[Go to first, previous, next page; contents; index]

4.3 Variations on a Scheme -- Nondeterministic Computing

In this section, we extend the Scheme evaluator to support a programming paradigm called nondeterministic computing by building into the evaluator a facility to support automatic search. This is a much more profound change to the language than the introduction of lazy evaluation in section 4.2.

Nondeterministic computing, like stream processing, is useful for ``generate and test'' applications. Consider the task of starting with two lists of positive integers and finding a pair of integers -- one from the first list and one from the second list -- whose sum is prime. We saw how to handle this with finite sequence operations in section 2.2.3 and with infinite streams in section 3.5.3. Our approach was to generate the sequence of all possible pairs and filter these to select the pairs whose sum is prime. Whether we actually generate the entire sequence of pairs first as in chapter 2, or interleave the generating and filtering as in chapter 3, is immaterial to the essential image of how the computation is organized.

The nondeterministic approach evokes a different image. Imagine simply that we choose (in some way) a number from the first list and a number from the second list and require (using some mechanism) that their sum be prime. This is expressed by following procedure:

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

It might seem as if this procedure merely restates the problem, rather than specifying a way to solve it. Nevertheless, this is a legitimate nondeterministic program.[42]

The key idea here is that expressions in a nondeterministic language can have more than one possible value. For instance, `an-element-of` might return any element of the given list. Our nondeterministic program evaluator will work by automatically choosing a possible value and keeping track of the choice. If a subsequent requirement is not met, the evaluator will try a different choice, and it will keep trying new choices until the evaluation succeeds, or until we run out of choices. Just as the lazy evaluator freed the programmer from the details of how values are delayed and forced, the nondeterministic program evaluator will free the programmer from the details of how choices are made.

It is instructive to contrast the different images of time evoked by nondeterministic evaluation and stream processing. Stream processing uses lazy evaluation to decouple the time when the stream of possible answers is assembled from the time when the actual stream elements are produced. The

evaluator supports the illusion that all the possible answers are laid out before us in a timeless sequence. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Some of the possible worlds lead to dead ends, while others have useful values. The nondeterministic program evaluator supports the illusion that time branches, and that our programs have different possible execution histories. When we reach a dead end, we can revisit a previous choice point and proceed along a different branch.

The nondeterministic program evaluator implemented below is called the `amb` evaluator because it is based on a new special form called `amb`. We can type the above definition of `prime-sum-pair` at the `amb` evaluator driver loop (along with definitions of `prime?`, `an-element-of`, and `require`) and run the procedure as follows:

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)
```

The value returned was obtained after the evaluator repeatedly chose elements from each of the lists, until a successful choice was made.

Section 4.3.1 introduces `amb` and explains how it supports nondeterminism through the evaluator's automatic search mechanism. Section 4.3.2 presents examples of nondeterministic programs, and section 4.3.3 gives the details of how to implement the `amb` evaluator by modifying the ordinary Scheme evaluator.

4.3.1 Amb and Search

To extend Scheme to support nondeterminism, we introduce a new special form called `amb`.[43] The expression `(amb` $\langle e_1 \rangle$ $\langle e_2 \rangle$ `...` $\langle e_n \rangle$`)` returns the value of one of the n expressions $\langle e_i \rangle$ ``ambiguously.'' For example, the expression

```
(list (amb 1 2 3) (amb 'a 'b))
```

can have six possible values:

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

`Amb` with a single choice produces an ordinary (single) value.

`Amb` with no choices -- the expression `(amb)` -- is an expression with no acceptable values. Operationally, we can think of `(amb)` as an expression that when evaluated causes the computation to ``fail'': The computation aborts and no value is produced. Using this idea, we can express the requirement that a particular predicate expression `p` must be true as follows:

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

With `amb` and `require`, we can implement the `an-element-of` procedure used above:

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

`An-element-of` fails if the list is empty. Otherwise it ambiguously returns either the first element of the list or an element chosen from the rest of the list.

We can also express infinite ranges of choices. The following procedure potentially returns any integer greater than or equal to some given n:

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

This is like the stream procedure `integers-starting-from` described in section 3.5.2, but with an important difference: The stream procedure returns an object that represents the sequence of all integers beginning with n, whereas the `amb` procedure returns a single integer.[44]

Abstractly, we can imagine that evaluating an `amb` expression causes time to split into branches, where the computation continues on each branch with one of the possible values of the expression. We say that `amb` represents a nondeterministic choice point. If we had a machine with a sufficient number of processors that could be dynamically allocated, we could implement the search in a straightforward way. Execution would proceed as in a sequential machine, until an `amb` expression is encountered. At this point, more processors would be allocated and initialized to continue all of the parallel executions implied by the choice. Each processor would proceed sequentially as if it were the only choice, until it either terminates by encountering a failure, or it further subdivides, or it finishes.[45]

On the other hand, if we have a machine that can execute only one process (or a few concurrent processes), we must consider the alternatives sequentially. One could imagine modifying an evaluator to pick at random a branch to follow whenever it encounters a choice point. Random choice, however, can easily lead to failing values. We might try running the evaluator over and over, making random choices and hoping to find a non-failing value, but it is better to systematically search all possible execution paths. The `amb` evaluator that we will develop and work with in this section implements a systematic search as follows: When the evaluator encounters an application of `amb`, it initially selects the first alternative. This selection may itself lead to a further choice. The evaluator will always initially choose the first alternative at each choice point. If a choice results in a failure, then the evaluator automagically[46] backtracks to the most recent choice point and tries the next alternative. If it runs out of alternatives at any choice point, the evaluator will back up to the previous choice point and resume from there. This process leads to a search strategy known as depth-first search or chronological backtracking.[47]

Driver loop

The driver loop for the `amb` evaluator has some unusual properties. It reads an expression and prints the value of the first non-failing execution, as in the `prime-sum-pair` example shown above. If we want to see the value of the next successful execution, we can ask the interpreter to backtrack and attempt to generate a second non-failing execution. This is signaled by typing the symbol `try-again`. If any expression except `try-again` is given, the interpreter will start a new problem,

discarding the unexplored alternatives in the previous problem. Here is a sample interaction:

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
(3 110)
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
(8 35)
;;; Amb-Eval input:
try-again
;;; There are no more values of
(prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))
;;; Amb-Eval input:
(prime-sum-pair '(19 27 30) '(11 36 58))
;;; Starting a new problem
;;; Amb-Eval value:
(30 11)
```

Exercise 4.35. Write a procedure `an-integer-between` that returns an integer between two given bounds. This can be used to implement a procedure that finds Pythagorean triples, i.e., triples of integers (i,j,k) between the given bounds such that $i \leq j$ and $i^2 + j^2 = k^2$, as follows:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

Exercise 4.36. Exercise 3.69 discussed how to generate the stream of all Pythagorean triples, with no upper bound on the size of the integers to be searched. Explain why simply replacing `an-integer-between` by `an-integer-starting-from` in the procedure in exercise 4.35 is not an adequate way to generate arbitrary Pythagorean triples. Write a procedure that actually will accomplish this. (That is, write a procedure for which repeatedly typing `try-again` would in principle eventually generate all Pythagorean triples.)

Exercise 4.37. Ben Bitdiddle claims that the following method for generating Pythagorean triples is more efficient than the one in exercise 4.35. Is he correct? (Hint: Consider the number of possibilities that must be explored.)

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high))
        (hsq (* high high)))
    (let ((j (an-integer-between i high)))
      (let ((ksq (+ (* i i) (* j j))))
        (require (>= hsq ksq))
        (let ((k (sqrt ksq)))
          (require (integer? k))
          (list i j k))))))
```

4.3.2 Examples of Nondeterministic Programs

Section 4.3.3 describes the implementation of the `amb` evaluator. First, however, we give some examples of how it can be used. The advantage of nondeterministic programming is that we can suppress the details of how search is carried out, thereby expressing our programs at a higher level of abstraction.

Logic Puzzles

The following puzzle (taken from Dinesman 1968) is typical of a large class of simple logic puzzles:

> Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

We can determine who lives on each floor in a straightforward way by enumerating all the possibilities and imposing the given restrictions:[48]

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Evaluating the expression `(multiple-dwelling)` produces the result

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

Although this simple procedure works, it is very slow. Exercises 4.39 and 4.40 discuss some possible improvements.

Exercise 4.38. Modify the multiple-dwelling procedure to omit the requirement that Smith and Fletcher do not live on adjacent floors. How many solutions are there to this modified puzzle?

Exercise 4.39. Does the order of the restrictions in the multiple-dwelling procedure affect the answer? Does it affect the time to find an answer? If you think it matters, demonstrate a faster program obtained from the given one by reordering the restrictions. If you think it does not matter, argue your case.

Exercise 4.40. In the multiple dwelling problem, how many sets of assignments are there of people to floors, both before and after the requirement that floor assignments be distinct? It is very inefficient to generate all possible assignments of people to floors and then leave it to backtracking to eliminate them. For example, most of the restrictions depend on only one or two of the person-floor variables, and can thus be imposed before floors have been selected for all the people. Write and demonstrate a much more efficient nondeterministic procedure that solves this problem based upon generating only those possibilities that are not already ruled out by previous restrictions. (Hint: This will require a nest of `let` expressions.)

Exercise 4.41. Write an ordinary Scheme program to solve the multiple dwelling puzzle.

Exercise 4.42. Solve the following ``Liars'' puzzle (from Phillips 1934):

> Five schoolgirls sat for an examination. Their parents -- so they thought -- showed an undue degree of interest in the result. They therefore agreed that, in writing home about the examination, each girl should make one true statement and one untrue one. The following are the relevant passages from their letters:
>
> - Betty: ``Kitty was second in the examination. I was only third.''
> - Ethel: ``You'll be glad to hear that I was on top. Joan was second.''
> - Joan: ``I was third, and poor old Ethel was bottom.''
> - Kitty: ``I came out second. Mary was only fourth.''
> - Mary: ``I was fourth. Top place was taken by Betty.''
>
> What in fact was the order in which the five girls were placed?

Exercise 4.43. Use the `amb` evaluator to solve the following puzzle:[49]

> Mary Ann Moore's father has a yacht and so has each of his four friends: Colonel Downing, Mr. Hall, Sir Barnacle Hood, and Dr. Parker. Each of the five also has one daughter and each has named his yacht after a daughter of one of the others. Sir Barnacle's yacht is the Gabrielle, Mr. Moore owns the Lorna; Mr. Hall the Rosalind. The Melissa, owned by Colonel Downing, is named after Sir Barnacle's daughter. Gabrielle's father owns the yacht that is named after Dr. Parker's daughter. Who is Lorna's father?

Try to write the program so that it runs efficiently (see exercise 4.40). Also determine how many solutions there are if we are not told that Mary Ann's last name is Moore.

Exercise 4.44. Exercise 2.42 described the ``eight-queens puzzle'' of placing queens on a chessboard so that no two attack each other. Write a nondeterministic program to solve this puzzle.

Parsing natural language

Programs designed to accept natural language as input usually start by attempting to parse the input, that is, to match the input against some grammatical structure. For example, we might try to recognize simple sentences consisting of an article followed by a noun followed by a verb, such as ``The cat eats.'' To accomplish such an analysis, we must be able to identify the parts of speech of individual words. We could start with some lists that classify various words:[50]

```
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
```

We also need a grammar, that is, a set of rules describing how grammatical elements are composed from simpler elements. A very simple grammar might stipulate that a sentence always consists of two pieces -- a noun phrase followed by a verb -- and that a noun phrase consists of an article followed by a noun. With this grammar, the sentence ``The cat eats'' is parsed as follows:

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))
```

We can generate such a parse with a simple program that has separate procedures for each of the grammatical rules. To parse a sentence, we identify its two constituent pieces and return a list of these two elements, tagged with the symbol `sentence`:

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

A noun phrase, similarly, is parsed by finding an article followed by a noun:

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

At the lowest level, parsing boils down to repeatedly checking that the next unparsed word is a member of the list of words for the required part of speech. To implement this, we maintain a global variable `*unparsed*`, which is the input that has not yet been parsed. Each time we check a word, we require that `*unparsed*` must be non-empty and that it should begin with a word from the designated list. If so, we remove that word from `*unparsed*` and return the word together with its part of speech (which is found at the head of the list):[51]

```
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))
```

To start the parsing, all we need to do is set `*unparsed*` to be the entire input, try to parse a sentence, and check that nothing is left over:

```
(define *unparsed* '())
(define (parse input)
```

```
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))
```

We can now try the parser and verify that it works for our simple test sentence:

```
;;; Amb-Eval input:
(parse '(the cat eats))
;;; Starting a new problem
;;; Amb-Eval value:
(sentence (noun-phrase (article the) (noun cat)) (verb eats))
```

The `amb` evaluator is useful here because it is convenient to express the parsing constraints with the aid of `require`. Automatic search and backtracking really pay off, however, when we consider more complex grammars where there are choices for how the units can be decomposed.

Let's add to our grammar a list of prepositions:

```
(define prepositions '(prep for to in by with))
```

and define a prepositional phrase (e.g., ``for the cat'') to be a preposition followed by a noun phrase:

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
        (parse-word prepositions)
        (parse-noun-phrase)))
```

Now we can define a sentence to be a noun phrase followed by a verb phrase, where a verb phrase can be either a verb or a verb phrase extended by a prepositional phrase:[52]

```
(define (parse-sentence)
  (list 'sentence
         (parse-noun-phrase)
         (parse-verb-phrase)))
(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                             verb-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
```

While we're at it, we can also elaborate the definition of noun phrases to permit such things as ``a cat in the class.'' What we used to call a noun phrase, we'll now call a simple noun phrase, and a noun phrase will now be either a simple noun phrase or a noun phrase extended by a prepositional phrase:

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))
(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
                             noun-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
```

Our new grammar lets us parse more complex sentences. For example

```
(parse '(the student with the cat sleeps in the class))
```

produces

```
(sentence
 (noun-phrase
  (simple-noun-phrase (article the) (noun student))
  (prep-phrase (prep with)
               (simple-noun-phrase
                (article the) (noun cat))))
 (verb-phrase
  (verb sleeps)
  (prep-phrase (prep in)
               (simple-noun-phrase
                (article the) (noun class)))))
```

Observe that a given input may have more than one legal parse. In the sentence ``The professor lectures to the student with the cat,'' it may be that the professor is lecturing with the cat, or that the student has the cat. Our nondeterministic program finds both possibilities:

```
(parse '(the professor lectures to the student with the cat))
```

produces

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb-phrase
   (verb lectures)
   (prep-phrase (prep to)
                (simple-noun-phrase
                 (article the) (noun student))))
  (prep-phrase (prep with)
               (simple-noun-phrase
                (article the) (noun cat)))))
```

Asking the evaluator to try again yields

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb lectures)
  (prep-phrase (prep to)
               (noun-phrase
                (simple-noun-phrase
                 (article the) (noun student))
                (prep-phrase (prep with)
                             (simple-noun-phrase
                              (article the) (noun cat)))))))
```

Exercise 4.45. With the grammar given above, the following sentence can be parsed in five different ways: ``The professor lectures to the student in the class with the cat.'' Give the five parses and explain the differences in shades of meaning among them.

Exercise 4.46. The evaluators in sections 4.1 and 4.2 do not determine what order operands are evaluated in. We will see that the `amb` evaluator evaluates them from left to right. Explain why our parsing program wouldn't work if the operands were evaluated in some other order.

Exercise 4.47. Louis Reasoner suggests that, since a verb phrase is either a verb or a verb phrase followed by a prepositional phrase, it would be much more straightforward to define the procedure `parse-verb-phrase` as follows (and similarly for noun phrases):

```
(define (parse-verb-phrase)
  (amb (parse-word verbs)
       (list 'verb-phrase
             (parse-verb-phrase)
             (parse-prepositional-phrase))))
```

Does this work? Does the program's behavior change if we interchange the order of expressions in the `amb`?

Exercise 4.48. Extend the grammar given above to handle more complex sentences. For example, you could extend noun phrases and verb phrases to include adjectives and adverbs, or you could handle compound sentences.[53]

Exercise 4.49. Alyssa P. Hacker is more interested in generating interesting sentences than in parsing them. She reasons that by simply changing the procedure `parse-word` so that it ignores the ``input sentence'' and instead always succeeds and generates an appropriate word, we can use the programs we had built for parsing to do generation instead. Implement Alyssa's idea, and show the first half-dozen or so sentences generated.[54]

4.3.3 Implementing the `Amb` Evaluator

The evaluation of an ordinary Scheme expression may return a value, may never terminate, or may signal an error. In nondeterministic Scheme the evaluation of an expression may in addition result in the discovery of a dead end, in which case evaluation must backtrack to a previous choice point. The interpretation of nondeterministic Scheme is complicated by this extra case.

We will construct the `amb` evaluator for nondeterministic Scheme by modifying the analyzing evaluator of section 4.1.7.[55] As in the analyzing evaluator, evaluation of an expression is accomplished by calling an execution procedure produced by analysis of that expression. The difference between the interpretation of ordinary Scheme and the interpretation of nondeterministic Scheme will be entirely in the execution procedures.

Execution procedures and continuations

Recall that the execution procedures for the ordinary evaluator take one argument: the environment of execution. In contrast, the execution procedures in the `amb` evaluator take three arguments: the environment, and two procedures called continuation procedures. The evaluation of an expression will finish by calling one of these two continuations: If the evaluation results in a value, the success continuation is called with that value; if the evaluation results in the discovery of a dead end, the failure continuation is called. Constructing and calling appropriate continuations is the mechanism by which the nondeterministic evaluator implements backtracking.

It is the job of the success continuation to receive a value and proceed with the computation. Along with that value, the success continuation is passed another failure continuation, which is to be called subsequently if the use of that value leads to a dead end.

It is the job of the failure continuation to try another branch of the nondeterministic process. The essence of the nondeterministic language is in the fact that expressions may represent choices among alternatives. The evaluation of such an expression must proceed with one of the indicated alternative choices, even though it is not known in advance which choices will lead to acceptable results. To deal with this, the evaluator picks one of the alternatives and passes this value to the success continuation. Together with this value, the evaluator constructs and passes along a failure continuation that can be called later to choose a different alternative.

A failure is triggered during evaluation (that is, a failure continuation is called) when a user program explicitly rejects the current line of attack (for example, a call to `require` may result in execution of `(amb)`, an expression that always fails -- see section 4.3.1). The failure continuation in hand at that point will cause the most recent choice point to choose another alternative. If there are no more alternatives to be considered at that choice point, a failure at an earlier choice point is triggered, and so on. Failure continuations are also invoked by the driver loop in response to a `try-again` request, to find another value of the expression.

In addition, if a side-effect operation (such as assignment to a variable) occurs on a branch of the process resulting from a choice, it may be necessary, when the process finds a dead end, to undo the side effect before making a new choice. This is accomplished by having the side-effect operation produce a failure continuation that undoes the side effect and propagates the failure.

In summary, failure continuations are constructed by

- `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.

Failures are initiated only when a dead end is encountered. This occurs

- if the user program executes `(amb)`;
- if the user types `try-again` at the top-level driver.

Failure continuations are also called during processing of a failure:

- When the failure continuation created by an assignment finishes undoing a side effect, it calls the failure continuation it intercepted, in order to

propagate the failure back to the choice point that led to this assignment or to the top level.

- When the failure continuation for an `amb` runs out of choices, it calls the failure continuation that was originally given to the `amb`, in order to propagate the failure back to the previous choice point or to the top level.

Structure of the evaluator

The syntax- and data-representation procedures for the `amb` evaluator, and also the basic `analyze` procedure, are identical to those in the evaluator of section 4.1.7, except for the fact that we need additional syntax procedures to recognize the `amb` special form:[56]

```
(define (amb? exp) (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))
```

We must also add to the dispatch in `analyze` a clause that will recognize this special form and generate an appropriate execution procedure:

```
((amb? exp) (analyze-amb exp))
```

The top-level procedure `ambeval` (similar to the version of `eval` given in section 4.1.7) analyzes the given expression and applies the resulting execution procedure to the given environment, together with two given continuations:

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

A success continuation is a procedure of two arguments: the value just obtained and another failure continuation to be used if that value leads to a subsequent failure. A failure continuation is a procedure of no arguments. So the general form of an execution procedure is

```
(lambda (env succeed fail)
  ;; succeed is (lambda (value fail) ...)
  ;; fail is (lambda () ...)
  ...)
```

For example, executing

```
(ambeval <exp>
         the-global-environment
         (lambda (value fail) value)
         (lambda () 'failed))
```

will attempt to evaluate the given expression and will return either the expression's value (if the evaluation succeeds) or the symbol `failed` (if the evaluation fails). The call to `ambeval` in the driver loop shown below uses much more complicated continuation procedures, which continue the loop and support the `try-again` request.

Most of the complexity of the `amb` evaluator results from the mechanics of passing the continuations around as the execution procedures call each other. In going through the following code, you should compare each of the execution

procedures with the corresponding procedure for the ordinary evaluator given in section 4.1.7.

Simple expressions

The execution procedures for the simplest kinds of expressions are essentially the same as those for the ordinary evaluator, except for the need to manage the continuations. The execution procedures simply succeed with the value of the expression, passing along the failure continuation that was passed to them.

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env succeed fail)
      (succeed qval fail))))
(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env)
             fail)))
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env)
               fail))))
```

Notice that looking up a variable always ``succeeds.'' If `lookup-variable-value` fails to find the variable, it signals an error, as usual. Such a ``failure'' indicates a program bug -- a reference to an unbound variable; it is not an indication that we should try another nondeterministic choice instead of the one that is currently being tried.

Conditionals and sequences

Conditionals are also handled in a similar way as in the ordinary evaluator. The execution procedure generated by `analyze-if` invokes the predicate execution procedure `pproc` with a success continuation that checks whether the predicate value is true and goes on to execute either the consequent or the alternative. If the execution of `pproc` fails, the original failure continuation for the `if` expression is called.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
             ;; success continuation for evaluating the predicate
             ;; to obtain pred-value
             (lambda (pred-value fail2)
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             ;; failure continuation for evaluating the predicate
             fail))))
```

Sequences are also handled in the same way as in the previous evaluator, except for the machinations in the subprocedure `sequentially` that are required for passing

the continuations. Namely, to sequentially execute `a` and then `b`, we call `a` with a success continuation that calls `b`.

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
         ;; success continuation for calling a
         (lambda (a-value fail2)
           (b env succeed fail2))
         ;; failure continuation for calling a
         fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

Definitions and assignments

Definitions are another case where we must go to some trouble to manage the continuations, because it is necessary to evaluate the definition-value expression before actually defining the new variable. To accomplish this, the definition-value execution procedure `vproc` is called with the environment, a success continuation, and the failure continuation. If the execution of `vproc` succeeds, obtaining a value `val` for the defined variable, the variable is defined and the success is propagated:

```
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)
               (define-variable! var val env)
               (succeed 'ok fail2))
             fail))))
```

Assignments are more interesting. This is the first place where we really use the continuations, rather than just passing them around. The execution procedure for assignments starts out like the one for definitions. It first attempts to obtain the new value to be assigned to the variable. If this evaluation of `vproc` fails, the assignment fails.

If `vproc` succeeds, however, and we go on to make the assignment, we must consider the possibility that this branch of the computation might later fail, which will require us to backtrack out of the assignment. Thus, we must arrange to undo the assignment as part of the backtracking process.[57]

This is accomplished by giving `vproc` a success continuation (marked with the comment ``*1*'' below) that saves the old value of the variable before assigning the new value to the variable and proceeding from the assignment. The failure continuation that is passed along with the value of the assignment (marked with the comment ``*2*'' below) restores the old value of the variable before continuing the failure. That is, a successful assignment provides a failure

continuation that will intercept a subsequent failure; whatever failure would otherwise have called `fail2` calls this procedure instead, to undo the assignment before actually calling `fail2`.

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)        ; *1*
               (let ((old-value
                      (lookup-variable-value var env)))
                 (set-variable-value! var val env)
                 (succeed 'ok
                          (lambda ()    ; *2*
                            (set-variable-value! var
                                                 old-value
                                                 env)
                            (fail2)))))
             fail))))
```

Procedure applications

The execution procedure for applications contains no new ideas except for the technical complexity of managing the continuations. This complexity arises in `analyze-application`, due to the need to keep track of the success and failure continuations as we evaluate the operands. We use a procedure `get-args` to evaluate the list of operands, rather than a simple `map` as in the ordinary evaluator.

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail2)
               (get-args aprocs
                         env
                         (lambda (args fail3)
                           (execute-application
                            proc args succeed fail3))
                         fail2))
             fail))))
```

In `get-args`, notice how `cdr`ing down the list of `aproc` execution procedures and `cons`ing up the resulting list of `args` is accomplished by calling each `aproc` in the list with a success continuation that recursively calls `get-args`. Each of these recursive calls to `get-args` has a success continuation whose value is the `cons` of the newly obtained argument onto the list of accumulated arguments:

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
                    ;; success continuation for this aproc
                    (lambda (arg fail2)
                      (get-args (cdr aprocs)
                                env
                                ;; success continuation for recursive
                                ;; call to get-args
                                (lambda (args fail3)
                                  (succeed (cons arg args)
                                           fail3))
```

```
                                fail2))
                    fail)))
```

The actual procedure application, which is performed by `execute-application`, is accomplished in the same way as for the ordinary evaluator, except for the need to manage the continuations.

```
(define (execute-application proc args succeed fail)
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))
          succeed
          fail))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

Evaluating `amb` expressions

The `amb` special form is the key element in the nondeterministic language. Here we see the essence of the interpretation process and the reason for keeping track of the continuations. The execution procedure for `amb` defines a loop `try-next` that cycles through the execution procedures for all the possible values of the `amb` expression. Each execution procedure is called with a failure continuation that will try the next one. When there are no more alternatives to try, the entire `amb` expression fails.

```
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
                           succeed
                           (lambda ()
                             (try-next (cdr choices))))))
      (try-next cprocs))))
```

Driver loop

The driver loop for the `amb` evaluator is complex, due to the mechanism that permits the user to try again in evaluating an expression. The driver uses a procedure called `internal-loop`, which takes as argument a procedure `try-again`. The intent is that calling `try-again` should go on to the next untried alternative in the nondeterministic evaluation. `Internal-loop` either calls `try-again` in response to the user typing `try-again` at the driver loop, or else starts a new evaluation by calling `ambeval`.

The failure continuation for this call to `ambeval` informs the user that there are no more values and re-invokes the driver loop.

The success continuation for the call to `ambeval` is more subtle. We print the obtained value and then invoke the internal loop again with a `try-again` procedure that will be able to try the next alternative. This `next-alternative` procedure is the second argument that was passed to the success continuation. Ordinarily, we think of this second argument as a failure continuation to be used if the current evaluation branch later fails. In this case, however, we have completed a successful evaluation, so we can invoke the ``failure'' alternative branch in order to search for additional successful evaluations.

```
(define input-prompt ";;; Amb-Eval input:")
(define output-prompt ";;; Amb-Eval value:")
(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline)
            (display ";;; Starting a new problem ")
            (ambeval input
                     the-global-environment
                     ;; ambeval success
                     (lambda (val next-alternative)
                       (announce-output output-prompt)
                       (user-print val)
                       (internal-loop next-alternative))
                     ;; ambeval failure
                     (lambda ()
                       (announce-output
                        ";;; There are no more values of")
                       (user-print input)
                       (driver-loop)))))))
  (internal-loop
   (lambda ()
     (newline)
     (display ";;; There is no current problem")
     (driver-loop))))
```

The initial call to `internal-loop` uses a `try-again` procedure that complains that there is no current problem and restarts the driver loop. This is the behavior that will happen if the user types `try-again` when there is no evaluation in progress.

Exercise 4.50. Implement a new special form `ramb` that is like `amb` except that it searches alternatives in a random order, rather than from left to right. Show how this can help with Alyssa's problem in exercise 4.49.

Exercise 4.51. Implement a new kind of assignment called `permanent-set!` that is not undone upon failure. For example, we can choose two distinct elements from a list and count the number of trials required to make a successful choice as follows:

```
(define count 0)
(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b c))))
  (permanent-set! count (+ count 1))
  (require (not (eq? x y)))
  (list x y count))
;;; Starting a new problem
;;; Amb-Eval value:
(a b 2)
;;; Amb-Eval input:
```

```
try-again
;;; Amb-Eval value:
(a c 3)
```

What values would have been displayed if we had used `set!` here rather than `permanent-set!` ?

Exercise 4.52. Implement a new construct called `if-fail` that permits the user to catch the failure of an expression. `If-fail` takes two expressions. It evaluates the first expression as usual and returns as usual if the evaluation succeeds. If the evaluation fails, however, the value of the second expression is returned, as in the following example:

```
;;; Amb-Eval input:
(if-fail (let ((x (an-element-of '(1 3 5))))
           (require (even? x))
           x)
         'all-odd)
;;; Starting a new problem
;;; Amb-Eval value:
all-odd
;;; Amb-Eval input:
(if-fail (let ((x (an-element-of '(1 3 5 8))))
           (require (even? x))
           x)
         'all-odd)
;;; Starting a new problem
;;; Amb-Eval value:
8
```

Exercise 4.53. With `permanent-set!` as described in exercise 4.51 and `if-fail` as in exercise 4.52, what will be the result of evaluating

```
(let ((pairs '()))
  (if-fail (let ((p (prime-sum-pair '(1 3 5 8) '(20 35 110))))
             (permanent-set! pairs (cons p pairs))
             (amb))
           pairs))
```

Exercise 4.54. If we had not realized that `require` could be implemented as an ordinary procedure that uses `amb`, to be defined by the user as part of a nondeterministic program, we would have had to implement it as a special form. This would require syntax procedures

```
(define (require? exp) (tagged-list? exp 'require))

(define (require-predicate exp) (cadr exp))
```

and a new clause in the dispatch in `analyze`

```
((require? exp) (analyze-require exp))
```

as well the procedure `analyze-require` that handles `require` expressions. Complete the following definition of `analyze-require`.

```
(define (analyze-require exp)
  (let ((pproc (analyze (require-predicate exp))))
    (lambda (env succeed fail)
      (pproc env
             (lambda (pred-value fail2)
               (if <??>
                   <??>
```

```
        (succeed 'ok fail2)))
  fail))))
```

---

[42] We assume that we have previously defined a procedure `prime?` that tests whether numbers are prime. Even with `prime?` defined, the `prime-sum-pair` procedure may look suspiciously like the unhelpful ``pseudo-Lisp'' attempt to define the square-root function, which we described at the beginning of section 1.1.7. In fact, a square-root procedure along those lines can actually be formulated as a nondeterministic program. By incorporating a search mechanism into the evaluator, we are eroding the distinction between purely declarative descriptions and imperative specifications of how to compute answers. We'll go even farther in this direction in section 4.4.

[43] The idea of `amb` for nondeterministic programming was first described in 1961 by John McCarthy (see McCarthy 1967).

[44] In actuality, the distinction between nondeterministically returning a single choice and returning all choices depends somewhat on our point of view. From the perspective of the code that uses the value, the nondeterministic choice returns a single value. From the perspective of the programmer designing the code, the nondeterministic choice potentially returns all possible values, and the computation branches so that each value is investigated separately.

[45] One might object that this is a hopelessly inefficient mechanism. It might require millions of processors to solve some easily stated problem this way, and most of the time most of those processors would be idle. This objection should be taken in the context of history. Memory used to be considered just such an expensive commodity. In 1964 a megabyte of RAM cost about $400,000. Now every personal computer has many megabytes of RAM, and most of the time most of that RAM is unused. It is hard to underestimate the cost of mass-produced electronics.

[46] Automagically: ``Automatically, but in a way which, for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), the speaker doesn't feel like explaining.'' (Steele 1983, Raymond 1993)

[47] The integration of automatic search strategies into programming languages has had a long and checkered history. The first suggestions that nondeterministic algorithms might be elegantly encoded in a programming language with search and automatic backtracking came from Robert Floyd (1967). Carl Hewitt (1969) invented a programming language called Planner that explicitly supported automatic chronological backtracking, providing for a built-in depth-first search strategy. Sussman, Winograd, and Charniak (1971) implemented a subset of this language, called MicroPlanner, which was used to support work in problem solving and robot planning. Similar ideas, arising from logic and theorem proving, led to the genesis in Edinburgh and Marseille of the elegant language Prolog (which we will discuss in section 4.4). After sufficient frustration with automatic search, McDermott and Sussman (1972) developed a language called Conniver, which included mechanisms for placing the search strategy under programmer control. This proved unwieldy, however, and Sussman and Stallman (1975) found a more tractable approach while investigating methods of symbolic analysis for electrical circuits. They developed a non-chronological backtracking scheme that was based on tracing out the logical dependencies connecting facts, a technique that has come to be known as dependency-directed backtracking. Although their method was complex, it produced reasonably efficient programs because it did little redundant search. Doyle (1979) and McAllester (1978, 1980) generalized and clarified the methods of Stallman and Sussman, developing a new paradigm for formulating search that is now called truth maintenance. Modern problem-solving systems all use some form of truth-maintenance system as a substrate. See Forbus and deKleer 1993 for a discussion of elegant ways to build truth-maintenance systems and applications using truth maintenance. Zabih, McAllester, and Chapman 1987 describes a nondeterministic extension to Scheme that is based on `amb`; it is similar to the interpreter described in this section, but more sophisticated, because it uses dependency-directed backtracking rather than chronological backtracking. Winston 1992 gives an introduction to both kinds of backtracking.

[48] Our program uses the following procedure to determine if the elements of a list are distinct:

```
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
```

`Member` is like `memq` except that it uses `equal?` instead of `eq?` to test for equality.

[49] This is taken from a booklet called ``Problematical Recreations,'' published in the 1960s by Litton Industries, where it is attributed to the Kansas State Engineer.

[50] Here we use the convention that the first element of each list designates the part of speech for the rest of the words in the list.

[51] Notice that `parse-word` uses `set!` to modify the unparsed input list. For this to work, our `amb` evaluator must undo the effects of `set!` operations when it backtracks.

[52] Observe that this definition is recursive -- a verb may be followed by any number of prepositional phrases.

[53] This kind of grammar can become arbitrarily complex, but it is only a toy as far as real language understanding is concerned. Real natural-language understanding by computer requires an elaborate mixture of syntactic analysis and interpretation of meaning. On the other hand, even toy parsers can be useful in supporting flexible command languages for programs such as information-retrieval systems. Winston 1992 discusses computational approaches to real language understanding and also the applications of simple grammars to command languages.

[54] Although Alyssa's idea works just fine (and is surprisingly simple), the sentences that it generates are a bit boring -- they don't sample the possible sentences of this language in a very interesting way. In fact, the grammar is highly recursive in many places, and Alyssa's technique ``falls into'' one of these recursions and gets stuck. See exercise 4.50 for a way to deal with this.

[55] We chose to implement the lazy evaluator in section 4.2 as a modification of the ordinary metacircular evaluator of section 4.1.1. In contrast, we will base the `amb` evaluator on the analyzing evaluator of section 4.1.7, because the execution procedures in that evaluator provide a convenient framework for implementing backtracking.

[56] We assume that the evaluator supports `let` (see exercise 4.22), which we have used in our nondeterministic programs.

[57] We didn't worry about undoing definitions, since we can assume that internal definitions are scanned out (section 4.1.6).

[Go to first, previous, next page; contents; index]