| **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 $n$th 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 $\pi$ 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 $\pi$. 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.