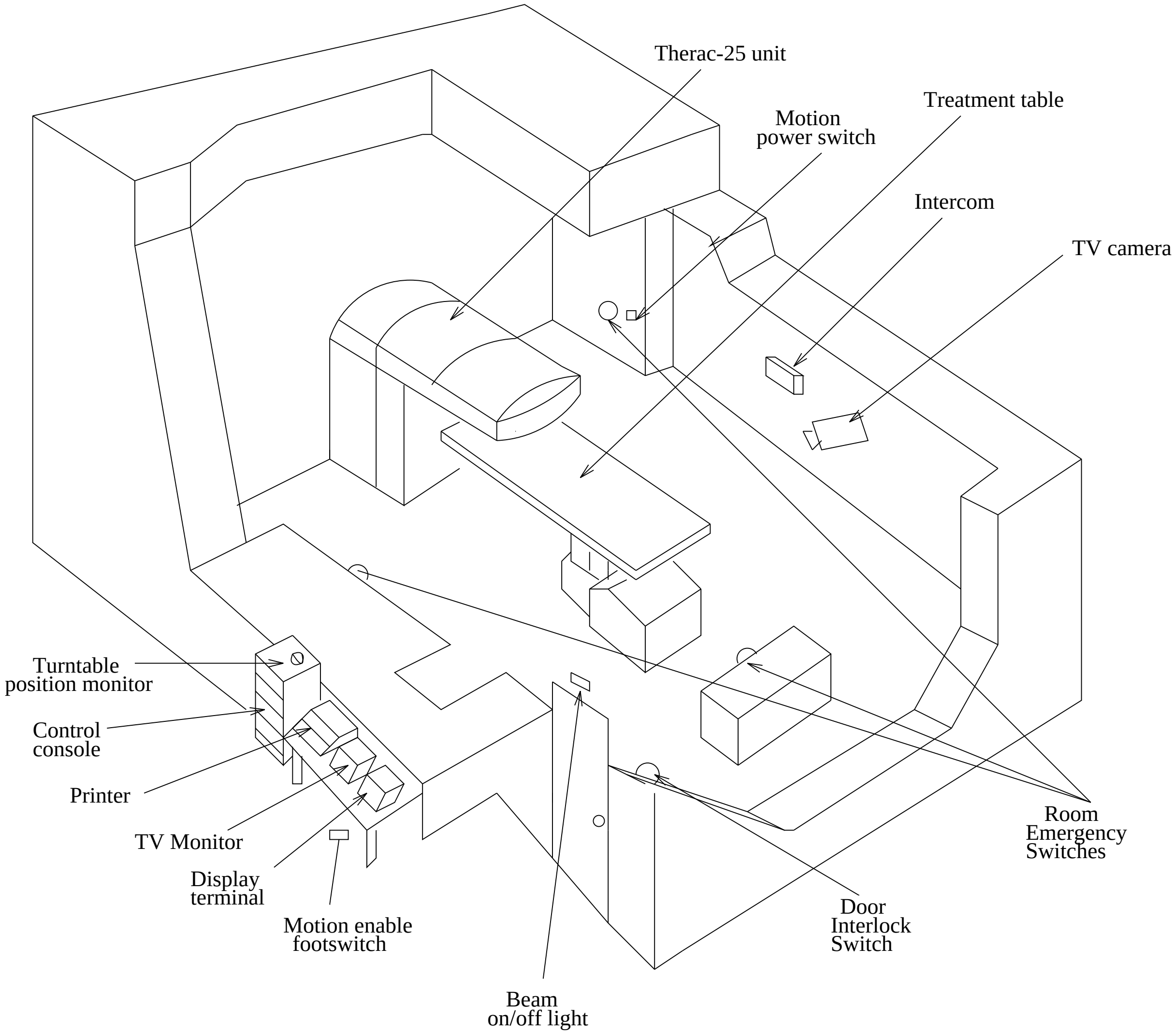


Figure 5: A typical Therac-25 facility after the final CAP.

Ed Miller, the director of the Division of Standards Enforcement, Center for Devices and Radiological Health at the FDA, wrote in 1987:

> FDA has performed extensive review of the Therac-25 software and hardware safety systems. We cannot say with absolute certainty that all software problems that might result in improper dose have been found and eliminated. However, we are confident that the hardware and software safety features recently added will prevent future catastrophic consequences of failure.

No Therac-25 accidents have been reported since the final corrective action plan was implemented.

4 Causal Factors

Many lessons can be learned from this series of accidents. A few are considered here.

Overconfidence in Software. A common mistake in engineering, in this case and in many others, is to put too much confidence in software. There seems to be a feeling among nonsoftware professionals that software will not or cannot fail, which leads to complacency and overreliance on computer functions.

A related tendency among engineers is to ignore software. The first safety analysis on the Therac-25 did not include software—although nearly full responsibility for safety rested on it. When problems started occurring, it was assumed that hardware had caused them, and the investigation looked only at the hardware.

Confusing Reliability with Safety. This software was highly reliable. It worked tens of thousands of times before overdosing anyone, and occurrences of erroneous behavior were few and far between. AECL assumed that their software was safe because it was reliable, and this led to complacency.

Lack of Defensive Design. The software did not contain self-checks or other error-detection and error-handling features that would have detected the inconsistencies and coding errors. Audit trails were limited because of a lack of memory. However, today larger memories are available and audit trails and other design techniques must be given high priority in making tradeoff decisions.

Patient reactions were the only real indications of the seriousness of the problems with the Therac-25; there were no independent checks that the machine and its software were operating correctly. Such verification cannot be assigned to operators without providing them with some means of detecting errors: The Therac-25 software "lied" to the operators, and the machine itself was not capable of detecting that a massive overdose had occurred. The ion chambers on the Therac-25 could not handle the high density of ionization from the unscanned electron beam at high beam current; they thus became saturated and gave an indication of a low dosage. Engineers need to design for the worst case.

Failure to Eliminate Root Causes. One of the lessons to be learned from the Therac-25 experiences is that focusing on particular software design errors is not the way to make a system safe. Virtually all complex software can be made to behave in an unexpected fashion under some conditions: There will always be another software bug. Just as engineers would not rely on a design with a hardware single point of failure that could lead to catastrophe, they should not do so if that single point of failure is software.

The Therac-20 contained the same software error implicated in the Tyler deaths, but this machine included hardware interlocks that mitigated the consequences of the error. Protection against software errors can and should be built into both the system and the software itself. We cannot eliminate all software errors, but we can often protect against their worst effects, and we can recognize their likelihood in our decision making.

One of the serious mistakes that led to the multiple Therac-25 accidents was the tendency to believe that the cause of an accident had been determined (e.g., a microswitch failure in the case of Hamilton) without adequate evidence to come to this conclusion and without looking at all possible contributing factors. Without a thorough investigation, it is not possible to determine whether a sensor provided the wrong information, the software provided an incorrect command, or the actuator had a transient failure and did the wrong thing on its own. In the case of the Hamilton accident, a transient microswitch failure was assumed to be the cause even though the engineers were unable to reproduce the failure or to find anything wrong with the microswitch.

In general, it is a mistake to patch just one causal factor (such as the software) and assume that future accidents will be eliminated. Accidents are unlikely to occur in exactly the same way again. If we patch only the symptoms and ignore the deeper underlying causes, or if we fix only the specific cause of one accident, we are unlikely to have much effect on future accidents. The series of accidents involving the Therac-25 is a good example of exactly this problem: Fixing each individual software flaw as it was found did not solve the safety problems of the device.

Complacency. Often it takes an accident to alert people to the dangers involved in technology. A medical physicist wrote about the Therac-25 accidents:

> In the past decade or two, the medical accelerator "industry" has become perhaps a little complacent about safety. We have assumed that the manufacturers have all kinds of safety design experience since they've been in the business a long time. We know that there are many safety codes, guides, and regulations to guide them and we have been reassured by the hitherto excellent record of these machines. Except for a few incidents in the 1960's (e.g., at Hammersmith, Hamburg) the use of medical accelerators has been remarkably free of serious radiation accidents until now. Perhaps, though we have been spoiled by this success [6].

This problem seems to be common in all fields.

Unrealistic Risk Assessments. The first hazard analyses initially ignored software, and then they treated it superficially by assuming that all software errors were equally likely. The probabilistic risk assessments generated undue confidence in the machine and in the results of the risk assessment themselves. When the first Yakima accident was reported to AECL, the company did not investigate. Their evidence for their belief that the radiation burn could not have been caused by their machine included a probabilistic risk assessment showing that safety had increased by five orders of magnitude as a result of the microswitch fix.

The belief that safety had been increased by such a large amount seems hard to justify. Perhaps it was based on the probability of failure of the microswitch (typically 10^{-5}) AND-ed with the other interlocks. The problem with all such analyses is that they typically make many independence assumptions and exclude aspects of the problem—in this case, software—that are difficult to quantify but which may have a larger impact on safety than the quantifiable factors that are included.

Inadequate Investigation or Followup on Accident Reports. Every company building safety-critical systems should have audit trails and incident analysis procedures that are applied whenever any hint of a problem is found that might lead to an accident. The first phone call by Tim Still should have led to an extensive investigation of the events at Kennestone. Certainly, learning about the first lawsuit should have triggered an immediate response.

Inadequate Software Engineering Practices. Some basic software engineering principles that apparently were violated in the case of the Therac-25 include the following:

- Software specifications and documentation should not be an afterthought.
- Rigorous software quality assurance practices and standards should be established.
- Designs should be kept simple and dangerous coding practices avoided.
- Ways to detect errors and and get information about them, such as software audit trails, should be designed into the software from the beginning.
- The software should be subjected to extensive testing and formal analysis at the module and software level; system testing alone is not adequate. Regression testing should be performed on all software changes.
- Computer displays and the presentation of information to the operators, such as error messages, along with user manuals and other documentation need to be carefully designed.

The manufacturer said that the hardware and software were "tested and exercised separately or together over many years." In his deposition for one of the lawsuits, the quality assurance manager explained that testing was done in two parts. A "small amount" of software testing was done on a simulator, but most of the testing was done as a system. It appears that unit and software testing was minimal, with most of the effort directed at the integrated system test. At a Therac-25 user's meeting, the same man stated that the Therac-25 software was tested for 2,700 hours. Under questioning by the users, he clarified this as meaning "2700 hours of use." The FDA difficulty in getting an adequate test plan out of the company and the lack of regression testing are evidence that testing was not done well.

The design is unnecessarily complex for such critical software. It is untestable in the sense that the design ensured that the known errors (there may very well be more that have just not been found) would most likely not have been found using standard testing and verification techniques. This does not mean that software testing is not important, only that software must be designed to be testable and that simple designs may prevent errors in the first place.

Software Reuse. Important lessons about software reuse can be found in these accidents. A naive assumption is often made that reusing software or using commercial off-the-shelf software will increase safety because the software will have been exercised extensively. Reusing software modules does not guarantee safety in the new system to which they are transferred and sometimes leads to awkward and dangerous designs. Safety is a quality of the system in which the software is used; it is not a quality of the software itself. Rewriting the entire software in order to get a clean and simple design may be safer in many cases.

Safe versus Friendly User Interfaces. Making the machine as easy as possible to use may conflict with safety goals. Certainly, the user interface design left much to be desired, but eliminating multiple data entry and assuming that operators would check the values carefully before pressing the return key was unrealistic.

User and Government Oversight and Standards. Once the FDA got involved in the Therac-25, their response was impressive, especially considering how little experience they had with similar problems in computer-controlled medical devices. Since the Therac-25 events, the FDA has moved to improve the reporting system and to augment their procedures and guide-lines to include software. The input and pressure from the user group was also important in getting the machine fixed and provides an important lesson to users in other industries.

References

[1] C.A. Bowsher. Medical device recalls: Examination of selected cases. Technical Report GAO Report GAO/PEMD-90-6, U.S. Government Accounting Organization, October 1990.

[2] C.A. Bowsher. Medical devices: The public health at risk. Technical Report GAO Report GAO/T-PEMD-90-2, U.S. Government Accounting Organization, 1990.

[3] M. Kivel, editor. *Radiological Health Bulletin*, volume XX:8. Center for Devices and Radiological Health, Food and Drug Administration, Rockville, Maryland, December 1986.

[4] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[5] Ed Miller. The Therac-25 experience. In *Conference of State Radiation Control Program Directors*, 1987.

[6] J.A. Rawlinson. Report on the Therac-25. In *OCTRF/OCI Physicists Meeting*, Kingston, Ontario, May 1987.

[7] R. Saltos. Man killed by accident with medical radiation. *Boston Globe*, June 20 1986.