CS 61C **Project 1**
Summer 2004

Goals

This project is intended to solidify your understanding of pointers and structs in C, and to understand some aspects of an interpreter similar to what you used in CS 61A.

Administrative Requirements

Submit your solution online by 11:59 PM on Tuesday, July 6. Do this by creating a directory named `proj1` that contains the files named `lisp.c` and `README`. From within that directory, type `submit proj1`. There will be two homework assignments and two lab assignments each week, so get started early on this assignment.

You must work individually. Your submission must be solely your work. You may discuss the project with others in the class, but you should understand all aspects of the code you submit.

Problem

The file `~cs61c/lib/lisp.c` implements an interpreter of a tiny subset of common Lisp. Your job is to extend the interpreter in two ways. Before you begin coding, make sure you understand the code provided for you. Part of the object of this exercise is for you to learn C, so ask questions!

1. Add a symbol table and implement `define`

Add a symbol table so that `eval` can look up the values of symbols. Since there are no user-defined procedures in this Lisp, all symbols will be global. Don't worry about optimizing lookup speed. A simple linear search structure will suffice.

Also add the `define` special form that adds an entry to the symbol table. Some examples appear below. You should check that the first argument to `define` is a symbol.

```
> (define x (+ 2 3))
x
> x
5
> (define a (cons ‘x ‘( )))
a
> (car a)
x
> (cons ‘a (cons a a))
(a (x) x)
> (define b (cons ‘y a))
b
> (define a (cons 3 b))
a
> a
(3 y x)
> b
(y x)
```

As in common Lisp, your interpreter will maintain separate name spaces for procedures and data. Don't worry about ever having to evaluate a procedure other than to call it directly. `define` should return the name of the defined variable.

2. Add the vector data type

Add a vector data type to the interpreter. For this data type, the `struct thing` should include a pointer to a dynamically-allocated block of storage just large enough for the vector and an integer indicating the length of the vector. The elements of the vectors should be pointers to `thing`s. The elements of a vector can be any type.

Also add the following primitive procedures:

`(make-vector length)`	returns a vector of `length` empty lists
`(vector-ref vector index)`	returns the `index`th element of `vector`
`(vector-set! vector index value)`	changes the value of the `index`th element of `vector` to `value`

These procedures should error-check their arguments. Make sure that a length provided to `make-vector` is a positive integer, the first argument to `vector-ref` and `vector-set!` is a vector, and the second argument to `vector-ref` and `vector-set!` is a legal index for the vector. Given invalid arguments, these procedures should return `NIL`. A successful `make-vector` should return the vector, and a successful `vector-set!` should return `okay`.

Finally, add the notation `#(element element ... element)` to the reader and printer. This notation represents a vector whose `element`s are as specified. Each `element` should be regarded as quoted, except an element may also be another vector.

```
> (define v #(#(2 3) x ‘(a b)))
v
> (vector-ref v 0)
#(2 3)
> (vector-ref v 1)
x
> (vector-ref v 2)
(quote (a b))
> (vector-ref (vector-ref v 0) 1)
3
> (car (vector-ref v 2))
quote
> (define x 10)
x
> (vector-set! v 1 x)
okay
> v
#(#(2 3) 10 ‘(a b))
> (define x 11)
x
> v
#(#(2 3) 10 ‘(a b))
> x
11
```

If in doubt, use the command `stk` in an instructional machine to test the behavior of a reference Lisp interpreter.

README

In your `README` file, describe clearly what changes you made to the code we’ve provided. Also, describe what parts of your project work and what parts don’t work. Include enough comments in your code to ensure that readers will understand your solution.

```
/* micro-lisp */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

typedef int boolean;
#define FALSE 0
#define TRUE !FALSE

struct thing {
    char th_type;
    union {
        char *thu_symbol;
        int thu_number;
        struct {struct thing *thup_car, *thup_cdr;} thu_pair;
    } th_union;
};

/* types in th_type */
#define SYMBOL 1
#define NUMBER 2
#define PAIR 3

#define th_symbol th_union.thu_symbol
#define th_number th_union.thu_number
#define th_car th_union.thu_pair.thup_car
#define th_cdr th_union.thu_pair.thup_cdr

typedef struct thing *ThingPtr;

#define NIL (ThingPtr)0

ThingPtr cons(ThingPtr a, ThingPtr b);
ThingPtr make_symbol(char *str);
ThingPtr make_number(int num);
ThingPtr car(ThingPtr pair);
ThingPtr cdr(ThingPtr pair);
void printthing(ThingPtr th);
ThingPtr readthing(void);
boolean isNumber(char *str);
ThingPtr quoted(ThingPtr th);
ThingPtr readlist(void);
ThingPtr plus(ThingPtr a, ThingPtr b);
ThingPtr eval(ThingPtr exp);

/* constructors */

ThingPtr cons(ThingPtr a, ThingPtr b) {
    ThingPtr new;

    new = (ThingPtr)malloc(sizeof(struct thing));
    new->th_type = PAIR;
    new->th_car = a;
    new->th_cdr = b;
    return(new);
}
```



```
ThingPtr make_symbol(char *str) {
    ThingPtr new;

    new = (ThingPtr)malloc(sizeof(struct thing));
    new->th_type = SYMBOL;
    new->th_symbol = str;
    return(new);
}

ThingPtr make_number(int num) {
    ThingPtr new;

    new = (ThingPtr)malloc(sizeof(struct thing));
    new->th_type = NUMBER;
    new->th_number = num;
    return(new);
}

/* selectors */

ThingPtr car(ThingPtr pair) {
    if (pair == NIL || pair->th_type != PAIR) {
        printf("Error: car of non-pair.\n");
        return(NIL);
    }
    return(pair->th_car);
}

ThingPtr cdr(ThingPtr pair) {
    if (pair == NIL || pair->th_type != PAIR) {
        printf("Error: cdr of non-pair.\n");
        return(NIL);
    }
    return(pair->th_cdr);
}

#define cadr(x) (car(cdr(x)))
#define caddr(x) (car(cdr(cdr(x))))


/* printer */

void printthing(ThingPtr th) {
    void printtail(ThingPtr);

    if (th == NIL)
           printf("()");
    else switch (th->th_type) {
        case SYMBOL:
            printf("%s",th->th_symbol);
               break;
        case NUMBER:
               printf("%d",th->th_number);
               break;
        case PAIR:
               printf("(");
               printthing(car(th));
               printtail(cdr(th));
    }
}
```

```
void printtail(ThingPtr th) {
    if (th == NIL)
          printf(")");
    else switch (th->th_type) {
        case SYMBOL:
              printf(" . %s)",th->th_symbol);
              break;
        case NUMBER:
              printf(" . %d)",th->th_number);
              break;
        case PAIR:
              printf(" ");
              printthing(car(th));
              printtail(cdr(th));
    }
}

/* reader */

ThingPtr readthing(void) {
    ThingPtr readlist(void), quoted(ThingPtr);
    char ch, buffer[30], *new;

    while (isspace((int) ch = getchar())) ;
    if (ch == '(')
          return(readlist());
    if (ch == '\'')
          return(quoted(readthing()));
    if (ch == ')') {
          printf("Unmatched close paren.\n");
          return(NIL);
    }
    ungetc(ch, stdin);
    scanf("%29[^() \n\t]",buffer);
    if (isNumber(buffer))
          return(make_number(atoi(buffer)));
    new = (char *)malloc(1+strlen(buffer));
    strcpy(new,buffer);
    return(make_symbol(new));
}

boolean isNumber(char *str) {
    char ch = *str;

    if (ch == '-')
          str++;
    while ((ch = *str++)) {
        if (ch < '0' || ch > '9')
            return(FALSE);
    }
    return(TRUE);
}

ThingPtr quoted(ThingPtr th) {
    return(cons(make_symbol("quote"),cons(th,NIL)));
}
```



```
ThingPtr readlist(void) {
    ThingPtr new;
    char ch;

    while (isspace((int) ch = getchar())) ;
    if (ch == ')')
          return(NIL);
    ungetc(ch,stdin);
    new = readthing();
    if (new != NIL && new->th_type == SYMBOL && !strcmp(new-
>th_symbol,".")) {
        new = readthing();
        ch = getchar();
        if (ch != ')')
            printf("Error in dot notation.\n");
        return(new);
    }
    return(cons(new,readlist()));
}

/* vast collection of primitives */

ThingPtr plus(ThingPtr a, ThingPtr b) {
    if (a == NIL || a->th_type != NUMBER
                 || b == NIL || b->th_type != NUMBER) {
        printf("Non-numeric arg to plus.\n");
        return(NIL);
    }
    return(make_number(a->th_number + b->th_number));
}

/* eval */

ThingPtr eval(ThingPtr exp) {
    char *fn;

    if (exp == NIL)
          return(NIL);
    switch(exp->th_type) {
        case NUMBER:
              return(exp);
        case SYMBOL:
              printf("What do you expect, variables???\n");
              return(NIL);
        case PAIR:
              if ((car(exp))->th_type != SYMBOL) {
                    printf("Application of non-procedure\n");
                    return(NIL);
              }
              fn = (car(exp))->th_symbol;
              if (!strcmp(fn,"quote"))
                    return(cadr(exp));
              if (!strcmp(fn,"+"))
                    return(plus(eval(cadr(exp)),eval(caddr(exp))));
              if (!strcmp(fn,"car"))
                    return(car(eval(cadr(exp))));
              if (!strcmp(fn,"cdr"))
                    return(cdr(eval(cadr(exp))));
              if (!strcmp(fn,"cons"))
                    return(cons(eval(cadr(exp)),eval(caddr(exp))));
```

```
    if (!strcmp(fn,"exit"))
        exit(0);
    printf("Unknown function\n");
    return(NIL);
  }
  return 0;
}

/* read-eval-print loop */

int main() {
  for (;;) {
      printf("> ");
      printthing(eval(readthing()));
      printf("\n");
  }
  return 0;
}
```