# 22

# Nondeterminism

Programming languages save us from being swamped by a mass of detail. Lisp is a good language because it handles so many details itself, enabling programmers to make the most of their limited tolerance for complexity. This chapter describes how macros can make Lisp handle another important class of details: the details of transforming a nondeterministic algorithm into a deterministic one.

This chapter is divided into five parts. The first explains what nondeterminism is. The second describes a Scheme implementation of nondeterministic *choose* and *fail* which uses continuations. The third part presents Common Lisp versions of *choose* and *fail* which build upon the continuation-passing macros of Chapter 20. The fourth part shows how the cut operator can be understood independently of Prolog. The final part suggests refinements of the original nondeterministic operators.

The nondeterministic choice operators defined in this chapter will be used to write an ATN compiler in Chapter 23 and an embedded Prolog in Chapter 24.

## 22.1 The Concept

A nondeterministic algorithm is one which relies on a certain sort of supernatural foresight. Why talk about such algorithms when we don't have access to computers with supernatural powers? Because a nondeterministic algorithm can be *simulated* by a deterministic one. For purely functional programs—that is, those with no side-effects—simulating nondeterminism is particularly straightforward. In purely functional programs, nondeterminism can be implemented by search with backtracking.

This chapter shows how to simulate nondeterminism in functional programs. If we have a simulator for nondeterminism, we can expect it to produce results whenever a truly nondeterministic machine would. In many cases, writing a program which depends on supernatural insight to solve a problem is easier than writing one which doesn't, so such a simulator would be a good thing to have.

In this section we will define the class of powers that nondeterminism allows us; the next section demonstrates their utility in some sample programs. The examples in these first two sections are written in Scheme. (Some differences between Scheme and Common Lisp are summarized on page 259.)

A nondeterministic algorithm differs from a deterministic one because it can use the two special operators *choose* and *fail*. *Choose* is a function which takes a finite set and returns one element. To explain how *choose* chooses, we must first introduce the concept of a computational *future.*

Here we will represent *choose* as a function `choose` which takes a list and returns one element. For each element, there is a set of futures the computation could have if that element were chosen. In the following expression

```
(let ((x (choose '(1 2 3))))
  (if (odd? x)
      (+ x 1)
      x))
```

there are three possible futures for the computation when it reaches the point of the `choose`:

1. If `choose` returns `1`, the computation will go through the then-clause of the `if`, and will return `2`.
2. If `choose` returns `2`, the computation will go through the else-clause of the `if`, and will return `2`.
3. If `choose` returns `3`, the computation will go through the then-clause of the `if`, and will return `4`.

In this case, we know exactly what the future of the computation will be as soon as we see what `choose` returns. In the general case, each choice is associated with a *set* of possible futures, because within some futures there could be additional `choose`s. For example, with

```
(let ((x (choose '(2 3))))
  (if (odd? x)
      (choose '(a b))
      x))
```

there are two sets of futures at the time of the first `choose`:

1. If `choose` returns 2, the computation will go through the else-clause of the `if`, and will return 2.
2. If `choose` returns 3, the computation will go through the then-clause of the `if`. At this point, the path of the computation splits into two possible futures, one in which `a` is returned, and one in which `b` is.

The first set has one future and the second set has two, so the computation has three possible futures.

The point to remember is, if *choose* is given a choice of several alternatives, each one is associated with a set of possible futures. Which choice will it return? We can assume that *choose* works as follows:

1. It will only return a choice for which some future does not contain a call to *fail*.
2. A *choose* over zero alternatives is equivalent to a *fail*.

So, for example, in

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (fail)
      x))
```

each of the possible choices has exactly one future. Since the future for a choice of 1 contains a call to `fail`, only 2 can be chosen. So the expression as a whole is deterministic: it always returns 2.

However, the following expression is not deterministic:

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (let ((y (choose '(a b))))
        (if (eq? y 'a)
            (fail)
            y))
      x))
```

At the first `choose`, there are two possible futures for a choice of 1, and one for a choice of 2. Within the former, though, the future is really deterministic, because a choice of `a` would result in a call to `fail`. So the expression as a whole could return either `b` or 2.

Finally, there is only one possible value for the expression

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (choose '())
      x))
```

because if `1` is chosen, the future goes through a `choose` with no choices. This example is thus equivalent to the last but one.

It may not be clear yet from the preceding examples, but we have just got ourselves an abstraction of astounding power. In nondeterministic algorithms we are allowed to say "choose an element such that nothing we do later will result in a call to *fail*." For example, this is a perfectly legitimate nondeterministic algorithm for discovering whether you have a known ancestor called Igor:

```
Function Ig(n)
  if name(n) = 'Igor'
    then return n
  else if parents(n)
    then return Ig(choose(parents(n)))
  else fail
```

The *fail* operator is used to influence the value returned by *choose*. If we ever encounter a *fail*, *choose* would have chosen incorrectly. By definition *choose* guesses correctly. So if we want to guarantee that the computation will never pursue a certain path, all we need do is put a *fail* somewhere in it, and that path will never be followed. Thus, as it works recursively through generations of ancestors, the function Ig is able to choose at each step a path which leads to an Igor—to guess whether to follow the mother's or father's line.

It is as if a program can specify that *choose* pick some element from a set of alternatives, use the value returned by *choose* for as long as it wants, and then retroactively decide, by using *fail* as a veto, what it wants *choose* to have picked. And, presto, it turns out that that's just what *choose* did return. Hence the model in which *choose* has foresight.

In reality *choose* cannot have supernatural powers. Any implementation of *choose* must simulate correct guessing by backtracking when it discovers mistakes, like a rat finding its way through a maze. But all this backtracking can be done beneath the surface. Once you have some form of *choose* and *fail*, you get to write algorithms like the one above, as if it really were possible to guess what ancestor to follow. By using *choose* it is possible to write an algorithm to search some problem space just by writing an algorithm to traverse it.

```
(define (descent n1 n2)
  (if (eq? n1 n2)
      (list n2)
      (let ((p (try-paths (kids n1) n2)))
        (if p (cons n1 p) #f))))

(define (try-paths ns n2)
  (if (null? ns)
      #f
      (or (descent (car ns) n2)
          (try-paths (cdr ns) n2))))
```

Figure 22.1: Deterministic tree search.

```
(define (descent n1 n2)
  (cond ((eq? n1 n2) (list n2))
        ((null? (kids n1)) (fail))
        (else (cons n1 (descent (choose (kids n1)) n2)))))
```

Figure 22.2: Nondeterministic tree search.

## 22.2 Search

Many classic problems can be formulated as search problems, and for such problems nondeterminism often turns out to be a useful abstraction. Suppose `nodes` is bound to a list of nodes in a tree, and `(kids` *n*`)` is a function which returns the descendants of node *n*, or `#f` if there are none. We want to write a function `(descent` $n_1$ $n_2$`)` which returns a list of nodes on some path from $n_1$ to its descendant $n_2$, if there is one. Figure 22.1 shows a deterministic version of this function.

Nondeterminism allows the programmer to ignore the details of finding a path. It's possible simply to tell `choose` to find a node *n* such that there is a path from *n* to our destination. Using nondeterminism we can write the simpler version of `descent` shown in Figure 22.2.

The version shown in Figure 22.2 does not explicitly search for a node on the right path. It is written on the assumption that `choose` has chosen an `n` with the desired properties. If we are used to looking at deterministic programs, we may not perceive that `choose` has to work as if it could *guess* what `n` would make it

```
(define (two-numbers)
  (list (choose '(0 1 2 3 4 5))
        (choose '(0 1 2 3 4 5))))

(define (parlor-trick sum)
  (let ((nums (two-numbers)))
    (if (= (apply + nums) sum)
        `(the sum of ,@nums)
        (fail))))
```

Figure 22.3: Choice in a subroutine.

through the computation which follows without failing.

Perhaps a more convincing example of the power of *choose* is its ability to guess what will happen even in calling functions. Figure 22.3 contains a pair of functions to guess two numbers which sum to a number given by the caller. The first function, `two-numbers`, nondeterministically chooses two numbers and returns them in a list. When we call `parlor-trick`, it calls `two-numbers` for a list of two integers. Note that, in making its choice, `two-numbers` doesn't have access to the number entered by the user.

If the two numbers guessed by `choose` don't sum to the number entered by the user, the computation fails. We can rely on `choose` having avoided computational paths which fail, if there are any which don't. Thus we can assume that if the caller gives a number in the right range, `choose` will have guessed right, as indeed it does:[1]

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

In simple searches, the built-in Common Lisp function `find-if` would do just as well. Where is the advantage of nondeterministic choice? Why not just iterate through the list of alternatives in search of the element with the desired properties? The crucial difference between *choose* and conventional iteration is that its extent with respect to *fails* is unbounded. Nondeterministic *choose* can see arbitrarily far into the future; if something is going to happen at any point in the future which would have invalidated some guess *choose* might make, we can assume that *choose* knows to avoid guessing it. As we saw in `parlor-trick`,

[1] Since the order of argument evaluation is unspecified in Scheme (as opposed to Common Lisp, which specifies left-to-right), this call might also return `(THE SUM OF 5 2)`.

the *fail* operator works even after we return from the function in which the *choose* occurs.

This kind of failure happens in the search done by Prolog, for example. Nondeterminism is useful in Prolog because one of the central features of this language is its ability to return answers to a query one at a time. By following this course instead of returning all the valid answers at once, Prolog can handle recursive rules which would otherwise yield infinitely large sets of answers.

The initial reaction to `descent` may be like the initial reaction to a merge sort: where does the work get done? As in a merge sort, the work gets done implicitly, but it does get done. Section 22.3 describes an implementation of *choose* in which all the code examples presented so far are real running programs.

These examples show the value of nondeterminism as an abstraction. The best programming language abstractions save not just typing, but thought. In automata theory, some proofs are difficult even to conceive of without relying on nondeterminism. A language which allows nondeterminism may give programmers a similar advantage.

## 22.3 Scheme Implementation

This section explains how to use continuations to simulate nondeterminism. Figure 22.4 contains Scheme implementations of *choose* and *fail*. Beneath the surface, `choose` and `fail` simulate nondeterminism by backtracking. A backtracking search program must somehow store enough information to pursue other alternatives if the chosen one fails. This information is stored in the form of continuations on the global list `*paths*`.

The function `choose` is passed a list of alternatives in `choices`. If `choices` is empty, then `choose` calls `fail`, which sends the computation back to the previous `choose`. If `choices` is (*first* . *rest*), `choose` first pushes onto `*paths*` a continuation in which `choose` is called on *rest*, then returns *first*.

The function `fail` is simpler: it just pops a continuation off `*paths*` and calls it. If there aren't any saved paths left, then `fail` returns the symbol `@`. However, it won't do simply to return it as a function ordinarily returns values, or it will be returned as the value of the most recent `choose`. What we really want to do is return `@` right to the toplevel. We do this by binding `cc` to the continuation where `fail` is defined, which presumably is the toplevel. By calling `cc`, `fail` can return straight there.

The implementation in Figure 22.4 treats `*paths*` as a stack, always failing back to the most recent choice point. This strategy, known as *chronological backtracking*, results in depth-first search of the problem space. The word "nondeterminism" is often used as if it were synonymous with the depth-first imple-

```
(define *paths* ())
(define failsym '@)

(define (choose choices)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (set! *paths*
                (cons (lambda ()
                        (cc (choose (cdr choices))))
                      *paths*))
          (car choices)))))

(define fail)

(call-with-current-continuation
  (lambda (cc)
    (set! fail
          (lambda ()
            (if (null? *paths*)
                (cc failsym)
                (let ((p1 (car *paths*)))
                  (set! *paths* (cdr *paths*))
                  (p1)))))))
```

Figure 22.4: Scheme implementation of *choose* and *fail*.

mentation. Floyd's classic paper on nondeterministic algorithms uses the term in this sense, and this is also the kind of nondeterminism we find in nondeterministic parsers and in Prolog. However, it should be noted that the implementation given in Figure 22.4 is not the only possible implementation, nor even a correct one. In principle, *choose* ought to be able to choose an object which meets any computable specification. But a program which used these versions of `choose` and `fail` to search a graph might not terminate, if the graph contained cycles. ○

In practice, nondeterminism usually means using a depth-first implementation equivalent to the one in Figure 22.4, and leaving it to the user to avoid loops in the search space. However, for readers who are interested, the last section in this chapter describes how to implement true *choose* and *fail*.

## 22.4 Common Lisp Implementation

This section describes how to write a form of *choose* and *fail* in Common Lisp. As the previous section showed, `call/cc` makes it easy to simulate nondeterminism in Scheme. Continuations provide the direct embodiment of our theoretical concept of a computational future. In Common Lisp, we can use instead the continuation-passing macros of Chapter 20. With these macros we will be able to provide a form of *choose* slightly uglier than the Scheme version presented in the previous section, but equivalent in practice.

Figure 22.5 contains a Common Lisp implementation of *fail*, and two versions of *choose*. The syntax of a Common Lisp `choose` is slightly different from the Scheme version. The Scheme `choose` took one argument: a list of choices from which to select a value. The Common Lisp version has the syntax of a `progn`. It can be followed by any number of expressions, from which it chooses one to evaluate:

```
> (defun do2 (x)
    (choose (+ x 2) (* x 2) (expt x 2)))
DO2
> (do2 3)
5
> (fail)
6
```

At the toplevel, we see more clearly the backtracking which underlies nondeterministic search. The variable `*paths*` is used to store paths which have not yet been followed. When the computation reaches a `choose` expression with several alternatives, the first alternative is evaluated, and the remaining choices are stored on `*paths*`. If the program later on encounters a `fail`, the last stored choice will be popped off `*paths*` and restarted. When there are no more paths left to restart, `fail` returns a special value:

```
> (fail)
9
> (fail)
@
```

In Figure 22.5 the constant `failsym`, which represents failure, is defined to be the symbol `@`. If you wanted to be able to have `@` as an ordinary return value, you could make `failsym` a gensym instead.

The other nondeterministic choice operator, `choose-bind`, has a slightly different form. It should be given a symbol, a list of choices, and a body of code. It will do a *choose* on the list of choices, bind the symbol to the value chosen, and evaluate the body of code:

```
(defparameter *paths* nil)
(defconstant failsym '@)

(defmacro choose (&rest choices)
  (if choices
      `(progn
         ,@(mapcar #'(lambda (c)
                       `(push #'(lambda () ,c) *paths*))
                   (reverse (cdr choices)))
         ,(car choices))
      '(fail)))

(defmacro choose-bind (var choices &body body)
  `(cb #'(lambda (,var) ,@body) ,choices))

(defun cb (fn choices)
  (if choices
     (progn
       (if (cdr choices)
           (push #'(lambda () (cb fn (cdr choices)))
                 *paths*))
       (funcall fn (car choices)))
     (fail)))

(defun fail ()
  (if *paths*
      (funcall (pop *paths*))
      failsym))
```

Figure 22.5: Nondeterministic operators in Common Lisp.

```
> (choose-bind x '(marrakesh strasbourg vegas)
    (format nil "Let's go to ~A." x))
"Let's go to MARRAKESH."
> (fail)
"Let's go to STRASBOURG."
```

It is only for convenience that the Common Lisp implementation provides two choice operators. You could get the effect of `choose` from `choose-bind` by always translating

```
(choose (foo) (bar))
```

into

```
(choose-bind x '(1 2)
  (case x
    (1 (foo))
    (2 (bar))))
```

but programs are easier to read if we have a separate operator for this case.[2]

The Common Lisp choice operators store the bindings of relevant variables using closures and variable capture. As macros, `choose` and `choose-bind` get expanded within the lexical environment of the containing expressions. Notice that what they push onto `*paths*` is a closure over the choice to be saved, locking in all the bindings of the lexical variables referred to within it. For example, in the expression

```
(let ((x 2))
  (choose
    (+ x 1)
    (+ x 100)))
```

the value of `x` will be needed when the saved choices are restarted. This is why `choose` is written to wrap its arguments in lambda-expressions. The expression above gets macroexpanded into:

```
(let ((x 2))
  (progn
    (push #'(lambda () (+ x 100))
          *paths*)
    (+ x 1)))
```

The object which gets stored on `*paths*` is a closure containing a pointer to `x`. It is the need to preserve variables in closures which dictates the difference between the syntax of the Scheme and Common Lisp choice operators.

If we use `choose` and `fail` together with the continuation-passing macros of Chapter 20, a pointer to our continuation variable `*cont*` will get saved as well. By defining functions with `=defun`, calling them with `=bind`, and having them return values with `=values`, we will be able to use nondeterminism in any Common Lisp program.

With these macros, we can successfully run the example in which the nondeterministic choice occurs in a subroutine. Figure 22.6 shows the Common Lisp version of `parlor-trick`, which works as it did in Scheme:

[2] If desired, the exported interface to this code could consist of just a single operator, because `(fail)` is equivalent to `(choose)`.

```
(=defun two-numbers ()
  (choose-bind n1 '(0 1 2 3 4 5)
    (choose-bind n2 '(0 1 2 3 4 5)
      (=values n1 n2))))

(=defun parlor-trick (sum)
  (=bind (n1 n2) (two-numbers)
    (if (= (+ n1 n2) sum)
        `(the sum of ,n1 ,n2)
        (fail))))
```

Figure 22.6: Common Lisp choice in a subroutine.

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

This works because the expression

```
(=values n1 n2)
```

gets macroexpanded into

```
(funcall *cont* n1 n2)
```

within the `choose-binds`. Each `choose-bind` is in turn macroexpanded into a closure, which keeps pointers to all the variables referred to in the body, including `*cont*`.

The restrictions on the use of `choose`, `choose-bind`, and `fail` are the same as the restrictions given in Figure 20.5 for code which uses the continuation-passing macros. Where a choice expression occurs, it must be the last thing to be evaluated. Thus if we want to make sequential choices, in Common Lisp the choices have to be nested:

```
> (choose-bind first-name '(henry william)
    (choose-bind last-name '(james higgins)
      (=values (list first-name last-name))))
(HENRY JAMES)
> (fail)
(HENRY HIGGINS)
> (fail)
(WILLIAM JAMES)
```

which will, as usual, result in depth-first search.

The operators defined in Chapter 20 claimed the right to be the last expressions evaluated. This right is now preempted by the new layer of macros; an `=values` expression should appear within a `choose` expression, and not vice versa. That is,

```
(choose (=values 1) (=values 2))
```

will work, but

```
(=values (choose 1 2))                                    ; wrong
```

will not. (In the latter case, the expansion of the `choose` would be unable to capture the instance of `*cont*` in the expansion of the `=values`.)

As long as we respect the restrictions outlined here and in Figure 20.5, nondeterministic choice in Common Lisp will now work as it does in Scheme. Figure 22.7 shows a Common Lisp version of the nondeterministic tree search program given in Figure 22.2. The Common Lisp `descent` is a direct translation, though it comes out slightly longer and uglier.

We now have Common Lisp utilities which make it possible to do nondeterministic search without explicit backtracking. Having taken trouble to write this code, we can reap the benefits by writing in very few lines programs which would otherwise be large and messy. By building another layer of macros on top of those presented here, we will be able to write an ATN compiler in one page of code (Chapter 23), and a sketch of Prolog in two (Chapter 24).

Common Lisp programs which use `choose` should be compiled with tail-recursion optimization—not just to make them faster, but to avoid running out of stack space. Programs which "return" values by calling continuation functions never actually return until the final `fail`. Without the optimization of tail-calls, ○ the stack would just grow and grow.

## 22.5 Cuts

This section shows how to use cuts in Scheme programs which do nondeterministic choice. Though the word *cut* comes from Prolog, the concept belongs to nondeterminism generally. You might want to use cuts in any program that made nondeterministic choices.

Cuts are easier to understand when considered independently of Prolog. Let's imagine a real-life example. Suppose that the manufacturer of Chocoblob candies decides to run a promotion. A small number of boxes of Chocoblobs will also contain tokens entitling the recipient to valuable prizes. To ensure fairness, no two of the winning boxes are sent to the same city.

```
> (=defun descent (n1 n2)
    (cond ((eq n1 n2) (=values (list n2)))
          ((kids n1) (choose-bind n (kids n1)
                       (=bind (p) (descent n n2)
                          (=values (cons n1 p)))))
          (t (fail))))
DESCENT
> (defun kids (n)
    (case n
      (a '(b c))
      (b '(d e))
      (c '(d f))
      (f '(g))))
KIDS
> (descent 'a 'g)
(A C F G)
> (fail)
@
> (descent 'a 'd)
(A B D)
> (fail)
(A C D)
> (fail)
@
> (descent 'a 'h)
@
```

Figure 22.7: Nondeterministic search in Common Lisp

After the promotion has begun, it emerges that the tokens are small enough to be swallowed by children. Hounded by visions of lawsuits, Chocoblob lawyers begin a frantic search for all the special boxes. Within each city, there are multiple stores that sell Chocoblobs; within each store, there are multiple boxes. But the lawyers may not have to open every box: once they find a coin-containing box in a given city, they do not have to search any of the other boxes in that city, because each city has at most one special box. To realize this is to do a cut.

What's *cut* is a portion of the search tree. For Chocoblobs, the search tree exists physically: the root node is at the company's head office; the children of this node are the cities where the special boxes were sent; the children of those nodes are the stores in each city; and the children of each store represent the boxes in

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (newline)
    (let* ((store (choose '(1 2)))
           (box (choose '(1 2))))
      (let ((triple (list city store box)))
        (display triple)
        (if (coin? triple)
            (display 'c))
        (fail)))))

(define (coin? x)
  (member x '((la 1 2) (ny 1 1) (bos 2 2))))
```

Figure 22.8: Exhaustive Chocoblob search.

that store. When the lawyers searching this tree find one of the boxes containing a coin, they can prune off all the unexplored branches descending from the city they're in now.

Cuts actually take two operations: you can do a cut when you know that part of the search tree is useless, but first you have to *mark* the tree at the point where it can be cut. In the Chocoblob example, common sense tells us that the tree is marked as we enter each city. It's hard to describe in abstract terms what a Prolog cut does, because the marks are implicit. With an explicit mark operator, the effect of a cut will be more easily understood.

Figure 22.8 shows a program that nondeterministically searches a smaller version of the Chocoblob tree. As each box is opened, the program displays a list of (*city store box*). If the box contains a coin, a `c` is printed after it:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C(LA 2 1)(LA 2 2)
(NY 1 1)C(NY 1 2)(NY 2 1)(NY 2 2)
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
@
```

To implement the optimized search technique discovered by the Chocoblob lawyers, we need two new operators: `mark` and `cut`. Figure 22.9 shows one way to define them. Whereas nondeterminism itself can be understood independently of any particular implementation, pruning the search tree is an optimization technique, and depends very much on how `choose` is implemented. The `mark` and

```
(define (mark) (set! *paths* (cons fail *paths*)))

(define (cut)
  (cond ((null? *paths*))
        ((equal? (car *paths*) fail)
         (set! *paths* (cdr *paths*)))
        (else
         (set! *paths* (cdr *paths*))
         (cut))))
```

Figure 22.9: Marking and pruning search trees.

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (mark)                                                    ;
    (newline)
    (let* ((store (choose '(1 2)))
           (box (choose '(1 2))))
      (let ((triple (list city store box)))
        (display triple)
        (if (coin? triple)
            (begin (cut) (display 'c)))                       ;
        (fail)))))
```

Figure 22.10: Pruned Chocoblob search.

`cut` defined in Figure 22.9 are suitable for use with the depth-first implementation of `choose` (Figure 22.4).

The general idea is for `mark` to store markers in `*paths*`, the list of unexplored choice-points. Calling `cut` pops `*paths*` all the way down to the most recent marker. What should we use as a marker? We could use e.g. the symbol `m`, but that would require us to rewrite `fail` to ignore the `m`s when it encountered them. Fortunately, since functions are data objects too, there is at least one marker that will allow us to use `fail` as is: the function `fail` itself. Then if `fail` happens on a marker, it will just call itself.

Figure 22.10 shows how these operators would be used to prune the search tree in the Chocoblob case. (Changed lines are indicated by semicolons.) We call `mark` upon choosing a city. At this point, `*paths*` contains one continuation,

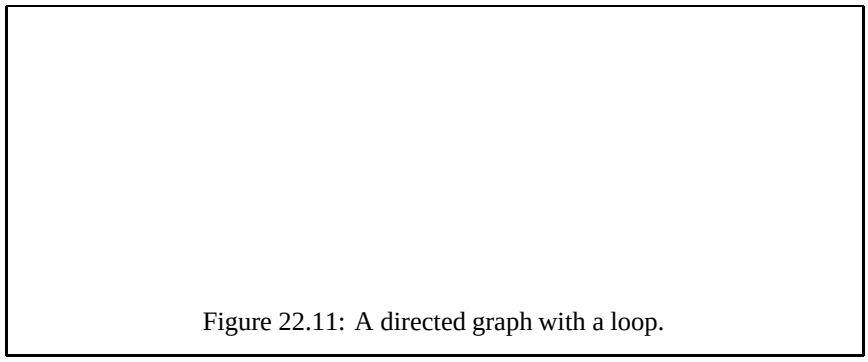

Figure 22.11: A directed graph with a loop.

representing the search of the remaining cities.

If we find a box with a coin in it, we call `cut`, which sets `*paths*` back to the value it had at the time of the `mark`. The effects of the cut are not visible until the next call to `fail`. But when it comes, after the `display`, the next `fail` sends the search all the way up to the topmost `choose`, even if there would otherwise have been live choice-points lower in the search tree. The upshot is, as soon as we find a box with a coin in it, we resume the search at the next city:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C
(NY 1 1)C
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
@
```

In this case, we open seven boxes instead of twelve.

## 22.6 True Nondeterminism

A deterministic graph-searching program would have to take explicit steps to avoid getting caught in a circular path. Figure 22.11 shows a directed graph containing a loop. A program searching for a path from node `a` to node `e` risks getting caught in the circular path ⟨`a`, `b`, `c`⟩. Unless a deterministic searcher used randomization, breadth-first search, or checked explicitly for circular paths, the search might never terminate. The implementation of `path` shown in Figure 22.12 avoids circular paths by searching breadth-first.

In principle, nondeterminism should save us the trouble of even considering circular paths. The depth-first implementation of *choose* and *fail* given in Section 22.3 is vulnerable to the problem of circular paths, but if we were being picky, we would expect nondeterministic *choose* to be able to select an object

```
(define (path node1 node2)
  (bf-path node2 (list (list node1))))

(define (bf-path dest queue)
  (if (null? queue)
      '@
      (let* ((path (car queue))
             (node (car path)))
        (if (eq? node dest)
            (cdr (reverse path))
            (bf-path dest
                     (append (cdr queue)
                             (map (lambda (n)
                                    (cons n path))
                                  (neighbors node))))))))
```

Figure 22.12: Deterministic search.

```
(define (path node1 node2)
  (cond ((null? (neighbors node1)) (fail))
        ((memq node2 (neighbors node1)) (list node2))
        (else (let ((n (true-choose (neighbors node1))))
                (cons n (path n node2))))))
```

Figure 22.13: Nondeterministic search.

which meets any computable specification, and this case is no exception. Using a correct *choose*, we should be able to write the shorter and clearer version of `path` shown in Figure 22.13.

This section shows how to implement versions *choose* and *fail* which are safe even from circular paths. Figure 22.14 contains a Scheme implementation of true nondeterministic *choose* and *fail*. Programs which use these versions of *choose* ○ and *fail* should find solutions whenever the equivalent nondeterministic algorithms would, subject to hardware limitations.

The implementation of `true-choose` defined in Figure 22.14 works by treating the list of stored paths as a queue. Programs using `true-choose` will search their state-space breadth-first. When the program reaches a choice-point, continuations to follow each choice are appended to the end of the list of stored paths.

```
(define *paths* ())
(define failsym '@)

(define (true-choose choices)
  (call-with-current-continuation
    (lambda (cc)
      (set! *paths* (append *paths*
                            (map (lambda (choice)
                                   (lambda () (cc choice)))
                                 choices)))
      (fail))))

(define fail)

(call-with-current-continuation
  (lambda (cc)
    (set! fail
          (lambda ()
            (if (null? *paths*)
                (cc failsym)
                (let ((p1 (car *paths*)))
                  (set! *paths* (cdr *paths*))
                  (p1)))))))
```

Figure 22.14: Correct *choose* in Scheme.

(Scheme's `map` returns the same values as Common Lisp's `mapcar`.) After this there is a call to `fail`, which is unchanged.

This version of *choose* would allow the implementation of `path` defined in Figure 22.13 to find a path—indeed, the shortest path—from `a` to `e` in the graph displayed in Figure 22.11.

Although for the sake of completeness this chapter has provided correct versions of *choose* and *fail*, the original implementations will usually suffice. The value of a programming language abstraction is not diminished just because its implementation isn't formally correct. In some languages we act as if we had access to all the integers, even though the largest one may be only 32767. As long as we know how far we can push the illusion, there is little danger to it—little enough, at least, to make the abstraction a bargain. The conciseness of the programs presented in the next two chapters is due largely to their use of nondeterministic *choose* and *fail*.