

6 Control Flow

6.5.4 Generators in Icon

EXAMPLE 6.85

Simple generator in Icon

Like the iterators of Clu, Python, Ruby, and C#, Icon generators can be used for enumeration-controlled iteration. Our canonical for loop example would be written as follows in Icon:

```
every i := first to last by step do {  
    ...  
}
```

Here `... to... by...` is a built-in “mixfix” generator. ■

Because Icon is intended largely for string manipulation, most of its built-in generators operate on strings. `Find(substr, str)`, for example, generates the positions (indices) within string `str` at which an occurrence of the substring `substr` can be found. `Upto(chars, str)` generates the positions within string `str` at which any character in `chars` appears. (The initial argument to `find` is a string, delimited by double quote marks; the initial argument to `upto` is a `cset` [character set], delimited by single quote marks.) The prefix operator `!` generates all elements of its operand, which can be a string, list, record, file, or table.

In comparison to conventional iterators, however, the generators of Icon are more deeply embedded in the semantics of the language. A generator can be used in any context that expects an expression. The larger context is then capable of generating multiple results. The following code will print all positions in `s` that follow a blank:

EXAMPLE 6.86

A generator inside an expression

```
every i := 1 + upto(' ', s) do {  
    write(i)  
}
```

This can even be written as

```
every write(1 + upto(' ', s))
```

 ■

EXAMPLE 6.87

Generating in search of success

Generators in Icon are used not only for iteration, but also for *goal-directed search*, implemented via *backtracking*. (Backtracking is also fundamental to Prolog, which we will study in Chapter 11.) Where most languages use Boolean expressions to control selection and logically controlled loops, Icon uses a more general notion of *success* and *failure*. A conditional statement such as

```
if 2 < 3 then {
    ...
}
```

is said to execute not because the condition $2 < 3$ is true, but because the *comparison* $2 < 3$ *succeeds*. The distinction is important for generators, which are capable of producing results repeatedly until one of them causes the surrounding context to succeed (or until no more results can be produced). For example, in

```
if (i := find("abc", s)) > 6 then {
    ...
}
```

the body of the `if` statement will be executed only if the string "abc" appears beyond the sixth position in `s`. Because `find` generates its results in order, `i` will represent the first such position (if any). The execution model is as follows: `find` is capable of generating all positions at which "abc" occurs in `s`. Suppose the first such occurrence is at position 2. Then `i` is assigned the value 2, but the comparison $2 > 6$ fails. Because there is a generator inside the failed expression, Icon will resume that generator and reevaluate the expression for the next generated value. It will continue this reevaluation process until the comparison succeeds, or until the generator runs out of values, in which case it (the generator) fails, the overall expression fails definitively, and the body of the `if` is skipped. ■

EXAMPLE 6.88

Backtracking with multiple generators

If a failed expression contains more than one generator, all possible values will be explored systematically. The body of the following `if`, for example, will be executed if and only if an `x` appears at the same position in both `s` and `t`, with `i` denoting the first such matching position:

```
if (i := find("x", s)) = find("x", t) then {
    ...
}
```

If there is no matching position, then `i` will be set to the position of the final `x` in `s`, but the body of the loop will be skipped. If the programmer wishes to avoid changing `i` in the case where the overall test fails, then the *reversible assignment* operator, `<=` can be used instead of `:=`. When Icon backtracks past a reversible assignment, it restores the original value. ■

Any user-defined subroutine in Icon can be a generator if it uses the `suspend` *expr* statement instead of `return` *expr*. `Suspend` is Icon's equivalent of `yield`. If the expression following `suspend` contains an invocation of a generator, then the subroutine will suspend repeatedly, once for each generated value.

✓ CHECK YOUR UNDERSTANDING

44. Explain how Icon generators differ from the iterators of Clu, Python, Ruby, and C#, and from the iterator objects of Euclid, C++, and Java.
 45. Describe the notions of *success* and *failure* in Icon.
 46. What is *backtracking*? Why is it useful?
 47. Name a language other than Icon in which backtracking plays a fundamental role.
-

