MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2013
Problem Set 4

Issued: Wed. 27 Feb. 2013 **Due: Wed. 6 Mar. 2013**

Reading:
SICP, From Chapter 4: section 4.1.7--4.2 (from PS03)
section 4.3; (pp. 412--437)

Code: utils.scm, ghelper.scm, syntax.scm, rtdata.scm,
load-analyze.scm, analyze.scm, repl.scm
load-amb.scm, analyze-amb.scm repl-amb.scm
multiple-dwelling.scm

Heavy Evaluator Hacking

In this problem set we build interpreters in a different direction. We start with the essential EVAL/APPLY interpreter, written as an analyzer of the syntax into a compiler of compositions of execution procedures -- a small combinator language. We will warm up by making modifications to this evaluator.

Next, we will change the evaluator to include AMB expressions. To add AMB, the execution procedures will all have a different shape: in addition to the environment, each will take two "continuation procedures" SUCCEED and FAIL. In general, when a computation comes up with a value it will invoke SUCCEED with the proposed value and a complaint department which, if invoked, will try to produce an alternate value. If a computation cannot come up with a value, it will invoke the complaint department passed to it in the FAIL continuation.

An important lesson to be learned here is how to use continuation procedures to partially escape the expression structure of the language. By construction, a functional expression has a unique value. However, in the AMB system an expression may be ambiguous as to its value... Think about how we arrange that to make sense!

Separating Syntactic Analysis from Execution (Compiling to Combinators)

It is important to read SICP section 4.1.7 carefully here. When you load "load-analyze.scm" you will get an evaluator similar to the one described in this section.

Problem 4.1: Warmup

It is often valuable to have procedures that can take an indefinite number of arguments. The addition and multiplication procedures in Scheme are examples of such procedures. Traditionally, a user may specify such a procedure in a definition by making the bound-variable specification of a lambda expression a symbol rather than a list of formal parameters. That symbol is expected to be bound to the list of arguments supplied. For example, to make a procedure that takes several arguments and returns a list of the squares of the arguments supplied, one may write:

```
(lambda x (map square x))
```

or

```
(define (ss . x) (map square x))
```

and then

```
(ss 1 2 3 4) ==> (1 4 9 16)
```

Modify the analyzing interpreter to allow this construction.

Hint: you do not need to change the code involving DEFINE or LAMBDA in syntax.scm! This is entirely a change in analyze.scm

Demonstrate that your modification allows this kind of procedure, and that it does not cause other troubles.

Problem 4.2: Infix notation

Many people like infix notation for small arithmetic expressions. It is not hard to write a special form, (INFIX <infix-string>), that takes a character string, parses it as an infix expression with the usual precedence rules, and reduces it to Lisp. Note that to do this you really don't have to delve into the combinator target mechanism of the evaluator, since this can be accomplished as a "macro" in the same way that COND and LET are implemented (see syntax.scm).

So, for example, we should be able to write the program:

```
(define (quadratic a b c)
  (let ((discriminant (infix "b^2-4*a*c")))
    (infix "(-b+sqrt(discriminant))/(2*a)")))
```

Hint: Do not try to parse numbers! That is hard -- let Scheme do it for you: use string->number (see MIT Scheme documentation). Just pass the substring that specifies the number to string->number to get the numerical value.

Write the INFIX special form, install it in the evaluator, and demonstrate that it works.

Please! Unless you have lots of time to burn, do not try to write a complete infix parser for some entire language, like Python (easy) or C++ (hard)! We just want parsing of simple arithmetic expressions.

Also, if you have written an infix parser before, and thus consider this a boring exercise, consider building it out of parser combinators.

AMB and Nondeterministic Programming

Now comes the real fun part of this problem set! Please read section 4.3 of SICP carefully before starting this part. This interpreter requires a change in the interface structure of the combinators that code compiles into, so it is quite different. Of course, our system differs from the one in SICP in that it is implemented with generic extension capability. The loader for the interpreter extended for AMB is "load-amb.scm".

Problem 4.3: Warmup: Programming with AMB

Run the multiple-dwelling program (to get a feeling for how to use the system).

Do exercises 4.38, 4.39, and 4.40 (p. 419) from SICP.

Note: we supply the multiple-dwelling.scm program so you need not type it in.

Problem 4.4: The AMB interpreter

Do exercises 4.51, 4.52, and 4.53 (pp. 436--437) from SICP.
