MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1997

Problem Set 2

Guessing Games

Issued: Tuesday, February 11

Due: In recitation on Wednesday, February 19 for recitations held at 9, 12, 1 and 2;
and on Friday, February 21 for recitations held at 10 and 11.

Tutorial Preparation: For the week of February 17 (remember that February 18 is a virtual Monday).

Reading Assignment: Sections 1.2 and 1.3 of *SICP.*

Introduction

As a reminder from last time, every homework assignment contains two kinds of assignments:

- *Written problems:* Your solutions to assigned problems should be handed in at recitation, **late work will not be accepted.** Assigned problems include both *pre-laboratory* exercises that should be worked at home and *laboratory exercises* that require interaction with a computer that interprets Scheme. Laboratory exercises should be read and understood *before coming to lab:* your efforts in lab will be more effective if you have a good understanding of what is required and have developed an initial plan for working the lab exercises. It is generally much more efficient to test, debug, and run a program that you have planned before coming into the lab than to try to do the planning "online". The experience of students who have taken 6.001 in previous terms has consistently indicated that failing to prepare ahead for laboratory assignments generally ensures that the assignments take much longer than necessary. Also, it is much to your advantage to start the lab work early, rather than waiting until just before it is due.

- *Tutorial preparation:* In tutorial you will be asked to discuss questions raised in the assignment. Your tutor may choose not to cover every question every week, but you should be prepared to discuss them. You need not write up formal answers to these, other than making notes for yourself, if you choose. These tutorial questions, or ones very similar to them, may also appear on quizzes this semester. Your tutor will also review your written homework and discuss the problems with you.

Tutorial Preparation

Tutorial Exercise 1: Write a recursive procedure that will add the integers from 1 to n, where n is an argument provided to the procedure. Write an iterative procedure to do the same thing.

Tutorial Exercise 2: Write a procedure that will add the square of the integers from 1 to n.

Tutorial Exercise 3: Write a procedure of two arguments, n and a procedure of one argument, f, that will add $f(1), f(2), \ldots, f(n)$.

Tutorial Exercise 4: Do Exercise 1.11 of SICP.

Tutorial Exercise 5: Do Exercise 1.29 of SICP.

Tutorial Exercise 4: Do Exercise 1.30 of SICP.

Procedure Types and Procedure Constructors

Beginning with this assignment, we will start using procedures which may be applied to many different types of arguments and may return different types of values. To keep track of this, it will be helpful to have some simple notation to describe types of Scheme values.

Two basic types of values are Sch-Num, the Scheme numbers such as 3, -4.2, 6.931479453e89, and Sch-Bool, the truth values `#t,#f`. The procedure `square` may be applied to a Sch-Num and will return another Sch-Num. We indicate this with the type notation:

$$\texttt{square} : \text{Sch-Num} \rightarrow \text{Sch-Num}$$

If `f` and `g` are procedures of type Sch-Num $\rightarrow$ Sch-Num, then we may *compose* them:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

Thus, for example `(compose square log)` is the procedure of type Sch-Num $\rightarrow$ Sch-Num that returns the square of the logarithm of its argument, while `(compose log square)` returns the logarithm of the square of its argument:

```
(log 2)
;Value: .6931471805599453

((compose square log) 2)
;Value: .4804530139182014

((compose log square) 2)
;Value: 1.3862943611198906
```

As we have used it above, the procedure `compose` takes as arguments two procedures of type F = Sch-Num $\rightarrow$ Sch-Num, and returns another such procedure. We indicate this with the notation:

$$\texttt{compose} : (F, F) \rightarrow F$$

Exercise 1: To get some practice in writing procedures and reasoning about their effects, write procedures to do each of the following. For each, also state the type of the procedure using the notation above.

- Compute the cube of an integer.
- Compute the product of the integers from 1 to n, where n is an argument to the procedure.
- Compute the product of the cube of the integers from 1 to n, where n is an argument to the procedure.
- Compute the product of $f(1), f(2), \ldots, f(n)$ where f is any appropriate procedure and n is an argument.

Just as squaring a number multiplies the number by itself, `twice` of a function composes the function with itself. That is, `((twice f) n)` will return the same number as `(f(f n))`:

```
(define (twice f)
  (compose f f))

((twice square) 3)
;Value: 81

(square (square 3))
;Value: 81
```

As used above, `twice` is of type $(F \rightarrow F)$. That is, it takes as input a function from numbers to numbers and returns the same kind of function. But `twice` will actually work for other kinds of input functions. It is enough for the input function to have a type of the form $T \rightarrow T$, where T may be any type. So more generally, we can write

$$\texttt{twice} : (T \rightarrow T) \rightarrow (T \rightarrow T)$$

Exercise 2: Composition, like multiplication, may be iterated. To explore this concept, consider the idea of repeatedly applying a procedure, for example:

```
((repeated sin 5) 3.1)
;Value: 4.1532801333692235e-2

(sin(sin(sin(sin(sin 3.1)))))
;Value: 4.1532801333692235e-2
```

Note that `repeated` can be characterized as having the following type:

$$\texttt{repeated} : ((T \to T), \text{Sch-Nonneg-Int}) \to (T \to T)$$

or, if you prefer:

$$\texttt{repeated} : (T', \text{Sch-Nonneg-Int}) \to T'$$

where $T' = (T \to T)$.

Write the procedure `repeated`.

Pre-Lab exercise 3.A The type of `twice` is of the form $(T' \to T')$ (where T' happens to equal $(T \to T)$), so we can legitimately use `twice` as an input to `twice`!

For what value of `n` will `(((twice twice) f) 0)` return the same value[1] as `((repeated f n) 0)`?

See if you can now predict what will happen when the following expressions are evaluated. Briefly explain what goes on in each case.

1. `(((twice twice) (lambda (x) (+ 1 x))) 6)`
2. `(((twice twice) identity) compose)`
3. `(((twice twice) square) 1)`
4. `(((twice twice) square) 2)`.

Lab exercise 3.B When you get to the Lab, test your predictions. (**Warning**: Before you do this, make sure you understand how to *interrupt* a Scheme evaluation.)

Guessing Games

Before the invention of GameBoys (back in the dark ages), taking long car trips with young children was a very painful experience. Without something to distract the children, the trip was frequently punctuated with complaints like "Are we there yet" (usually while still in the driveway!), or "I'm thirsty" (usually within 5 minutes of having stopped for a drink), or "I have to go to the bathroom" (usually within 5 minutes of the previous case) or "He's bothering me" (anytime). To reduce such disturbances, parents often resorted to distractions, such as singing the entire rendition of "99 Bottles of Beer on the Wall", or playing games like "I Spy" or "I know a number".

In this last game, one child secretly chooses an integer in some range, then announces, "I am thinking of a number between n and m" (well, okay, most eight year olds don't use such formal

[1] "Sameness" of procedure values is a sticky issue which we don't want to get into here. We can avoid it by assuming that `f` is bound to a value of type F, so evaluation of `(((twice twice) f) 0)` will return a number.

mathematical jargon – they say something like "I am thinking of a number between 1 and 1000). Each other child then takes turns guessing numbers until someone gets the right answer.

Let's think about how we might build a set of procedures to simulate this simple game.

First, we need to generate the secret number. We can specify a range in which the number will lie by defining some end points:

```
(define low-end 1)
(define high-end 1000)
```

Note that the type of each of these constants is Sch-Int.

We will let the global[2] variable `hidden` of type Sch-Int represent the secret number, and we provide a procedure `make-hidden` of type: $\{\} \rightarrow$ Sch-Int, which means that it takes no arguments as input and produces a result of type integer. Evaluating `(make-hidden)` will cause the the variable `hidden` to take on a new value in the range between `low-end` and `high-end`. Note that we are not providing you with a definition of `make-hidden` – you can simply treat it as a black box abstraction that, upon evaluation, sets the value of the hidden variable.

Our goal is to develop strategies for guessing the number, and to see the effectiveness of different strategies. We will define strategies to be a set of three procedures. The first procedure will be used to generate a guess and will be of type:

$$(\text{Sch-Int}, \text{Sch-Int}) \rightarrow \text{Sch-Int}$$

This notation means that the guess generator will take in two arguments as inputs, both integers, and will produce as output an integer. The inputs denote the strategies current knowledge about the lowest and highest value that the hidden number could be – initially these will be `low-end` and `high-end`. One very simple strategy is a random guesser:

```
(define guess
  (lambda (low high)
    ;; takes inputs defining range in which number could lie
    ;; and generates an integer that lies within that range by
    ;; using the Scheme primitive random
    (+ low (random (+ 1 (- high low))))))
```

Associated with a guess generator will be two other procedures, one which generates a new upper bound on what the hidden number could be, and one which generates a new lower bound. Both of these procedures will be of type:

$$(\text{Sch-Int}, \text{Sch-Int}, \text{Sch-Int}) \rightarrow \text{Sch-Int}$$

and will take as arguments the current lower and upper bound as well as the value of the latest guess.

[2] By global we mean that this variable can be accessed by any procedure – later in the semester we will see a more formal definition of this term

Given a strategy consisting of these three procedures, we can create a procedure to test how long it takes for the strategy to find a hidden value (look this procedure over carefully to see how bounds are generated, how the number of trials is counted, and so on):

```
(define how-many-trials
  (lambda (strategy strategy-low strategy-high low high)
    ;; input is the three procedures that constitute a strategy
    ;; plus the current values of the strategy's knowledge of
    ;; the low and high bounds on the value
    (let ((value (strategy low high)))
      ; get new guess by using strategy
      (cond ((= (test value) 0)  ;; if it is correct
             1)                  ;; then it took one try
            (else (+ 1           ;; otherwise count this try
                                 ;; and try again with a new low
                                 ;; and high bound on the range
                    (how-many-trials strategy strategy-low strategy-high
                                     (strategy-low low high value)
                                     (strategy-high low high value)))))))))

(define test
  (lambda (v)
    ;; this procedure is the only legal way to test a guess,
    ;; it returns 1 if the guess is too high, -1 if too low,
    ;; and 0 if correct
    (cond ((> v hidden) 1)
          ((= v hidden) 0)
          ((< v hidden) -1))))
```

Exercise 4 To complete our random guessing strategy, we need methods for updating the strategies estimate for the upper and lower bounds on the range of possible values. Since this is a purely random strategy, we will treat each guess as entirely independent. Thus our low bound updater should always keep the low bound at the original value, and similary keep the high bound at the original value. Provide definitions for `guess-low` and `guess-high` that implement this simple idea. Turn in a listing of your work.

Exercise 5 Now we can run some trials, using `guess, guess-low` and `guess-high` to see how long this strategy takes. In particular, try evaluating

```
(make-hidden)
```

to set the secret value, then try evaluating

```
(how-many-trials guess guess-low guess-high low-end high-end)
```

to see how many guesses it takes. Try this several times, and record the results – turn in a transcript of your session.

Exercise 6 Now let's be a bit more systematic about our strategy. Suppose instead of randomly guessing, we start at one end of the range, e.g., the low end, and first guess the lowest possible number. Then if that is not right, we move on to the next number, and so on. Define a strategy that implements this, that is, define a guess generator, a low bound updater and a high bound updater that implements this idea. Turn in a listing of your work.

Repeat the testing you did in Exercise 5 for this case.

Exercise 7 Suppose we let n denote the range of possible values, in this case the range from `low-end` to `high-end`. What is the order of growth in time for the strategy you created in Exercise 6 as a function of n?

What about the random strategy we started with? On average, can you estimate the number of steps required as a function of n?

It would be handy to have a way of running multiple trials and gathering statistics on those trials. The following procedure provides a way of measuring the mean (or average) number of guesses a strategy takes, based on some specified number of trials, and the variance in that number (basically how widely spread the numbers are about the mean).

```
(define square (lambda (x) (* x x)))

(define measure-mean-and-variance
  (lambda (no-of-trials strategy strategy-low strategy-high)
    ;; procedures a strategy triple as input, together with a number
    ;; of trials to run.  It iteratively counts the number of trials, the
    ;; running sum of the number of guesses over all trials and the running
    ;; sum of the squares of the number of guesses.  It uses these to
    ;; compute the mean and deviation of the statistics for this strategy
    ;; over this number of trials
    (define helper
      (lambda (n sum sq-sum)  ;; parameters are number of trials,
                              ;; sum of trials, and sum of squares of trials
        (cond ((= n no-of-trials)   ;; if we are done
               (let ((mean (/ sum n)))   ;; create a temporary variable called
                                         ;; mean (see text for details on let)
                 (newline)
                 (display (round mean))  ;; print out mean number of guesses
                 (newline)
                      ;; print out the standard deviation of the guesses
                 (display (round (sqrt (/ (+ sq-sum
                                             (* -2 mean sum)
                                             (* n (square mean)))
                                          (- n 1)))))))
              (else             ;; if not done, then
               (make-hidden)    ;; get a new hidden number
                                ;; and run the strategy -- use let to make
                                ;; a temporary variable to hold the result
               (let ((trial (how-many-trials strategy strategy-low
                                 strategy-high low-end high-end)))
                 ;; iterate to the next trial, with new state variables
                 (helper (+ n 1)
                         (+ sum trial)
                         (+ sq-sum (square trial))))))))
    (helper 0 0 0)))
```

Exercise 8 Using this code, run some trials for the random guess strategy. Use 100 trials as the test. Do this several times, and report the results. Do the same for the systematic strategy you built in Exercise 7.

Now let's look at another strategy for guessing the hidden value. In this case, we want the guess generator to take the current bounds on the range of possible values, and use the middle of that range as the next guess. Be sure that your procedure returns an integer as its result. To update the lower bound, our strategy should use the current lower bound if the guess is too high, but should use our guess if it is too low. Similarly, our strategy should update the upper bound to be the guess if it is too high, but keep the current upper bound if the guess is too low.

Exercise 9 Implement this strategy and test it out. Include in your testing several trials that report mean and variance over sets of 100 trials. Turn in a listing and the results of your test.

Exercise 10 What is the order of growth in time of this approach?

Exercise 11 Once we have a set of possible strategies, we can consider variations on each of them. For example, we could use our random guessing strategy, but use the updating methods from Exercise 9. Try this and report on the results. How different are they from the other methods? What is the best strategy among those we've developed so far?

Exercise 12 Implement some new strategy of your own devising. Try it out and report the results. Do you think you can do better than the best strategy of those used above?