CS61A Summer 2010

Homework 1

Due: June 28, 2010 at 7am

Note: The number of stars (★) besides a question represents the estimated difficulty of the problem: the more the number of stars, the harder the question.

1 How to Start and Submit

First, download `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/hw1.scm` and use it as a template while filling out your procedures.

See `http://www-inst.eecs.berkeley.edu/~cs61a/su10/hw-faq.pdf` for submission instructions.

2 Case Study: Calculator

Ever wonder how your calculator works? Well, for your first homework, you will develop an approximation of a simple computer. To get you started, we have provided a framework that interprets the input and is capable of doing addition and subtraction, available as part of the skeleton file. You will augment this framework with other basic operations. Read the information at the top of the file to determine how to run the calculator.

For simplicity, your program does not have to take into account negative numbers.

1. (★) Fill in the `multiply` procedure, without using the `*` primitive. The calculator's version of multiplication takes in two numbers and outputs a single number. For example, typing `(* 3 4)` in the calculator (which calls your `multiply` function) returns 3×4. *Hint*: Think about multiplication as iterated addition. For instance, $4 \times 1 = 4$, while $4 \times 3 = 4 + (4 \times 2) = 4 + 4 + (4 \times 1)$, and so on.

2. (★) Fill in the `exponentiate` procedure, without using any primitives. The calculator's version of exponentiation takes in two numbers, a base and a power. It returns a single number: the base raised to the power. For example, typing `(^ 3 4)` in the calculator (which calls your `exponentiate` function) returns 3^4. *Hint*: The implementation is analogous to the multiplication procedure, but you will instead use multiplication iteratively.

3. (★★) Now, fill in the `divide` procedure. This procedure takes in two numbers—the divisor and the dividend—and returns a sentence of the quotient and the remainder. For example, typing `(/ 7 3)` in the calculator (which calls your `divide` function) returns the sentence `(2 1)`. Do not use the primitive `/` in your procedure. *Hint*: Think of division as iterated subtraction. However, you want to think about how to avoid negative and return a positive remainder instead.

4. (★★★) **Fast Multiplication**. The grade-school method of multiplication of long numbers works fine for relatively small multiplications. However, when we need to multiply thousand-digit numbers by other thousand-digit numbers, it turns out that there are faster ways to multiply, such as the Karatsuba algorithm. (To quantify exactly how much faster, we will have to wait until week 2).

   Learn the Karatsuba algorithm and implement it in the file provided, in the `karatsuba` function. To test it in your calculator, use `**` instead of `*`. You may assume the numbers have the same number of digits and that the number of digits is a power of 2.

   Hint: Useful resources are available at Wikipedia (`http://en.wikipedia.org/wiki/Karatsuba_algorithm`) and at Wolfram MathWorld (`http://mathworld.wolfram.com/KaratsubaMultiplication.html`).

3 Recursion

1. (★) Write a procedure `squares` that takes a sentence of numbers as its argument and returns a sentence of the squares of the numbers:

```
> (squares '(2 3 4 5))
(4 9 16 25)
```

2. (★) Write a procedure `switch` that takes a sentence as its argument and returns a sentence in which every instance of the words `I` or `me` is replaced by `you`, while every instance of `you` is replaced by `me`, except at the beginning of the sentence, where it is replaced by `I`. (Don't worry about capitalization of letters.) For example:

```
> (switch '(You told me that I should wake you up))
(i told you that you should wake me up)
```

3. You are writing Solitaire for Windows 7 and you decide to represent the win/loss record by a sentence of just the word `w` for win and `l` for loss.

   (a) (★) First, write a procedure `first-streak` that takes a sentence representing the win/loss record and finds the length of the first streak, be it a winning or a losing streak. For example:

```
> (first-streak '(w w l l l))
2

> (first-streak '(l w l w w w))
1
```

   (b) (★★) Now, using the `first-streak` procedure, write a new procedure `best-streak` that takes a sentence of win/loss record and returns the length of the longest streak, be it a winning or a losing streak. For example:

```
> (best-streak '(w w l l l))
3

> (best-streak '(l w l w w w))
3
```

   Hint: The key observation you need to make is that the longest streak is either the first streak, or the longest streak in the record if you don't count the first game.

4 Higher Order Functions

1. (★★) You wrote procedures `squares` and `switch`, and saw `pigl-sent`, that `pigl`ed each word in its argument sentence. Generalize this pattern to create a higher-order procedure `every` that applies an arbitrary procedure, given as an argument, to each word of an argument sentence. This procedure is used as follows:

```
> (every square '(1 2 3 4))
(1 4 9 16)

> (every first '(nowhere man))
 (n m)
```

2. (★★) Write `repeated`, a higher order function that takes in two arguments: a procedure and a number. It outputs another procedure that composes the given procedure as many times as specified. For example:

```
> ((repeated 1+ 7) 0)
7

> ((repeated butfirst 2) '(the rain in spain))
(in spain)
```

3. A common problem in computer science is to sort a collection of data. Although you will be learning more about this in CS61B, this problem will introduce you to the basics.

   (a) (★★) Write a procedure `insert` that takes a number and a sentence of numbers sorted in increasing order. `insert` should place the input number in the appropriate position in the sorted input sentence and output the resulting sentence. For example:

   ```
   > (insert 3 '(1 2 3 4 5))
   (1 2 3 3 4 5)
   ```

   (b) (★★) Use `insert` to write a procedure `insertion-sort` that takes a sentence of numbers and sorts them in increasing order. For example:

   ```
   > (insertion-sort '(3 1 4 1 5))
    (1 1 3 4 5)
   ```

   Hint: You may find it helpful to keep track of two sentences: a sentence containing numbers not yet sorted, and a sentence containing numbers that have already been sorted. The analogous real-world problem is sorting a pile of test papers in alphabetical order.

   (c) (★★) Sometimes we would like to sort words in dictionary order; sometimes we would like to sort numbers where all odd numbers come before the even numbers. Generalize the program to accept any ordering by defining a procedure `insertion-sort-general` that takes a sentence **and** a comparator function. The comparator function will take two arguments and output `#t` if the first argument is larger and `#f` if the second argument is larger or if the two arguments are equal. For example:

   ```
   > (define (odd-first num1 num2)
        (cond ((and (even? num1) (even? num2)) (> num1 num2))
              ((and (even? num1) (odd? num2)) #t)
              ((and (odd? num1) (even? num2)) #f)
              (else (> num1 num2))))

   > (insertion-sort-general '(3 1 4 1 5 2) odd-first)
   (1 1 3 5 2 4)
   ```

   Here, the comparator function `odd-first` imposes a new ordering on the set of numbers where the even numbers are always bigger than the odd numbers.

5 Feedback

Now that you're done, please leave me some feedback at the following link regarding how the course is going. This is not worth points, but will give me valuable feedback which in turn improves your course experience. Thanks! `https://spreadsheets.google.com/embeddedform?formkey=dEo3T3NaYVJMaTFCTFVmeDhKakZfQnc6MQ`