# Programming in Icon; Part I — Programming with Generators*

*Ralph E. Griswold*

TR 85-25

October 28, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8401831.

# Programming in Icon; Part I — Programming with Generators

## 1. Introduction

The Icon programming language [1] contains a number of features that are unusual and that are not found in most other programming languages. As a result, Icon encourages programming techniques that are distinctly different from those that are possible in more conventional programming languages. Not all of these techniques are obvious. Indeed, experience with other programming languages is, if anything, an impediment to discovering new possibilities and exploiting the full range of opportunities in Icon.

This is the first in a series of technical reports on various aspects of programming in Icon. These reports are designed to explore the unusual language features in Icon in depth with particular emphasis on how they can be used in a variety of programming situations. An appendix containing problems is included to illustrate programming techniques and to provide exercises for interested readers. Most of these problems were derived from classroom exercises and tests.

A familiarity with Version 5 of Icon is assumed in these reports, although features that are the specific focus of a report are reviewed in some detail.

## 2. Generators

The term *generator* is used in an informal way to describe a variety of programming language mechanisms that allow the production of a sequence of results from a particular structure or set of values. Generators offer the advantage of the automatic maintenance of the state of computation and allow computations to be internalized that otherwise would have to be written at the source level.

Several programming languages, including IPL-V [2], CLU [3], Alphard [4] and SETL [5] contain such features. In these languages, however, generators are restricted to certain contexts or data structures. Icon generators differ from those in other programming languages primarily in their generality and their scope of applicability.

In most programming languages, the evaluation of an expression always produces exactly one result. In Icon, an expression can yield a sequence of zero or more results. This property applies to all kinds of expressions in all contexts and is a fundamental and uniform aspect of expression evaluation in Icon. The term *generator* is used informally in Icon to refer to an expression that may produce more than one result, although an expression that produces a single result is just a special case of this more general mechanism.

The next section reviews expression evaluation in Icon. It concentrates on essential matters and does not attempt to cover all the details. More material on this subject can be found in [1, 6].

### 2.1 Expression Evaluation in Icon

The number of results that an expression can produce depends on the nature of the expression and in some cases on the values of its arguments. Most expressions in Icon produce a single result as in other programming languages. For example, an arithmetic operation such as `i + j` always produces a single result. Likewise, assignment, as in `i := 1` only produces one result (the variable i).

On the other hand, a comparison operation such as `i > j` produces one result (the value of j) if the specified relationship holds, but it produces no result at all if the specified relationship does not hold. Similarly, `read(f)` produces the next line from the file f if there is one, but it does not produce a result if the end of the file has been reached.

An expression that produces at least one result is said to *succeed*, while an expression that does not produce a result is said to *fail*. An expression that is capable of failing is said to be *conditional*.

Success and failure are used in Icon to control conditional evaluation. An example is

```
if i > max then write("bound exceeded") else write("in bounds")
```

The success or failure of the comparison, not a Boolean value, is used to select the arm of the control structure to evaluate.

Conditional operations also are used to control the evaluation of loops. For example,

```
while line := read() do
   write(line)
```

copies the input file to the output file. The expression **read()** fails when the end of the input file is reached, terminating the execution of the loop. If an argument of an operation or a function fails, the operation or function is not evaluated and fails in turn. The **do** clause of the **while-do** expression is optional, and the transmission of failure to an enclosing expression can be used to formulate the expression above more concisely:

```
while write(read())
```

There are only a few generators, which can produce more than one result. An example is the string analysis function **find(s1, s2)**, which is capable of producing the positions in **s2** at which **s1** occurs as a substring. The number of results that **find(s1, s2)** is capable of producing depends on the values of **s1** and **s2**. For example, suppose the value of **s** is **"However many there are"**. Then,

```
find("we", s)
```

produces the value 3. On the other hand,

```
find("were", s)
```

does not produce any result (fails), since the first argument does not occur as a substring of the second argument.

If **s1** occurs at several positions in **s2**, **find(s1, s2)** is capable of producing all those positions. For example,

```
find("e", s)
```

is capable of producing the values 4, 6, 16, 18, and 22, corresponding to the five positions in the second argument at which **"e"** occurs as a substring. The positions are in ascending order only because that is the way **find** is defined. Note that **find** is both a conditional operation and a generator.

It is useful to characterize the capability of an expression to produce results in terms of its *result sequence*. For example,

```
find("we", s)
```

has the unit result sequence {3},

```
find("were", s)
```

has the empty result sequence {}, and

```
find("e", s)
```

has the result sequence {4, 6, 16, 18, 22}.

A result sequence whose length is zero corresponds to the notion of failure, while a result sequence whose length is greater than zero corresponds to the notion of success.

An operation, such as

```
i + j
```

that always produces exactly one result is called *monogenic*. Most operations in Icon are monogenic.

## 2.2 Contexts for Producing Results

Regardless of how many results an expression is capable of producing (the length of its result sequence), it only produces the number of results that are required by the context in which it is evaluated.

If the context only requires a single result, as in

```
i := find("e", s)
```

only a single result is produced. The value assigned to i in this expression is 4.

There are two kinds of contexts in which an expression may produce more than one result: *iteration* and *goal-directed evaluation.*

### Iteration

Iteration is denoted by the control structure

every $expr_1$ do $expr_2$

which repeatedly *resumes* $expr_1$ to produce the values in its result sequence in succession. For each value produced by $expr_1$, $expr_2$ is evaluated. Continuing the example above,

```
every i := find("e", s) do write(i)
```

writes 4, 6, 16, 18, and 22.

Note that the control clause in this control structure is of the form

i := *expr*

where *expr* is capable of producing more than one result. Thus, the evaluation of *expr* is nested within an enclosing assignment expression. Although an assignment expression itself only produces a single result, the resumption of the assignment expression in turn causes the resumption of *expr*, which produces the successive results that are assigned to i.

The do clause in every-do is optional, and the expression above can be written more concisely as

```
every write(find("e", s))
```

### Goal-Directed Evaluation

In a nested expression, a subexpression is automatically resumed if its enclosing expression does not produce a result (fails). This aspect of expression evaluation is called *goal-directed evaluation*, since it has the effect of producing the successful evaluation of top-level expressions if there is any possible combination of results from its argument expressions that makes this possible. For example, in

```
find(s1, s2) > 10
```

if the first value produced by find(s1, s2) is not greater than 10, find(s1, s2) is resumed to produce its next value. This automatic resumption stops when the comparison succeeds or when the resumption of find(s1, s2) does not produce another result, in which case the entire expression fails.

### Order of Evaluation

Expressions are evaluated from left to right, but generators are resumed in a last-in, first-out order. The generator that most recently produced a result is the first one that is resumed in either iteration or goal-directed evaluation. The result is cross-product evaluation. For example, if the value of s1 is "She evened the score" and the value of s2 is "Follow his lead", the result sequence for find("he", s1) is {2, 13}, and the result sequence for find("l", s2) is {3, 4, 12}. Consider the expression

```
every write(find("he", s1) + find("l", s2))
```

The left argument first produces 2, the second argument produces 3, and their sum, 5, is written. Next the second argument is resumed, producing 4, which is added to 2 and the sum 6 is written. The next resumption of the second argument produces 12 and 14 is written. Another resumption of the second argument produces

no result, so the first argument is resumed and produces 13. At this point the second argument is *evaluated* again, and produces its first result, 3, and so on. The sums produced are

```
2 + 3
2 + 4
2 + 12
13 + 3
13 + 4
13 + 12
```

The iteration context forces the expression to produce all its results. This corresponds to the result sequence for its argument. Thus, the result sequence for

```
find("he", s1) + find("l", s2)
```

is {5, 6, 14, 16, 17, 25}.

As for any expression, the results that are actually produced depend on the context in which the expression is evaluated. In the case of iteration, as above, all possible combinations are produced, while in goal-directed evaluation, results are generated only until the outermost expression produces a result. Consequently,

```
i := find("he", s1) + find("l", s2)
```

assigns the value 5 to i and no other results are produced.

**Conjunction**

Since an operation is only performed if all of its argument expressions produce results, this can be used to determine if a number of expressions all succeed. The *conjunction* operation, which performs no computation, is useful for this purpose. Conjunction is denoted by

$expr_1$ & $expr_2$

and produces the result of $expr_2$ (provided both $expr_1$ and $expr_2$ produce results).

### 2.3 Generative Operations

There are only a few built-in operations that are, in themselves, capable of generating more than one result. There are three functions:

```
find(s1, s2)
upto(c, s)
bal(c1, c2, c3, s)
```

The functions upto and bal differ from find only in the details of the computations that they perform [1].

The expression

```
i to j
```

generates the integers in sequence from i to j. This generator can be used in combination with iteration to compose loops, such as

```
every i := 1 to 10 do f(i)
```

which evaluates f(1), f(2), ..., f(10). This expression can be written more concisely as

```
every f(1 to 10)
```

The only other built-in generator is !x, which generates the elements of x. This operation is polymorphic and applies to any type of value that can be considered as a sequence of elements. If s is a string, the result sequence for !s is the one-character substrings of s, from left to right. Thus the result sequence for

```
!"Hello"
```

is {"H", "e", "l", "l", "o"}. Similarly, if a is a list, the result sequence for !a consists of the elements of a, so

that

```
![["Hello", 5]
```

has the result sequence {"Hello", 5}. For lists, !a produces variables corresponding to the subscripting expressions a[1], a[2], ..., so that

```
every !a := 0
```

assigns the value 0 to every element of a.

### 2.4 Control Structures

In addition to the five built-in functions and operators that are generators, there are two control structures that compose result sequences from those of other expressions: *alternation* and *repeated alternation.*

The result sequence for the alternation control structure

$expr_1$ | $expr_2$

consists of the concatenation of the result sequences for $expr_1$ and $expr_2$. For example,

```
1 | 2
```

has the result sequence {1, 2}, and

```
(1 to 4) | (6 to 10)
```

has the result sequence {1, 2, 3, 4, 6, 7, 8, 9, 10}. If an operand of alternation produces a variable, alternation produces that variable. Thus,

```
every (x | y | z) := 0
```

assigns 0 to x, y, and z. Note that

```
(x | y | z) := 0
```

only assigns 0 to x, since the context only requires the alternation expression to produce one value.

The result sequence for the repeated alternation control structure,

|*expr*

consists of the result sequence for *expr* produced repeatedly. For example, the result sequence for

```
|1
```

is {1, 1, 1, ...} and the result sequence for

```
|(1 to 3)
```

is {1, 2, 3, 1, 2, 3, ...}.

Both of these result sequences are infinite, as indicated. Infinite result sequences, in themselves, do not cause termination problems, since expressions are only resumed to produce more results if the context requires it. Thus,

```
x := |0
```

simply assigns 0 to x.

On the other hand, repeated alternation allows infinite sequences to be characterized in a concise way. For example,

```
(i := 0) & |(i +:= 1)
```

characterizes the sequence of positive integers, {1, 2, 3, ...}. The repeated alternation control structure can be applied to either of the arguments of the augmented assignment operation, instead of to the whole operation:

```
(i := 0) & (|i +:= 1)
```

in which the result sequence { i, i, i, ... } supplies identifiers for the assignment, or

```
(i := 0) & (i +:= |1)
```

in which the result sequence { 1, 1, 1, ... } supplies values for the assignment. The latter formulation is more economical, conceptually. Another formulation uses alternation to provide the initial result:

```
(i := 1) | (i +:= |1)
```

### 2.5 Termination

The expression evaluation mechanism of Icon introduces a new aspect to the question of termination. For example, consider the result sequence for

```
|(0 = 1)
```

Formally, the repeated concatenation of an empty sequence is the empty sequence, but in terms of expression evaluation, the sequence is never produced, since any context for its production leads to a computation that does not terminate. (Note that there is no general method for determining if an expression has an empty result sequence).

To avoid such problems, there is a termination condition for repeated alternation: The result sequence for |*expr* is truncated if *expr* has an empty result sequence. Thus,

```
|(0 = 1)
```

has an empty result sequence and its computation terminates.

An expression that has a non-empty result sequence at one time may have an empty result sequence at another time. Consider, for example, the result sequence for

```
|read(f)
```

The result sequence for **read(f)** itself is a single string, except when the end of the file **f** is reached, at which point the result sequence for **read(f)** is empty. Consequently, the result sequence for **read(f)** is finite and consists of the lines from the file **f**.

The termination condition for repeated alternation does not solve the general problem for termination of expression evaluation. An example of a "black hole" in expression evaluation is:

```
1 = |0
```

which never produces a result and never terminates. This is simply a consequence of an expression with an infinite result sequence in a context that never produces a result. Fortunately, such expressions have proved to be no more of a problem in practice than infinite loops in conventional expression evaluation.

### 2.6 Generative Expressions

Since alternation and repeated alternation have result sequences that have lengths that are, in general, greater than one, they can be considered to be generators. Including these, there are only seven built-in generators in Icon:

```
find(s1, s2)
upto(c, s)
bal(c1, c2, c3, s)
i to j
!x
expr1 | expr2
|expr
```

The result sequences for the first five expressions depend on the values of their arguments, while the last two generators are control structures that derive their result sequences from those of their arguments.

As noted earlier, any expression that has an argument that is a generator becomes a generator itself. For example,

```
i + (1 to 10)
```

has a result sequence of length 10, even though addition itself only produces a single result.

Conditional operations may *filter* the results of their arguments, producing only those that satisfy the condition. An example is

```
numeric(expr)
```

which produces the numeric value of *expr* if it is a number or convertible to a number. Thus `numeric(0)` produces 0 and `numeric("2")` produces 2, but `numeric("a")` fails. Thus, if

```
a := [1, 3, "0", "x", 100]
```

then

```
numeric(!a)
```

has the result sequence {1, 3, 0, 100}.

### 2.7 Other Control Structures

As stated in the preceding section, control structures can be considered to be operations on result sequences. The two alternation control structures concatenate result sequences. The selection control structure

```
if expr1 then expr2 else expr3
```

selects the result sequence for $expr_2$ or $expr_3$, depending on the length (nonzero or zero) of the result sequence for $expr_1$. For example, the result sequence for

```
if expr then 1 to 5 else 3 to 6
```

is {1, 2, 3, 4, 5} if *expr* succeeds but is {3, 4, 5, 6} if *expr* fails.

The *limitation* control structure,

```
expr \ i
```

truncates the result sequence for *expr*, limiting it to at most i results. Thus

```
|(1 to 3) \ 5
```

has the result sequence {1, 2, 3, 1, 2}. The expression that specifies the limit may be a generator. For example,

```
(1 to 10) \ (2 to 4)
```

has the result sequence {1, 2, 1, 2, 3, 1, 2, 3, 4}. Note that for each value generated by `1 to 4`, the expression `1 to 10` is evaluated anew.

Another control structure related to generators is *mutual evaluation*,

```
i(expr1, expr2, ..., expri, ..., exprn)
```

whose result sequence is the result sequence for $expr_i$, provided all of $expr_1$, $expr_2$, ..., $expr_i$, ..., $expr_n$ succeed (that is have non-empty result sequences). If any of the expressions fails, however, the mutual evaluation fails. This control structure, therefore, allows the selection of one result sequence from a list of expressions, provided all the expressions succeed. For example,

```
3(j > i, m < n, i to j)
```

has the result sequence for i to j, provided that j is greater than i and that m is less than n, but it fails otherwise.

If i is omitted, the result sequence of the last expression in the list is selected, so the previous example can be written

```
(j > i, m < n, i to j)
```

Note that

$(expr_1, expr_2, \ldots, expr_n)$

is equivalent to

$expr_1$ & $expr_2$ & ... & $expr_n$

Another use for mutual evaluation is illustrated by

```
(i := 0, i +:= |1)
```

whose result sequence is the positive integers. Compare this formulation with the three formulations for this sequence given earlier.

### 2.8 The Scope of Generation

The evaluation of an expression in Icon, and hence its capacity to generate results, is limited to its lexical site in the program. The isolation of expressions from backtracking is determined by syntactic constructions. For example, expressions that are separated by semicolons are isolated from one another. Thus, in

$expr_1$; $expr_2$

once control passes from $expr_1$, $expr_1$ cannot be resumed, regardless of what happens in $expr_2$. Semicolons are implicit between lines of program text on which one expression ends and another begins. Thus

$expr_1$
$expr_2$

is equivalent to

$expr_1$; $expr_2$

Consequently, if expressions are written on separate lines, there can be no control backtracking from a line to a previous one.

It is worth noting that

$expr_1$ & $expr_2$

evaluates $expr_1$ and $expr_2$ in a mutual goal-directed context, while

$expr_1$ ; $expr_2$

evaluates $expr_1$ and $expr_2$ independently.

The control clauses of most control structures are isolated. Thus in

if $expr_1$ then $expr_2$ else $expr_3$

if $expr_1$ succeeds but $expr_2$ fails, $expr_1$ is not resumed to produce another result.

The outermost expression in any nested expression can produce at most one result — an expression is only resumed to produce another result by an enclosing expression.

### 2.9 Programmer-Defined Generators

In addition to the built-in generators of Icon, programmer-defined procedures can be generators. Like built-in operations, procedure calls can produce zero or more results. The result sequence for a procedure call depends on how the procedure is written and on the computations that it performs, not on static specifications.

There are three forms of return from a procedure:

```
return expr
fail
suspend expr
```

The expression

```
return expr
```

returns from the call of the procedure in which it is evaluated. The result of the call is the first result produced by *expr*, if there is one. An example is:

```
procedure max(i, j)
   if i < j then return j
   else return i
end
```

This procedure also can be formulated in the following fashion:

```
procedure max(i, j)
   return (i < j) | i
end
```

Thus, if *expr* succeeds, the effect of the return is the same as it is in most programming languages — a single result (the first result in the result sequence for *expr*) is returned. On the other hand, if *expr* fails, the procedure call fails. This is illustrated by the following procedure, which returns the maximum of i and j if they are different but fails otherwise:

```
procedure ifmax(i, j)
   return (i < j) | (j < i)
end
```

The expression **fail** causes the procedure call to return without producing a result. The use of **fail** is illustrated by the following, more conventional, formulation of the preceding procedure:

```
procedure ifmax(i, j)
   if i = j then fail
   if i < j then return j
   else return i
end
```

If a procedure call returns as a result of evaluating **return** *expr* or **fail**, the call cannot be resumed to produce additional results. On the other hand, the expression

```
suspend expr
```

suspends evaluation within the procedure call and returns the result of evaluating *expr*, but the call can be resumed for additional results. This is illustrated by the following procedure, which models the built-in generator i **to** j:

```
procedure To(i, j)
   while i <= j do {
      suspend i
      i +:= 1
      }
   fail
end
```

Flowing off the end of a procedure body is equivalent to evaluating **fail**, so that the procedure above can be written as follows:

```
procedure To(i, j)
   while i <= j do {
      suspend i
      i +:= 1
      }
end
```

When

```
suspend expr
```

is resumed, *expr* is resumed. If it produces another result, the procedure suspends with that result, and so on. Thus, **suspend** iterates over the result sequence for its argument in the same fashion that **every** iterates over its argument. This aspect of **suspend** often can be used to advantage, as in

```
procedure Interval(i, j)
   if i < j then suspend i to j else suspend j to i
end
```

which produces the sequence of integers between the values of i and j.

Recursion also can be used to produce a sequence of results. An example is

```
procedure Closure(s)
   suspend ("" | (Closure(s) || !s))
end
```

For example, Closure("xyz") has the result sequence {"", "x", "y", "z", "xx", "xy", "xz", "yx", ... }. The first result, the empty string, is provided by the left argument to the alternation control structure. This is followed by the result sequence for

```
Closure(s) || !s
```

which consists of the first result for Closure(s), the empty string, concatenated with the one-character substrings of s, followed by the second result for Closure(s) similarly concatenated with the one-character substrings of s, and so on.

### 3. Result Sequences

A result sequence represents, abstractly, the capability that an expression has to produce results. Result sequences provide a way of conceptualizing a computational process in which results are produced in order as the computation progresses. As such, they provide a programming technique that focuses on the formulation of expressions that have desired result sequences.

The result sequences for many expressions are independent of the time the expressions are evaluated and of factors outside of the expressions themselves. Such expressions are called *self-contained*. Examples are

```
|1
```

and

```
(i := 0) & (i +:= |1)
```

On the other hand, the result sequences for some expressions depend on external factors. An example is

```
|read()
```

The characterizations of result sequences in this section apply only to self-contained expressions.

The following notation is used for describing result sequences:

| | |
|---|---|
| $\mathfrak{S}(expr)$ | the result sequence for *expr* |
| $\ell(expr)$ | the length of the result sequence for *expr* |
| $S_1 \oplus S_2$ | the concatenation of the result sequences $S_1$ and $S_2$ |
| $S^i$ | the concatenation of S with itself *i* times |
| $\Phi$ | the empty result sequence, {} |

In this notation,

$$\mathfrak{S}(\text{if } expr_1 \text{ then } expr_2 \text{ else } expr_3) = \mathfrak{S}(expr_2) \text{ if } \ell(expr_1) > 0 \\ = \mathfrak{S}(expr_3) \text{ if } \ell(expr_1) = 0$$

Similarly,

$$\mathfrak{S}(expr_1 \mid expr_2) = \mathfrak{S}(expr_1) \oplus \mathfrak{S}(expr_2)$$

and

$$\mathfrak{S}(|expr) = \mathfrak{S}(expr)^{\infty} \text{ if } \ell(expr) > 0 \\ = \Phi \text{ if } \ell(expr) = 0$$

and

$$\mathfrak{S}(\{expr_1; expr_2; \ldots; expr_n\}) = \mathfrak{S}(expr_n)$$

All looping control structures have empty results sequences (in a conditional context, they fail). For example,

$$\mathfrak{S}(\text{while } expr_1 \text{ do } expr_2) = \Phi$$

The length of the result sequence for an operation depends on the lengths of the result sequences for its arguments. For a monogenic operation, the length of its result sequence is the product of the lengths of the result sequences for its arguments, since the operation is applied to all possible combination of argument values — the n-tuples in the cross-product of their result sequences. For example,

$$\ell(expr_1 + expr_2) = \ell(expr_1) * \ell(expr_2)$$

Some expressions have rather surprising result sequences. For example,

$$\mathfrak{S}(expr_1 \ \& \ expr_2) = \mathfrak{S}((expr_1, expr_2)) = \mathfrak{S}(expr_2)^{\ell(expr_1)}$$

Similarly,

$$\mathfrak{S}(expr_1 \ \& \ expr_2 \ \& \ expr_3) = \mathfrak{S}(expr_3)^{\ell(expr_1) * \ell(expr_2)}$$

and so on.

Equivalences between the result sequences for different expressions sometimes are useful in program formulation. Examples are:

$$\mathfrak{S}(expr(expr_1) \mid expr(expr_2) \mid \ldots \mid expr(expr_n)) = \mathfrak{S}(expr(expr_1 \mid expr_2 \mid \ldots \mid expr_n))$$

and

$$\mathfrak{S}(expr_1(expr) \mid expr_2(expr) \mid \ldots \mid expr_n(expr)) = \mathfrak{S}((expr_1 \mid expr_2 \mid \ldots \mid expr_n)(expr))$$

See [7] for proofs of these equivalences and related results.

## 4. Programming Techniques

### 4.1 Formulating Sequences

From a programming viewpoint, the objective is synthesis rather than analysis: how to formulate an expression that has a desired result sequence.

#### Initialization

The initialization of a sequence provides an example. Since

$$\mathcal{S}(\{\ expr_1;\ expr_2\ \}) = \mathcal{S}(expr_2)$$

$expr_1$ can be used to initialize $expr_2$. For example, the sequence of positive integers can be characterized by

```
{i := 0; i +:= |1}
```

This formulation has the virtue of being more straightforward than either

```
(i := 0) & (i +:= |1)
```

or

```
(i := 1) | (i +:= |1)
```

Similarly, the result sequence for conjunction suggests a way to replicate a desired sequence a specific number of times. For example, the result sequence {1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4} can be obtained by

```
(1 to 3) & (1 to 4)
```

The same result sequence can be obtained for any first argument that has three results, such as

```
(x | x | x) & (1 to 4)
```

or

```
!"xxx" & (1 to 4)
```

#### Sequences from Cross-Product Evaluation

The last-in, first-out "cross-product" evaluation mechanism of Icon can be used to advantage to produce sequences in which the rightmost component of a value varies over a range. An example is an expression that produces strings representing hexadecimal numbers:

```
!"0123456789abcdef" || !"0123456789abcdef"
```

whose result sequence is {"00", "01", "02", ..., "0f", "10", ..., "ff"}.

#### Modifying Result Sequences

When using result sequences as a basis for programming, one useful technique is to obtain a desired result sequence from another one.

A useful paradigm is *filtering*, in which only those results of a sequence with a specified property are selected. For example, the function integer(x) produces the integer value of x if that value is numeric, but fails otherwise. Thus the result sequence for integer(*expr*) consists of the integer values of the result sequence for *expr*. For example, if

```
a := ["Hello", 5, "World!", 6]
```

the result sequence for !a is {"Hello", 5, "World!", 6}, but the result sequence for integer(!a) is {5, 6}.

Selecting the even-numbered results in a result sequence provides another example of programming techniques:

```
{i := 0; 1(expr, i +:= 1, (i % 2) = 0)}
```

Note that this formulation is valid only if the result sequence for *expr* does not depend on the value of i.

On the other hand, there are no general methods for modifying result sequences in arbitrary ways, short of storing all the results and then producing them from the stored values. Even this technique does not work for expressions with infinite result sequences*. The nature of the problem can be seen from attempting to interleave the results from the result sequences of two expressions. Suppose

$$\mathcal{S}(expr_1) = \{ x_1, x_2, x_3, \ldots \}$$
$$\mathcal{S}(expr_2) = \{ y_1, y_2, y_3, \ldots \}$$

There is no general method of composing an expression out of $expr_1$ and $expr_2$ that has the result sequence

$$\{ x_1, y_1, x_2, y_2, x_3, y_3, \ldots \}$$

The problem is that in any expression containing $expr_1$ and $expr_2$, the implicit last-in, first-out order of resumption causes the production of all of the results of one of these expressions before the second result is produced for the other.

### 4.2 Procedural Encapsulation

One difficulty in using a generator to produce a sequence of results is that its results can only be produced at the lexical site in the program where the generator appears. Thus, if the same sequence is needed at several places in a program, the generator must be duplicated in the program text at every such place.

This problem often can be circumvented by *procedural encapsulation*. Given an expression *expr* and the procedure declaration

```
procedure p()
   suspend expr
end
```

the result sequences for *expr* and p() are the same, provided that *expr* is self-contained, as defined in Section 3.

For example, the expression

```
{i := 0; i +:= |1}
```

whose result sequence is the positive integers, can be encapsulated in the procedure

```
procedure posint()
   suspend {i := 0; i +:= |1}
end
```

Thus, the expression posint() has the same result sequence as the expression:

```
{i := 0; i +:= |1}
```

This procedure call can be used in the program text at any place where the sequence is needed. This technique also has the advantage of isolating the expression in the procedure declaration, providing an abstraction that it easy to locate, modify, and document.

Procedural encapsulation can be extended by providing arguments to parameterize a sequence, as in

```
procedure intseq(j, k)
   suspend {i := j; i +:= |k}
end
```

A procedure that encapsulates an expression also can contain other expressions that do not effect its result sequence. A useful model is

*Results sequences can be manipulated in arbitrary ways by using co-expressions [1], which allow the results from the result sequence of an expression to be produced as desired, instead of being produced automatically as the result of the implicit last-in, first-out order of resumption that occurs in iteration and goal-directed evaluation. Co-expressions are beyond the scope of this report but are described in the next report in this series.

```
procedure p()
   prologue
   suspend expr
   epilogue
end
```

There are many possible variations. For example, a procedure can be instrumented to write out the number of results it produces in the following fashion:

```
procedure p()
   local i
   i := 0
   suspend 1(expr, i +:= 1)
   write(i)
end
```

### 4.3 Logic Programming

While Icon is not specifically oriented toward logic programming in the manner that Prolog is [8], techniques of logic programming often can be applied to the formulation of programs in Icon.

As described earlier, the success or failure of the evaluation of an expression in Icon corresponds to whether or not the expression produces a result. The fact that the expression may produce more than one result is not significant in many contexts. The distinction between success and failure is important in three situations:

1. In the control clauses of control structures such as **if-then-else** and **while-do**.
2. In goal-directed evaluation, where failure of an expression causes the resumption of its subexpressions.
3. When the failure of an expression causes the failure of an enclosing expression.

From a programming viewpoint, success and failure correspond to the Boolean *true* and *false* values used by many other programming languages. The truth value of an expression, denoted by $\tau(expr)$, is defined as follows:

$$\tau(expr) = true \quad iff \quad \ell(expr) > 0$$
$$\tau(expr) = false \quad iff \quad \ell(expr) = 0$$

For example,

$$\tau(expr_1 \mid expr_2) = true \quad iff \quad \tau(expr_1) = true \text{ or } \tau(expr_2) = true$$
$$\tau(expr_1 \ \& \ expr_2) = true \quad iff \quad \tau(expr_1) = true \text{ and } \tau(expr_2) = true$$

Icon also provides a negation control structure to complete the propositional calculus:

$$\tau(\text{not } expr) = true \quad iff \ \tau(expr) = false$$

Thus, expressions can be combined with alternation, conjunction, and negation to provide "logic programming". An example is

$$(expr_1 \mid expr_2) \ \& \ \text{not } expr_3$$

which is "true" (succeeds) if either $expr_1$ or $expr_2$ is "true" and if $expr_3$ is "false".

The use of Icon's expression evaluation mechanism to effect logic programming paradigms is not limited to propositional forms. The **every-do** control structure is closely related to universal quantification, as can be seen from expressions such as

$$\text{every p}((expr_1 \mid expr_2) \ \& \ \text{not } expr_3)$$

Similarly, goal-directed evaluation is related to existential quantification. For example,

$$\tau(expr_1 = expr_2) \quad iff \quad \exists(i,j)(i \in \mathcal{S}(expr_1) \ \& \ j \in \mathcal{S}(expr_2) \ \& \ i = j)$$

These analogies can be used to cast propositional and predicate calculus formulations as expressions in Icon.

**Acknowledgements**

The original version of Icon was designed by Dave Hanson and Tim Korb in collaboration with the author. Cary Coutant and Steve Wampler collaborated in the design of the current version.

Dave Hanson and Janalee O'Bagy made a number of helpful suggestions on the presentation of the material in this report.

**References**

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. A. Newell, *Information Processing Language-V Manual*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1961.
3. B. Liskov, *CLU Reference Manual*, Springer-Verlag, 1981.
4. M. Shaw, *Alphard: Form and Content*, Springer-Verlag, 1981.
5. R. B. K. Dewar, E. Schonberg and J. T. Schwartz, *Higher Level Programming; Introduction to the Use of the Set-Theoretic Programming Language SETL*, Computer Science Department, Courant Institute of Mathematical Sciences, Jan. 1981.
6. R. E. Griswold, "Expression Evaluation in the Icon Programming Language", *1984 ACM Conference on LISP and Functional Programming*, 1984, 177-184.
7. S. B. Wampler and R. E. Griswold, "Result Sequences", *J. Computer Lang. 8*, 1 (1983), 1-14.
8. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

## Exercises

The exercises that follow are designed to give additional insight into the expression evaluation mechanism of Icon. Several exercises assume knowledge of Icon's function and operation repertoire beyond what is covered in this report. Refer to [1] for further information.

### 1. Analyzing Result Sequences

Give the result sequences for each of the following expressions (watch out for "black holes"):

(1) `(1 | 2) to (3 | 4)`

(2) `(1 | 2) & (3 | 4) & 5`

(3) `{1 to 4; 5 to 10}`

(4) `(1 to 3) = (1 to 3)`

(5) `|(1 to 3) | 4`

(6) `(1 to 3) + (1 to 3)`

(7) `(1 to 3) \ (1 to 3)`

(8) `(1 to 5) = (4 to 9)`

(9) `!"Hello || !"World!"`

(10) `(0 | 0) to 7`

(11) `(0 to 3) || (0 to 7) || (0 to 7)`

(12) `{i := 0; (1 to 4) \ (i +:= |1)}`

(13) `(1 to 3) || (1 to 3)`

(14) `(x | y | z) & 0`

(15) `(2 to 4) - (2 to 4)`

(16) `|(2 to 4) \ (2 | 7)`

(17) `"a" \ (1 to 4)`

(18) `(1 to 4) & "a"`

(19) `|?10`

(20) `?|10`

(21) `{i := 0; 1(i +:= |1, *i % 2 = 0)}`

(22) `{i := 0; (i +:= |1, |i \ i)}`

(23) `{i := 0; 1 to (i +:= |1)}`

(24) `{i := 0; (i +:= |1, i to i)}`

(25) `{j := i := 0; (i +:= |1, j +:= i)}`

(26) `{i := 0; |(i +:= 1 | i ^ 2)}`

(27) `(|"a" \ 4)`

(28) `|(1 to 3)(!"abcd",!"efg",!"hi")`

(29) `!"abc" == |"a"`

(30) `!"abc" == |"d"`

(31) `!"abc" == |"c"`

(32) `{i := 0; 1(!"abcd",i +:= 1,i % 2 = 0)}`

(33) `(x := !"abcd",x | "-")`

(34) `{i := 0; 2(j := ?|10, i +:= 1, j = 2)}`

(35) `{i := 0; (1 to 10) = (i +:= 1)}`

(36) `integer(!"37xf0")`

(37) `!|read()`

(38) `integer(|read())`

(39) `integer(!|read())`

**2. Composing Result Sequences**

Give expressions that have the following result sequences:

(1) {1, -1, 1, -1, ... }

(2) {1, 1, 2, 1, 2, 3, 1, 2, 3, 4, ... }

(3) {1, 2, 3, 4, 1, 2, 3, 1, 2, 1}

(4) A randomly distributed sequence of the strings "H" and "T".

(5) The sequence of letters in alphabetical order.

(6) An infinite sequence of randomly selected digits.

(7) The sequence of strings representing times in minutes for a day, starting at midnight: {"00:00", "00:01", ..., "00:59", "01:00", ..., "23:59"}.

**3. Modifying Result Sequences**

Without using procedures, give expressions in terms of an arbitrary expression *expr* that have the following result sequences:

(1) The first result in the result sequence for *expr*, if there is one, otherwise the null value.

(2) The even-length strings in the result sequence for *expr*.

(3) The running sum of integers in the result sequence for *expr*.

(4) The length of the result sequence for *expr*.

(5) The *i*th result in the result sequence for *expr*.

(6) The *i*th result in the result sequence for *expr*, repeated indefinitely.

(7) The *i*th result in the result sequence for *expr*, repeated *i* times.

(8) A result sequence in which every result in the result sequence for *expr* is repeated *i* times.

(9) A two-result sequence consisting of the maximum and minimum integers in the result sequence for *expr*.

**4. Procedures**

1. Write a procedure that generates a sequence of randomly selected integers between 1 and i but terminates if j is produced.

2. Write a procedure that generates an infinite sequence of strings whose characters are randomly chosen from s and whose length is randomly chosen between 1 and i.

3. Write a procedure that generates all the permutations of characters in the string s.

4. Write a procedure that generates all the combinations of i characters taken from the string s.

5. Write a recursive procedure that generates the triangular numbers 1, 3, 6, 10, 15, ....

**5. Miscellaneous Exercises**

In the following exercises, assume all the expressions are self-contained.

1. The expression

$(expr_1$ & $expr_2)$ | $expr_3$ (1)

bears a superficial resemblance to

if $expr_1$ then $expr_2$ else $expr_3$ (2)

(a) Give a formulation for the result sequence for (1) in terms of the result sequences for $expr_1$, $expr_2$, and $expr_3$.

(b) Under what conditions are the result sequences for (1) and (2) the same?

2. Repeat Exercise 1 above for

$((expr_1 \setminus 1)$ & $expr_2)$ | $expr_3$