Generators in Icon

RALPH E. GRISWOLD, DAVID R. HANSON, and JOHN T. KORB
The University of Arizona

Icon is a new programming language that includes a goal-directed expression evaluation mechanism. This mechanism is based on generators—expressions that are capable of producing more than one value. If the value produced by a generator does not lead to a successful result, the generator is automatically activated for an alternate value. Generators form an integral part of Icon and can be used anywhere. In addition, they form the basis for the string scanning facility and subsume some of the control expressions found in other languages. Several examples are given.

Key Words and Phrases: goal-directed programming, generators, nondeterministic programming, backtracking, Icon, SL5, programming languages
CR Categories: 4.2, 4.20, 4.22

1. INTRODUCTION

Icon is a new programming language intended for nonnumerical applications with an emphasis on string and structure processing. Icon has its roots in SNOBOL4 [16] and SL5 [11, 14, 18-20]. It shares many of the philosophical bases of these languages: concise, expressive features, run-time flexibility, support of untyped identifiers and heterogeneous structures, and automatic type conversions. Icon lacks some of the exotic features of both SNOBOL4 and SL5; in order to provide greater efficiency in the most frequently used operations, Icon restricts run-time flexibility. In this sense, Icon follows the more traditional method of binding many language operations at compile time.

One of the major motivations for Icon is the continuing effort to provide better string manipulation facilities. A complete discussion of the issues involved in this effort is given in [12]. Briefly stated, the major issue is that, in most languages, string manipulation facilities constitute a separate sublanguage, and interaction between the sublanguage and the language proper is awkward at best. In addition, if the facilities include a search and backtrack mechanism, its use is usually restricted to the sublanguage, and the programmer typically has little control over its operation.

One of the major differences between Icon and previous work is that the evaluation mechanism, which includes a limited form of backtracking, is not

This work was supported by the National Science Foundation under grants MCS75-01307 and MCS79-03890.

Authors' addresses: R. E. Griswold and D. R. Hanson, Department of Computer Science, The University of Arizona, Tucson, AZ 85721; J. T. Korb, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, CA 94304.

restricted to the string processing facility but is used everywhere. Unlike SNOBOL4, for example, the string processing facilities can be thought of as a natural consequence of the evaluation mechanism. The central concept in the Icon evaluation mechanism is the *generator*. A generator is an expression that is capable of producing a sequence of values. The first value is produced when the generator is evaluated during ordinary computation. Subsequent values are produced when alternatives are requested. The production of these alternate values is under the control of the programmer, much as the programmer controls ordinary iterative or conditional control structures. It is this integration of generator control with evaluative control that gives Icon its flexibility and power and permits concise, natural solutions to typical search and backtrack problems. Perhaps more important, generators facilitate the composition of conventional and unconventional control structures from more basic constituents. As a result, generators subsume several of the control structures found in traditional languages.

Facilities for programmable search and backtracking appear in several languages, especially languages intended for string processing [8] or AI applications [2]. These goal-directed linguistic facilities tend to suffer from two contrasting problems: They are either cumbersome to use, requiring considerable programming overhead to set up and control the search strategy (as in CONNIVER [30, 37]), or too inaccessible, not allowing the programmer the ability to terminate, prevent, or force the searching for alternatives (see, e.g., PLANNER [23]). Icon generators avoid these problems. Backtracking associated with the production of alternative values is limited in its extent by syntactic constructions. This limited "scope" of generators avoids the inefficiency of uncontrolled backtracking and gives the programmer more explicit control in the search for alternatives. This latter capability is especially important in light of the observation that, in many cases, backtracking is simply not needed [12, 25].

This paper describes the use of generators in Icon, concentrating on their use in applications other than string processing, which is treated in [10]. Implementations of Icon for the DEC-10 and CDC Cyber have been operational since mid-1978, and an implementation for the PDP-11 under UNIX has just been completed. In addition, Icon has been distributed to numerous installations elsewhere. Thus, the generator facility presented in this paper is the product of over two years of evaluation, and many of the design decisions have been based on pragmatic considerations resulting from actual use.

2. OVERVIEW OF ICON

Icon is an expression-oriented language and has a syntax similar to that of ALGOL 68 [38] and PASCAL [42]. Typical control structures include **if-then-else** and **while**. As in SL5, control structures are signal driven. Expressions return a result, which is composed of a value and a signal. The value component is used like ordinary values in other languages. The signal indicates success or failure and is used to control the flow of execution. For example, the expression

while e_1 **do** e_2

repeatedly evaluates e_2 as long as the evaluation of e_1 succeeds. Note that the value of e_2 has no role in controlling the flow of execution.

Standard arithmetic and lexical comparisons signal success or failure as appropriate. For example,

if $x < y$ **then** x :=: y

swaps the values of x and y if x is less than y.

Typical signal-driven control structures include

if e_1 **then** e_2 **else** e_3

while e_1 **do** e_2

repeat e

$e_1 \mid e_2$

The expression **repeat** e repeatedly evaluates e until it fails. In the expression $e_1 \mid e_2$, the expression e_2 is evaluated only if the evaluation of e_1 fails. The | expression has a more important interpretation in the presence of generators and is described in the next section.

Icon's type structure is similar to that of SNOBOL4 and SL5. There are a number of built-in types and a record-definition facility. The type of a variable may vary during execution, and type coercions are performed automatically as required by context. A more complete overview of Icon is presented in [15]. Complete details are available in the Icon reference manual [13].

3. GENERATORS

Generators form the basis of the goal-directed facilities in Icon. Some typical built-in generators produce an arithmetic sequence of integers, the positions of particular characters in a string, or the elements of a data structure. These values are produced one at a time as demanded by the expression in which the generator appears.

3.1 Goal-Directed Evaluation

An expression containing generators represents a goal, and the success of such an expression signifies that the goal was reached. Failure of an expression, on the other hand, implies that the goal was not reached.

In the absence of failure, operations are evaluated in the standard order: left to right and according to the precedence and associativity of the operators involved. Generators that appear in an expression are evaluated, produce their first value, and become dormant. A *dormant generator* is an operation whose evaluation has produced (at least) one value and which has the capacity to produce more, but which is not presently being evaluated.

Failure of an operation initiates backtracking, in which dormant generators are activated to produce alternate values. The *activation* of a dormant generator transfers control to the point in the expression where that generator became dormant. The generator produces a new value and evaluation continues from that point.

The success or failure signal is used to control the order of evaluation. As long as operations succeed, evaluation proceeds normally. If an operation fails, however, backtracking is begun in order to seek alternatives that may lead to successful evaluation.

Backtracking occurs in the evaluation of operands of operators and in the evaluation of arguments of functions. For example, in

$$e_1 + e_2$$

e_1 is evaluated first. If it fails, the addition operation fails. If it succeeds, e_2 is evaluated. If e_2 fails, however, the addition operation does not necessarily fail. Instead, backtracking occurs and an alternative value of e_1 is sought. If such an alternative exists, e_2 is evaluated again. Since the evaluation of e_1 may affect e_2 (by means of side effects), e_2 may now succeed. If so, the addition is performed.

Note that only "control" backtracking is done; side effects, such as assignments, are *not* reversed. This definition is less general than the classical meanings of backtracking and nondeterminism [6, 9] but admits programming language facilities that are easier to comprehend, more efficient, and simpler to implement than typical facilities for full backtracking [3, 32].

In the case of a function call such as $f(e_1, e_2)$, if e_2 fails, alternatives are sought for e_1. In fact, if e_1 and e_2 both succeed, but the function itself fails, alternatives are sought for the arguments (first e_2 and, failing that, e_1). If any argument has an alternative, the function is called again. If the function continues to fail, it is called for all alternative values of the arguments. The overall expression fails only if the function fails for all alternative values of the arguments. This method of evaluation applies regardless of the number of arguments in the function call.

In the absence of dormant generators, failure of an operation terminates evaluation of the expression in which it appears. More precisely, failure of an operation causes control to be transferred to the next *expression boundary*. Expression boundaries occur in two places: after expressions in control structures and after expressions separated by semicolons. For example, in the expression below boundaries are marked by arrows.

$$\textbf{if } a < b < c \underset{\uparrow}{\ } \textbf{ then } \{x := a; \underset{\uparrow}{\ } y := c \underset{\uparrow}{\ }\}$$

Similarly, in the three expressions

$$\{e_1; \underset{\uparrow}{\ } e_2; \underset{\uparrow}{\ } e_3 \underset{\uparrow}{\ }\}$$

boundaries appear after each expression. Failure of e_2, for instance, causes control to be transferred to the expression boundary following e_2. Evaluation then continues with expression e_3.

This handling of "undetected failure" is motivated by experience with Icon that suggests it is simply more useful than, say, treating undetected failure as an error as in SUMMER [25]. Many problems are composed of "segmented" goals in which failure may be a normal result. Forcing such goals into success-oriented molds limits flexibility and tends to negate the concise expression of intent. Errors resulting from "unwanted failure" have proved to be infrequent in practice, especially after the programmer learns to make use of undetected failure.

3.2 Built-In Generators

A simple generator is the alternation

$$e_1 \mid e_2$$

In this expression e_1 is evaluated first. If e_1 succeeds, its value is returned. If e_1 fails, e_2 is evaluated. If e_2 succeeds, its value is returned; otherwise, the expression fails. For example,

$$x = 5 \mid x = 10$$

succeeds if x equals 5 or if x equals 10.

In addition to the usual meaning of alternation described above, alternation is also a generator, generating the values of each of its arguments. If e_1 succeeds, the alternation expression becomes dormant. The alternation expression can be activated to evaluate e_2 if the expression in which it appears fails. Thus, the expression above can be rewritten as

$$x = (5 \mid 10)$$

After generating the first value, 5, the generator becomes dormant, and the value 5 is compared to x. If x equals 5, the expression succeeds. If not, the generator is activated and produces the value 10, which is compared to x. If x equals 10, the expression succeeds; otherwise it fails. In either event, evaluation then continues with the next expression in sequence.

As another example, consider the expression

$$x < ((1 \mid 2) + y)$$

where x is 4 and y is 3. In this example, as in many others involving the alternation generator, the expression $1 \mid 2$ is usually read "one then two" rather than the more conventional "one or two." The initial comparison of x to 4 (the sum of 1 and y) fails. Backtracking causes the alternation to produce its second value, the comparison of x and 5 succeeds, and the expression terminates successfully.

The evaluation mechanism often permits the internalization of loops. An example is the following expression, which is one way to determine whether n is divisible by any of several small primes.

$$mod(n, p := 2 \mid 3 \mid 5 \mid 7 \mid 11) = 0$$

If any of the primes divides n, the expression succeeds, and p is assigned the divisor; otherwise, the expression fails. If the value of p is not needed, the expression can be further simplified to

$$mod(n, 2 \mid 3 \mid 5 \mid 7 \mid 11) = 0$$

While alternation provides a convenient way to generate a pair of values, a sequence of values can be produced using the generator

$$e_1 \textbf{ to } e_2$$

which generates the arithmetic sequence of integers from e_1 through e_2, inclusive. For example, the expression

$$5 \textbf{ to } 10$$

generates the sequence 5, 6, 7, 8, 9, 10. An optional **by** clause may be specified to give an increment (or decrement) other than 1. For example,

100 **to** 1 **by** −1

generates the decreasing sequence of integers from 100 down to 1.

Another example is the expression

$p(i := 1$ **to** $5, j := 1$ **to** $5)$

which evaluates procedure p at differing values of i and j as long as $p(i, j)$ fails. The value of j varies most frequently, yielding the sequence of calls $p(1, 1)$, $p(1, 2), \ldots, p(5, 5)$. The expression terminates successfully when $p(i, j)$ succeeds for some i and j. If $p(i, j)$ does not succeed, the expression fails after p has been called with all combinations of i and j.

Generators can be used to index through the elements of an object. The element generator

$!e$

generates the elements of object e. The value of e may be a structure, string, or file. For example, if x and y are lists of numbers, the expression $!x = !y$ succeeds if the lists have any equal values. All the elements of y are generated and compared to each element of x. The elements are generated and compared until either two equal values are found or all elements have been generated.

If e is a string (or is convertible to a string), its characters are generated, one at a time. Thus, the expression

$!i > 5$

succeeds if the integer i (after conversion to string) contains a digit greater than 5.

Finally, if e is a file, $!e$ generates the lines in that file. For example, the following expression succeeds if string s appears in file f:

$s == !f$

The operator $==$ succeeds if its operands are identical strings.

In addition to the alternation, sequence, and element generators, there are several built-in generators for string analysis. In general, these functions perform a simple lexical analysis of a string. As generators, they may produce alternate values. The built-in function *find*(s_1, s_2) returns the position of the first occurrence of s_1 in s_2, or fails if s_2 does not contain the substring s_1. For example,

find("ab", "abaabbaaabbbaaaabbbb")

returns the value 1 initially. If the generator is activated for alternatives, *find*(s_1, s_2) generates the next position of s_1 in s_2. Repeated activation of the generator above produces the sequence 1, 4, 9, 16.

Another example is *upto*(c, s), which returns the position of any character in the character set c that appears in s. For example,

upto("aeiou", "kaleidoscope")

returns the value 2 initially. As a generator, *upto*(c, s) returns successive positions of characters in c in s. The example above yields the sequence 2, 4, 5, 7, 10, 12.

In an expression containing several generators, each is invoked, produces a value, and becomes dormant in the order determined by its position in the expression. Failure initiates backtracking. Dormant generators are activated in a last-in, first-out order: The generator that most recently became dormant is the first one activated to produce an alternate value. If that generator has no alternatives, the next dormant generator is activated. This process continues until some generator produces a successful alternate or until no dormant generators remain in the expression. If a successful alternate is found, evaluation continues from that point. Otherwise evaluation of the expression is terminated, and control passes to the next expression boundary.

The interaction of generators is illustrated by the expression

$$x = (1 \mid 2) + (y \mid z)$$

which contains two generators, $1 \mid 2$ and $y \mid z$. Assume the variables x, y, and z have the values 6, 3, and 4, respectively. Evaluation causes x to be compared with the sums $1 + y$, $1 + z$, $2 + y$, and $2 + z$, in that order, until the comparison succeeds. In this example, the comparison succeeds on the last alternative, $2 + z$.

Alternation operations can be combined to produce longer sequences of values. For example,

$$5 \mid 9 \mid 2 \mid 4$$

produces the sequence 5, 9, 2, 4. Groups of alternations can be used to produce complex comparisons. For example, the expression

$$(a \mid b \mid c) = (w \mid x \mid y \mid z)$$

compares the values of a, b, and c with those of w, x, y, and z. If any of the values on the left is equal to any of the values on the right, the expression succeeds; otherwise it fails.

As demonstrated by this latter example, alternation facilitates "cross-product" processing; a is compared to w, x, y, and z, as are b and c. There is not, however, a corresponding operation for "dot-product" processing. For example, there is no similar construct for comparing a to x, b to y, and c to z. Some processing of this kind can be done using additional operators proposed in [26].

The absence of such operators in the current version of Icon stems from the observation that they are generally less useful than the alternation-style operators. For example, in string scanning, as in many problems, there is a single focus of attention to which the current approach is ideally suited. Problems having multiple foci of attention can be easily imagined, but their solutions often pose additional complications beyond what can be handled by dot-product-style processing. The situation is much like that for pipes as implemented in the UNIX shell [34]. Pipes facilitate a linear array of processing and are applicable to only those problems that "fit" the model. The models underlying Icon and UNIX are derived from experience and have evolved to meet specific needs but are not so orthogonal and complete as might be desired on intellectual grounds. Current research is directed toward orthogonality and is focusing on the use of multiple independent generators to support dot-product-style processing.

3.3 Controlling Generators

Icon provides several constructs to control the evaluation of generators. These constructs can be used, for example, to suppress alternatives of a generator and to force a generator through all alternatives.

An important aspect of controlling generators is the ability to discard remaining alternatives if they are not needed. Dormant generators are discarded when the boundary of the expression in which they appear is crossed. In this way, generators whose values are no longer relevant are not activated in a futile attempt to make another, unrelated expression succeed. In addition, the space required for information associated with dormant generators is released.

The following expression sequence provides an example.

x := *upto*("aeiou", s_1); y := *find*("begin", s_2)

The first expression returns the position of the first vowel in s_1 (if one exists). The second expression finds an occurrence of the string "begin" in string s_2. The two expressions are independent; if the second expression fails, remaining alternatives in the first are not tried. In fact, once evaluation has passed through the boundary between the two expressions, the dormant *upto* generator is discarded; its alternatives are no longer accessible.

Because control structures form boundaries after their arguments, they can also be used to control the activation of generators. For example,

if x := *upto*("aeiou", s_1) **then** y := *find*("begin", s_2)

is similar to the example above, except that the second expression is only evaluated if the first expression succeeds. As in the previous example, the two expressions are otherwise isolated from one another and cannot interact.

The **every** expression is used to force a generator to produce all its alternatives. The expression

every e_1 **do** e_2

evaluates e_1 and repeatedly reactivates it until all of its alternatives are produced. After each alternative of e_1, the expression e_2 is evaluated. For example, if *sum* is initialized to zero, the expression

every i := !x **do** *sum* := *sum* + i

finds the sum of the elements of list x. Note that, by using the element generator ! instead of explicit indexing, this expression is independent of not only the size of x but also the type of x.

The **every** expression, in conjunction with generators, provides a facility for composing conventional as well as unconventional control structures from more basic units. By using the **to** generator, the equivalent of the ALGOL **for** statement can be obtained. For example, the expression

every i := 1 **to** 100 **do** $y[i]$:= $x[i]$

copies the first 100 elements of list x into list y.

As another example, if x is a list of 100 elements, the expression

every i := (1 **to** 10) | (91 **to** 100) **do** *write*($x[i]$)

writes the first and last ten elements of x.

The **every** expression is more concise than the standard ALGOL **for** loop in certain applications. The **for**-loop control variable is often just an artifact that can be omitted. For example, the expression

```
every 1 to 30 do f()
```

invokes the procedure *f* thirty times.

The **do** clause is optional and can often be omitted, allowing even more concise expressions. For instance, auxiliary variables that are needed in other languages to hold transitory values (such as list indexes) are not always needed in Icon. The above example, which writes the first and last ten elements of list *x*, can be rewritten as

```
every write(x[(1 to 10) | (91 to 100)])
```

Since the element generator generates the elements of a structure as variables, **every** can be used to zero the elements of a structure. For example,

```
every !x := 0
```

zeros the elements of *x*, and

```
every sum := sum + !x
```

computes their sum.

3.4 Programmer-Defined Generators

Icon procedures have the general form

```
procedure name(arg1, . . . , argn)
  declaration
  body
end
```

A typical procedure is

```
procedure sum(x)
  local s, i
  s := 0
  every i := !x do s := s + i
  if s >= 0 then return s
  fail
end
```

which returns the sum of the elements of a structure if the sum is positive and fails otherwise. As illustrated by this example, procedures return values via the expression **return** *e*. If *e* fails, the procedure invocation fails. The **fail** expression is similar to **return** in that it terminates the invocation, but it causes the calling expression to fail. Arguments to procedures are transmitted by value.

The **return** and **fail** expressions cause termination of the procedure activation in which they are executed. The expression

```
suspend e
```

is similar to **return** *e*, except that the procedure activation is left in suspension so that it may be resumed for additional computation. Execution of the procedure body is resumed at the point of suspension if the context in which the procedure invocation occurs requires another alternative. Thus, suspended procedures are dormant generators. The result of the **suspend** expression itself is the null value.

As a simple example, the following procedure behaves like the built-in **to–by** generator with a positive increment.

```
procedure toby(i, n, inc)
  if inc = 0 then inc := 1
  while i <= n do {
    suspend i
    i := i + inc
    }
  fail
end
```

The procedure suspends evaluation for each value in the sequence and fails after the sequence is exhausted.

Procedures may, of course, suspend in more than one place. For example, the following procedure generates the (infinite) sequence of Fibonacci numbers.

```
procedure fib
  local u1, u2, f, i
  suspend 1 | 1
  u1 := u2 := 1
  repeat {
    f := u1 + u2
    suspend f
    u1 := u2
    u2 := f
    }
end
```

The first two values of the sequence, 1 and 1, are treated as special cases in the first **suspend** expression. Subsequent values of the sequence are computed, and the procedure is suspended in the middle of the **repeat** loop after each computation. The following expression repeatedly assigns values of the Fibonacci sequence to f until a value divisible by n is found.

$$mod(f := fib(), n) = 0$$

Like **every**, **suspend** e produces all alternatives of e as required. For example,

suspend 1 | 2 | 5

suspends with the values 1, 2, and 5 on successive activations of the procedure in which it appears. If the procedure is activated again, execution continues with the expression following the **suspend**.

Programmer-defined generators are fully recursive and behave exactly like built-in generators. For example, multiple activaions of defined generators that are suspended simultaneously do not interfere with each other.

4. EXAMPLES

4.1 Word Intersection

In completing crossword puzzles it is necessary to determine the intersections at which two words have characters in common [40]. This is an example in which all possible situations in a search must be processed. The following procedures demonstrate the use of generators to print all intersections of common characters in two words.

```
procedure cross(word1, word2)
  local j, k
  every j := upto(word2, word1) do
     every k := upto(word1[j], word2) do
        print(word1, word2, j, k)
  return
end

procedure print(word1, word2, j, k)
  every write(right(word2[1 to k - 1], j))
  write(word1)
  every write(right(word2[k + 1 to size(word2)], j))
  return
end
```

The procedure *cross* first assigns to *j* a position in *word*1 at which a character of *word*2 is found. For this value of *j*, a position in *word*2 where the *j*th character of *word*1 occurs is then assigned to *k* (subscripting a string returns the character at the position given by the value of the subscript). The procedure *print* merely prints the intersection, with *word*1 being displayed horizontally and *word*2 being displayed vertically (*right*(*s*, *n*) returns the result of right-justifying *s* in a string of *n* blanks). The uses of **every** ensure that all intersections are located.

As an example,

cross("computer", "center")

produces the following output.

```
computer         c          c          c          c
e                e   computer          e          e
n                n          n          n          n
t         computer          t          t          t
e                e          e   computer          e
r                r          r          r   computer
```

4.2 Word Generation

The procedure *getword*(*f*) generates the words that appear in file *f*. A word is defined to be any sequence of lowercase letters.

```
procedure getword(f)
  local i, line
  while line := read(f) do {
    i := 1
    while i := upto("abc ··· z", line, i) do
       suspend section(line, i, i := many("abc ··· z", line, i))
    }
  fail
end
```

The procedure *section*(*s*, *i*, *j*) returns the section of string *s* between positions *i* and *j*; the third argument to *upto* specifies the position in *s* at which to begin examination; and *many*(*c*, *s*, *i*) returns the position in string *s* beginning at position *i* that follows an initial substring of characters in *c*.

One use of such a procedure is illustrated by counting the words in a file. The expression

```
ftab := table(0)
```

assigns an empty table to *ftab*; tables are associative arrays as in SNOBOL4. The expression

```
every ftab[getword(f)]+
```

builds a table containing the count of each word appearing in *f*. The suffix + operator increments its operand by 1. Finally,

```
every x := !ftab do write(x[2], " ", x[1])
```

prints the table. (Sequencing through a table returns each entry as a two-element list. The first element is the "index" and the second is the associated value.)

4.3 Data Structure Traversal

Generators can also be used to generate the elements of a user-defined data structure. Below are two procedures that generate the nodes of a binary tree. Nodes of the tree are represented using records with three fields:

```
record node
  data, ltree, rtree
end
```

The *data* field contains the value of the node. The fields *ltree* and *rtree* refer to the left and right subtrees, respectively, and may have null values. The built-in function *null*(*x*) succeeds if *x* has the null value and is used to test for leaf nodes.

The procedure *walk*(*t*), which follows, generates the data fields of all the nodes in tree *t*. The nodes are generated in postorder; that is, the left and right subtrees of a node are generated, followed by the node itself. The infix dot accesses the fields of a record as in PASCAL [42].

```
procedure walk(t)
  if null(t) then
     fail
  else {
    suspend walk(t.ltree | t.rtree)
    return t.data
    }
end
```

The **suspend** expression suspends with each of the values generated by walking the left subtree and then with each of the values generated by walking the right subtree. The **return** expression produces the value of the root node.

The procedure *leaves*(*t*) below generates only the leaves of the binary tree *t*. The leaves are generated by walking the tree in postorder.

```
procedure leaves(t)
  if null(t) then
    fail
  else if null(t.ltree = = = t.rtree) then
    return t.data
  else {
    suspend leaves(t.ltree | t.rtree)
    fail
    }
end
```

The = = = operator denotes object comparison much like IDENT in SNOBOL4. As for all comparison operators, if the comparison succeeds, the right operand is returned.

4.4 Data Backtracking with Generators

As mentioned above, the backtracking in Icon is limited to control backtracking. There are, however, several built-in operators that provide a limited form of data backtracking. Note, however, that the control backtracking is implicit in Icon's evaluation mechanism, whereas data backtracking must be explicitly specified by the programmer by the use of an operator.

The most useful of the data backtracking operators is reversible assignment:

$$e_1 \leftarrow e_2$$

The value of e_2 is assigned to e_1 and the assignment operator becomes dormant. If activated for alternatives, the previous value of e_1 is restored and the expression fails. For example, suppose a program is to read a parameter from the input file and assign the value to a global variable. If the line is of zero length, a default value, which was assigned earlier, is to be retained. The expression

$$size(param \leftarrow read()) \sim= 0$$

assigns the next input line to the variable *param*. If the line is empty, the assignment is reversed and the original (default) value is retained. The use of reversible assignment obviates the use of a temporary variable.

Although Icon provides only limited backtracking, more elaborate backtracking facilities can be constructed using programmer-defined generators. An example is the eight queens problem [5, 6, 9]. The object is to place eight queens on a chess board such that no queen may attack any other. The standard method of solving this problem uses a recursive procedure to place queens on successive rows, backtracking to a previous position if a row becomes blocked [41].

An alternate solution, using coroutines, eliminates the awkward hierarchical relationship among the queens and allows the main program to control the backtracking [19]. Each coroutine is still responsible for reversing the effects it causes, but the main program determines when that is to be done.

The following procedure is similar to the solution in [19] but is further simplified by the use of generators.

```
procedure q(c)
  local r
  every place(r := 1 to 8, c) do
    suspend r
  fail
end
```

The value returned by *q*(*c*) is the row on which queen *c* was placed. The expression

write(*q*(1), *q*(2), *q*(3), *q*(4), *q*(5), *q*(6), *q*(7), *q*(8))

generates and writes the first solution

```
15863724
```

(multiple arguments to *write* are concatenated onto the output file); inserting this expression in an **every** expression produces all 92 solutions. The procedure *place* is a generator that tests and occupies the sequence of available board positions for a given queen:

```
procedure place(r, c)
  if row[r] = up[r − c] = down[r + c] = 0 then
    suspend rows[r] ← up[r − c] ← down[r + c] ← 1
  fail
end
```

Following [41], three lists are used to keep track of the free rows, upward facing diagonals, and downward facing diagonals. If the square at row *r* and column *c* is free, then each of the list elements *rows*[*r*], *up*[*r* − *c*], and *down*[*r* + *c*] is 0; otherwise, at least one is 1. The **suspend** expression initially assigns a nonnull value (the integer 1) to each of the three list elements and suspends. If *place* is activated, the **suspend** expression activates the dormant reversible assignment operations, which restore the three list elements to 0. After restoring the variable, the reversible assignment operation fails and the procedure fails.

5. RELATED WORK

The concept of generators has appeared in various forms in many languages. An early language that has generators is IPL-V [31], in which a generator is a subroutine that calls a processing routine to operate on each object in a data structure. Other languages, such as ALGOL 68 [38] and COBOL, use the term generator to describe various language features, but this use is unrelated to the Icon notion.

Languages that support coroutines frequently use the term generator to describe a particular coroutine usage. SIMULA [1] is the oldest of such languages; more recent examples include extensions to POP-2 [24] and SL5 [20] and coroutine additions to PASCAL [27]. There also has been a substantial amount of work on the "lazy evaluation" approach to the incremental generation of sequences, usually in LISP or APL frameworks [7, 17, 22].

More recently, ALPHARD [43] and CLU [28] have been developed to support data structure abstraction. In these languages generators are used to iterate over the elements of programmer-defined data structures [35]. Included with the definition of a data abstraction (or "cluster" in CLU) is a procedure for generating the elements of the abstraction. In CLU, this procedure, called an iterator, produces its values using the **yield** statement. When another element is required, execution of the iterator continues where the **yield** statement left off. In each of these languages, generators are only accessible in a specific context: a particular type of **for** statement. This is significantly more restricted than in Icon, where generators may appear anywhere.

Superficially, generators in ALPHARD and CLU appear to correspond to the **every** expression in Icon. But while **every** has the syntactic appearance of a standard control structure, the nonstandard goal-directed evaluation mechanism makes its semantics more general than the corresponding ALPHARD and CLU constructs. It is the restricted applicability and the lack of goal-directed evaluation that differentiate these constructs from generators in Icon.

Backtracking mechanisms exist in a number of programming languages. Languages intended for research in artificial intelligence typically include such facilities [2]. Backtracking is also used for solving string processing problems [8].

Most artificial intelligence languages deal with lists and are usually based on the syntax of LISP [29]. PLANNER [23] and PROLOG [4, 39], which use theorem-proving techniques for AI research, provide pattern-directed procedure invocation. In PROLOG, for example, the theorem to be proved is used to select the procedures to invoke. Associated with each procedure is a template that defines the effect of the procedure. The theorem is compared to each template, and those procedures that match are selected. If only one exists, it is invoked. If more than one procedure matches, a choice is made and the selected procedure is invoked. If that procedure fails to prove the theorem, the system backtracks and a different procedure is invoked. PROLOG also has a mechanism, the cut determiner, that is analogous to expression boundaries in Icon and gives the programmer some control over the extent of backtracking. Another example is CONNIVER [30], which is based on PLANNER and gives the programmer greater control over the search strategy.

SAIL [33] is an ALGOL-based artificial intelligence language that supports backtracking. The backtracking facilities include primitives to REMEMBER, FORGET, and RESTORE variables in different contexts. Like CONNIVER, SAIL requires explicit manipulation of variables by the programmer to effect a backtracking control regime.

6. CONCLUSIONS

Generators provide a linguistic facility that allows concise expression of solutions to problems suitable for goal-directed programming. These problems are typically found in string processing, artificial intelligence, and other areas in which combinatorial searching is performed. Experience with Icon has demonstrated its utility in string processing and in problems involving simple combinatorial searches. Problems related to artificial intelligence research have not yet been studied.

The significant contribution of Icon is the development of unified linguistic facilities in which goal-directed evaluation is an integral part. In other languages supporting goal-directed programming, for example, MLISP2 [36] and ECL [32], the facilities are not so well integrated into the language. For example, these languages contain special statements for marking decision points and initiating backtracking. In Icon, generators are expressions and can be used anywhere a value is required. The manipulation and activation of generators are integrated using the control structures and signaling mechanism of the language.

Another advantage of generators stems from their use in constructs that are not usually considered related to goal-directed programming. For example, the **every** expression forces all alternatives of generators to be evaluated, not just the ones needed to reach a goal. When used with the alternation and **to** generators, **every** subsumes the less general **for** statement found in other languages. In addition, generators can be used to build more complex control structures from simpler ones. This construction is especially evident in the use of programmer-defined generators to augment the repertoire of built-in generators.

Some of the inherent inefficiencies of goaldirected facilities in other languages are eliminated by avoiding automatic backtracking and by limiting the scope of generators. Experience has shown that backtracking often is not necessary, that it can be the source of hidden inefficiencies, and that it often hinders the solution of a problem by destroying information.

The scope of generators is limited to the expression in which they appear. Since expressions may be arbitrarily complex, this places no restrictions on the programmer, but it allows unwanted alternatives of dormant generators to be explicitly discarded. Limiting the scope of generators not only prevents unwanted activation, but it also permits the space associated with them to be released. Once an expression boundary is crossed, the space occupied by dormant generators for that expression is no longer needed. Finally, generators have little impact on the efficiency of other language operations; very little overhead associated with generators is imposed on expressions that do not contain generators. The implementation of generators is described more fully in [21, 26].

REFERENCES

1. Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. *SIMULA Begin* (Student Literature). Auerbach, Philadelphia, 1973.
2. Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence research. *Comput. Surv.* (ACM) *6*, 3 (Sept. 1974), 153-174.
3. Bobrow, D.G., and Wegbreit, B. A model and stack implementation of multiple environments. *Commun. ACM 16*, 10 (Oct. 1973), 591-603.
4. Bowen, K.A. PROLOG. In *Proc. 1979 Ann Conf.* (Detroit, Mich., Oct. 29-31, 1979), ACM, New York, pp. 14-23.
5. Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming.* Academic Press, London, 1972, pp. 72-82.
6. Floyd, R.W. Nondeterministic algorithms. *J. ACM 14*, 4 (Oct. 1967), 636-644.
7. Friedman, D.P., and Wise, D.S. CONS should not evaluate its arguments. In *Automata, Languages, and Programming*, S. Michaelson and R. Milner, Eds., Edinburgh Univ. Press, Edinburgh, 1976, pp. 257-284.
8. Gimpel, J.F. *Algorithms in SNOBOL4.* Wiley, New York, 1976.
9. Golomb, S.W., and Baumert, L.D. Backtrack programming. *J. ACM 12*, 4 (Oct. 1965), 516-524.

10. GRISWOLD, R.E. The Icon programming language: A new approach to high-level string processing. In *Proc. 1979 Ann. Conf.* (Detroit, Mich., Oct. 29-31, 1979), ACM, New York, pp. 8-13.
11. GRISWOLD, R.E. The SL5 programming language and its use for goal-directed programming. In Proc. 5th Texas Conf. on Computer Systems, Univ. Texas, Austin, Oct. 1976.
12. GRISWOLD, R.E., AND HANSON, D.R. An alternative to the use of patterns in string processing. *ACM Trans. Program. Lang. Syst. 2*, 2 (April 1980), 153-172.
13. GRISWOLD, R.E., AND HANSON, D.R. Reference manual for the Icon programming language. Tech. Rep. TR 79-1a, Dep. Computer Science, Univ. Arizona, Tucson, Feb. 1980.
14. GRISWOLD, R.E., AND HANSON, D.R. An overview of SL5. *SIGPLAN Notices* (ACM) *12*, 4 (April 1977), 40-50.
15. GRISWOLD, R.E., HANSON, D.R., AND KORB, J.T. The Icon programming language: An overview. *SIGPLAN Notices* (ACM) *14*, 4 (April 1979), 18-31.
16. GRISWOLD, R.E., POAGE, J.F., AND POLONSKY, I.P. *The SNOBOL4 Programming Language*, 2d ed. Prentice-Hall, Englewood Cliffs, N.J., 1971.
17. GUIBAS, L.J., AND WYATT, D.K. Compilation and delayed evaluation in APL. In Conf. Rec., 5th Ann. ACM Symp. on Principles of Programming Languages, Tucson, Ariz., Jan. 23-25, 1978, pp. 1-8.
18. HANSON, D.R. Procedure-based linguistic mechanisms in programming languages. Ph.D. dissertation, Dep. Computer Science, Univ. Arizona, Tucson, 1976.
19. HANSON, D.R. A procedure mechanism for backtrack programming. In *ACM '76: Proc. Ann. Conf.*, Houston, Tex., Oct. 20-22, 1976, pp. 401-405.
20. HANSON, D.R., AND GRISWOLD, R.E. The SL5 procedure mechanism. *Commun. ACM 21*, 5 (May 1978), 392-400.
21. HANSON, D.R., AND HANSEN, W.J. Icon implementation notes. Tech. Rep. TR79-12a, Dep. Computer Science, Univ. Arizona, Feb. 1980.
22. HENDERSON, P., AND MORRIS, J.H. JR. A lazy evaluator. In Conf. Rec., 3d ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 19-21, 1976, pp. 95-103.
23. HEWITT, C. PLANNER: A language for manipulating models and proving theorems in a robot. In Proc. 2d Int'l Joint Conf. on Artificial Intelligence, London, 1971, pp. 167-182.
24. KAHN, G., AND MCQUEEN, D.B. Coroutines and networks of parallel processes. In Proc. IFIPS 77, 1977, pp. 993-998.
25. KLINT, P. An overview of the SUMMER programming language. In Conf. Rec., 7th Ann. ACM Symp. on Principles of Programming Languages, Las Vegas, Nev., Jan. 28-30, 1980, pp. 47-55.
26. KORB, J.T. The design and implementation of a goal-directed programming language. Ph.D. dissertation, Univ. Arizona, Tucson, 1979.
27. LINDSTROM, G. Backtracking in a generalized control setting. *ACM Trans. Program. Lang. Syst. 1*, 1 (July 1979), 8-26.
28. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564-576.
29. MCCARTHY, J., ABRAHAMS, P., EDWARDS, D., HART, T., AND LEVIN, M. *LISP 1.5 Programmer's Manual*, 2d ed. MIT Press, Cambridge, Mass., Feb. 1965.
30. MCDERMOTT, D.V., AND SUSSMAN, G.J. The CONNIVER reference manual. AI Lab. Memo 259, MIT, Cambridge, Mass., 1972.
31. NEWELL, A. (ED.) *Information Processing Language-V Manual*, Rand Corp., Prentice-Hall, Englewood Cliffs, N.J., 1961.
32. PRENNER, C.J., SPITZEN, J.M., AND WEGBREIT, B. An implementation of backtracking for programming languages. *SIGPLAN Notices* (ACM) 7 (Nov. 1972), 36-44.
33. REISER, J.F. SAIL. Tech. Rep., Stanford AI Lab., Computer Science Dep., Aug. 1976.
34. RITCHIE, D.M., AND THOMPSON, K. The UNIX timesharing system. *Commun. ACM 17*, 16 (July 1974), 365-375.
35. SHAW, M., WULF, W.A., AND LONDON, R.L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM 20*, 8 (Aug. 1977), 553-564.
36. SMITH, D.C., AND ENEA, H.J. Backtracking in MLISP2. In Proc. 3d Int'l Joint Conf. on AI, Stanford, Calif., 1973, pp. 677-685.
37. SUSSMAN, G.J., AND MCDERMOTT, D.V. From PLANNER to CONNIVER—A genetic approach. In Proc. AFIPS 1972 Fall Joint Computer Conf., vol. 41, AFIPS Press, Arlington, Va., 1972, pp. 1171-1179.

38. VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., AND FISKER, R.G. (Eds.). Revised report on the algorithmic language Algol 68. *Acta Inf. 5* (Jan. 1976), 1-236.
39. WARREN, D.H.D., PEREIRA, L.M., AND PEREIRA, F. PROLOG—The language and its implementation compared with LISP. In Proc. Symp. on Artificial Intelligence and Programming Languages, Rochester, N.Y., Aug. 1977, pp. 109-115.
40. WETHERELL, C. *Etudes for Programmers.* Prentice-Hall, Englewood Cliffs, N.J., 1978.
41. WIRTH, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
42. WIRTH, N. The programming language Pascal. *Acta Inf. 1* (Jan. 1971), 35-63.
43. WULF, W.A., LONDON, R.L., AND SHAW, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Softw. Eng. SE-2,* 4 (Dec. 1976), 253-265.

Received May 1979; revised June and November 1980; accepted November 1980