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 6.1: 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 6.2: 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)")))
```

We have provided (in the code/infix subdirectory) an infix parser that actually implements a complete language, including definitions, LAMBDA, and IF-THEN-ELSE that converts an infix representation of a program into Lisp syntax. For example, if you load "load-parser" into a Scheme you can say:

```
(pp (infix-string->combination
    "fib := lambda n:
      if n == 0
        then 0
      else if n == 1
        then 1
      else fib(n-1) + fib(n-2);
    fib(20)"))
```

#| (begin (define fib
      (lambda (n)
        (if (= n 0)
          0
          (if (= n 1)
            1
            (+ (fib (- n 1))
              (fib (- n 2)))))))
    (fib 20))
#|

The infix parser is seriously more complicated than the EVAL/APPLY interpreter. It is also not written beautifully. However, you can have fun with this code.
a. Write the INFIX special form, install it in the EVAL/APPLY interpreter, and demonstrate that you can use it intermixed with Scheme programs written in ordinary Lisp syntax.

b. Pick some nice construct that you like, from some language that you have used, and modify the infix parser we supplied so that you can use that construct in your Scheme programs. Demonstrate that your foreign construct works as you expected. (My favorite extension is Python set comprehension.)

c. Optional: You can write INFIX as an MIT Scheme macro. This will allow you to use the infix language in the native MIT Scheme system. Do this, if you feel energetic.

----------

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".

Generate and Test

We normally think of generate and test, and its extreme use in search, as an AI technique. However, it can be viewed as a way of making systems that are modular and independently evolvable, as in the exploratory behavior of biological systems. Consider a very simple example: suppose we have to solve a quadratic equation. There are two roots to a quadratic. We could return both, and assume that the user of the solution knows how to deal with that, or we could return one and hope for the best. (The canonical sqrt routine returns the positive square root, even though there are two square roots!) The disadvantage of returning both solutions is that the receiver of that result must know to try the computation with both and either reject one, for good reason, or return both results of the computation, which may itself have made some choices. The disadvantage of returning only one solution is that it may not be the right one for the receiver’s purpose.
Perhaps a better way to handle this is to build a backtracking mechanism into the infrastructure. The square-root procedure should return one of the roots, with the option to change its mind and return the other one if the first choice is determined to be inappropriate by the receiver. It is, and should be, the receiver’s responsibility to determine if the ingredients to its computation are appropriate and acceptable. This may itself require a complex computation, involving choices whose consequences may not be apparent without further computation, so the process is recursive. Of course, this gets us into potentially deadly exponential searches through all possible assignments to all the choices that have been made in the program. As usual, modular flexibility can be dangerous.

Linguistically Implicit Search

It is important to consider the extent to which a search strategy can be separated from the other parts of a program, so that one can interchange search strategies without greatly modifying the program. In this problem set we take the further step of pushing search and search control into the infrastructure that is supported by the language, without explicitly building search into our program at all.

This idea has considerable history. In 1961 John McCarthy had the idea of a nondeterministic operator AMB, which could be useful for representing nondeterministic automata. In 1967 Bob Floyd had the idea of building backtracking search into a computer language as part of the linguistic glue. In 1969 Carl Hewitt proposed a language, PLANNER, that embodied these ideas. In the early 1970s Colmerauer, Kowalski, Roussel, and Warren developed Prolog, a language based on a limited form of first-order predicate calculus, which made backtracking search implicit.
Problem 6.3: 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 6.4: Formalize and solve the following puzzle using AMB:

Six people, two women and four men are seated at a round table, playing cards. Each has a hand; no two of the hands are equally strong.

Ben is seated opposite Eva.
The man at Alyssa’s right has a stronger hand than Jake has.
The man at Eva’s right has a stronger hand than Ben has.
The man at Ben’s right has a stronger hand than Fred has.
The man at Eva’s right has a stronger hand than Eva has.
The woman at Jake’s right has a stronger hand than Fred has.
The woman at Fred’s right has a stronger hand than Bill has.

What is the arrangement at the table? Is it unique up to rotation of the table?

Use AMB to specify the alternatives that are possible for each choice. Also determine how many solutions there are if we are not told that "The man at Ben’s right has a stronger hand than Fred has," but rather that "The man on Ben’s right is not Fred." Explain this result.

Note: The most straightforward solution is slow: It takes a few hours on my laptop. However, there is a clever solution that converges in only about 2 minutes.

Problem 6.5: The AMB interpreter

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