By Joshua Cantrell
jjc@cory.berkeley.edu

Nondeterministic Evaluator

How Amb Works...

To make the evaluator nondeterministic the special form *amb*[1] is added. *Amb* is given a list of expressions that represent the possible values of the *amb* expression. Any of the values can be chosen, but they don't all have to make sense given the restrictions we apply in our program. Because *amb* chooses a guess whenever it is called, if this guess is discovered to be invalid, it must have some mechanism to continuously try other possible values until it finds one that is valid, or is forced to give up. Once it fails, it falls back to the previous *amb* to try a new value[2]. It is also useful to know that *amb* delays the evaluation of its expressions until it needs the value, so you can make infinite sets using *amb*[3].

`(amb <`e_1`> <`e_2`> ... <`e_n`>)` - Defines a set of possible resultant values of the expression.
`(amb)` - The set of no possible values, which always fails.

How Require Works...

The *require* procedure is a simple application of *amb*[4]. It's written as shown:

```
(define (require p)
  (if (not p) (amb)))
```

By looking at our definition of *amb*, we can see that if the predicate *p* is not satisfied, *(amb)* is evaluated. This causes an immediate failure, and the previously evaluated *amb* is forced to try a new value, or else fail if it has already tried all of it's values.

Keeping track of previously evaluated *amb*s!

A first step of seeing how *amb* works is to take a simple procedure and actually trace through it. In order to trace through the procedure, we need to know how and where to backtrack when an *amb* fails! This can be difficult if the proper information isn't kept available during tracing the evaluation by hand. The procedure I propose to trace is based on the *prime-sum-pair* example given on page 412 of SICP.

```
(define (prime-sum-pair set1 set2)
  (require (prime? (+ a b)))
  (list a b))
```

---

[1] *Amb* for ambiguously defined.
[2] This is the way it's defined in our nondeterministic evaluator. It's called *depth-first search* which blindly picks values delving deeper into a tree like structure until it runs out of possibilities, and then backs up until it can go further forward. Another name for it is the *greedy algorithm*, because it doesn't take into account hopeless paths.
[3] The *depth-first search* technique used by our evaluator makes infinite *amb* sets impractical because if the last *amb* evaluated has an infinite number of values, it will be trying new values from that set forever without ever trying any other values from previous *amb* expressions.
[4] An *if* special form with only an alternative case is not an error. An unspecified value is returned if the consequent case occures.

By Joshua Cantrell
jjc@cory.berkeley.edu

The major difference between the book's example and mine is that the book's takes lists instead of sets created by the *amb* special form. The effects are the same except the arguments are of a different type.

Given the expression to evaluate:

```
(prime-sum-pair (amb 1 3 5 8) (amb 20 35 110))
```

Step 1:
Evaluating the first amb expression, we choose the first value in the amb, remove it, then take a snap shot of the environment in case we have to return to try a new value.

```
                 Current Evaluation
           (prime-sum-pair 1 (amb 20 35 110))
             Saved Backtrack Expression
      (prime-sum-pair (amb 3 5 8) (amb 20 35 110))
```

Step 2:
Evaluating the second amb expression, we choose the first value in the amb, remove it, then take a snap shot of the environment in case we have to return to try a new value.

```
                 Current Evaluation
                (prime-sum-pair 1 20)
             Saved Backtrack Expression
           (prime-sum-pair 1 (amb 35 110))
```

Step 3:
Evaluation continues as normal until we get to the require procedure...

```
                 Current Evaluation
            (require (prime? (+ 1 20)))
                         ↓
               (require (prime? 21))
                         ↓
                   (require #f)
                         ↓
                (if (not #f) (amb))
                         ↓
                   (if #t (amb))
```

Step 4:
This amb function is devoid of values, so it immediately fails. This means we need to go back to the last evaluated amb in Step 2 and try again using the saved backtrack expression.

```
                 Current Evaluation
                       (amb)
                         ↓
           (prime-sum-pair 1 (amb 35 110))
```

By Joshua Cantrell
jjc@cory.berkeley.edu

Step 5:

Current Evaluation
```
(prime-sum-pair 1 35)
```
Saved Backtrack Expression
```
(prime-sum-pair 1 (amb 110))
```

Step 6:

Current Evaluation
```
(require (prime? (+ 1 35)))
            ↓
   (require (prime? 36))
            ↓
       (require #f)
            ↓
   (if (not #f) (amb))
            ↓
      (if #t (amb))
```

Step 7:
Must go back to the saved backtrack expression of Step 5.

Current Evaluation
```
           (amb)
             ↓
(prime-sum-pair 1 (amb 110))
```

Step 8:

Current Evaluation
```
(prime-sum-pair 1 110)
```
Saved Backtrack Expression
```
(prime-sum-pair 1 (amb))
```

Step 9:

Current Evaluation
```
(require (prime? (+ 1 110)))
            ↓
   (require (prime? 111))
            ↓
       (require #f)
            ↓
   (if (not #f) (amb))
            ↓
      (if #t (amb))
```

Step 10:
Must go back to the saved backtrack expression of Step 8.

Current Evaluation
```
         (amb)
           ↓
(prime-sum-pair 1 (amb))
```

Step 11:
This amb expression has exhausted all of its possibilites and fails. Must go back to the saved backtrack expression of Step 1

Current Evaluation
```
                    (amb)
                      ↓
(prime-sum-pair (amb 3 5 8) (amb 20 35 110))
```

By Joshua Cantrell
jjc@cory.berkeley.edu

Step 12:

Current Evaluation
```
(prime-sum-pair 3 (amb 20 35 110))
```
Saved Backtrack Expression
```
(prime-sum-pair (amb 5 8) (amb 20 35 110))
```

Step 13:

Current Evaluation
```
(prime-sum-pair 3 20)
```
Saved Backtrack Expression
```
(prime-sum-pair 3 (amb 35 110))
```

Step 14:

Current Evaluation
```
(require (prime? (+ 3 20)))
            ↓
(require (prime? 23))
            ↓
(require #t)
            ↓
(if (not #t) (amb))
            ↓
(if #f (amb))
            ↓
#<unspecified>
```

Step 15:
Passed the require procedure, so we move on to the next expression in the procedure body.

Current Evaluation
```
(list 3 20)
     ↓
(3 20)
```

Done: **`(3 20)`**

As an added feature of the evaluator, we can force the successful evaluation to act as though it failed and it will compute another answer by continuing from the last "saved backtrack expression". The command that allows us to do this is *try-again*.

Amb's Equivalent in Mathematics

To help understand how the *amb* special form works, we can relate it to sets in mathematics. When we say, "*a* is an element of the set containing 1, 2, 3, and 4," or equivalently $a \in \{1, 2, 3, 4\}$, it's the same as `(define a (amb 1 2 3 4))`. It's clear that all three statements are equivalent because they don't specify the value of *a* except by saying it's one of the following. What about *amb*s within *amb*s? `(define x (amb 1 2 (amb 3 4))` can be equated with $x \in \{1, 2, y \in \{3, 4\}\}$. Notice that both expressions can be easily expanded to be the definition of *a*.

By Joshua Cantrell
jjc@cory.berkeley.edu

Implementing the Evaluator

SICP decided to build the evaluator off of the *Analyzing Scheme MCE* because the differences between the code is found in the analysis procedures and what arguments are needed to evaluate the analyzed expressions. The new basic form is:

Analyzer's Form: *`(analyze-something <expression>)`*
Analyzer Result's Form: *`(analyzed-expression <env> <succeed> <fail>)`*

The arguments *succeed* and *fail* are both procedures with the following structures:

Succeed: *(lambda (<value> <fail>) <body>)*
Fail: *(lambda () <body>)*

Succeed is the procedure that is used if the resultant value is a success. The *value* is the successful value, and *fail* is the procedure meant to be called if there is a failure in the future (normally with the necessity of backtracking). *Fail* is the procedure used to store the information needed for backtracking.

There are a limited number of locations where we need to backtrack to when a failure occurs. These will be the places where we define the *fail* procedure to give us the means to properly backtrack. The summary of these places is given on page 428 of SICP:

- ***amb* expressions:** to provide a mechanism to make alternative choices if the current choice made by the *amb* expression leads to a dead end.
- **the top-level driver:** to provide a mechanism to report failure when the choices are exhausted.
- **assignments:** to intercept failures and undo assignments during backtracking.

There are also a limited number of places where the *fail* procedures will need to be evaluated. These are also summarized on page 428 of SICP:

- If the user program executes *(amb)*.
- If the user types *try-again* at the top-level driver.
- When an assignment is undone and further backing out must occur from that point.
- When an *amb* expression runs out of new values to choose from.

For more depth and the code required to implement this feature, I direct you to pages 426-436 of SICP.