String Scanning in the Icon Programming Language

R. E. GRISWOLD
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, USA

As a consequence of the primary historical role of computers as a tool for numerical computation, most programming languages have minimal facilities for string manipulation. This paper discusses some of the language design and implementation issues involved in incorporating string scanning, analysis and synthesis facilities into a general-purpose programming language and describes how they have been introduced in the Icon programming language.

Received January 1989

1. INTRODUCTION

Most programming languages have limited and relatively low-level facilities for processing text – strings of characters. This situation is partly a consequence of the primary historical role of computers, which is dominated by numerical computation. It also is the result of the lack of a strong conceptual basis for string processing prior to the advent of modern computers. While numerical computation has a long history and a vast body of experience and notational conventions from which programming language designers can draw, the manipulation of textual data prior to modern computers was limited and largely unsystematic. In other words, there are few models for string computation upon which to build programming languages. Nor has the architecture of most computers provided much help.

Consequently, most programming languages have minimal facilities for string manipulation, and these facilities often are modelled after numerical computation (such as operations on arrays of characters), instead of providing more powerful facilities at a conceptual level closer to the problem domains in which string processing is important. This, in turn, has made nonnumerical computation difficult and tedious, tending to limit the scope of problems that are attempted.

There were some early attempts to provide higher-level facilities for string processing. COMIT [1], which was motivated by early work on translating natural language with computers, introduced the concept of strings of textual objects (such as words) and contained the germ of the idea of focussing attention on one string at a time. COMIT also introduced the idea of patterns to characterize string structure from which analysis was driven. SCL [2] extended these ideas to using strings of characters to represent mathematical expressions. SNOBOL [3] introduced the notion of a string of characters as a data object in its own right (as opposed to an array of characters) and provided a more uniform and general characterization of string data and patterns. SNOBOL4 went the additional step of treating patterns as first-class data objects and providing operations for constructing complex patterns out of simpler ones [4]. Other interesting approaches include SUMMER [5] and features of some versions of LISP [6]. Nonetheless, most commonly-used programming languages still have relatively low-level and awkward string processing facilities.

The Icon programming language [7] is a high-level, general-purpose programming language with a large repertoire of facilities for string processing. It includes a high-level string processing facility, called *string scanning*. This facility, which has its roots in SNOBOL4-style pattern matching, provides greater flexibility than earlier forms of pattern matching. At the same time, it is integrated with conventional forms of computation – something lacking in earlier programming languages [8].

This paper describes string scanning and related aspects of string analysis and synthesis in Icon. It also discusses some of the programming language design and implementation issues related to these facilities. The elements of Icon that are needed in the description of string scanning are reviewed here. However, a more thorough understanding of Icon [7] may be helpful in appreciating the subtler aspects of some of the examples.

2. STRING SCANNING

The design of string scanning in Icon is based on the long-standing observation that most kinds of string analysis can be cast in terms of a succession of computations on one string at a time. A string, called the *subject*, is the focus of attention during a string scanning operation. Different string scanning operations may, of course, have different subjects.

Another observation is that most string analysis operations can be cast in terms of the examination of the subject at a particular position. This position may be changed as a result of the examination – hence the term *scanning*. Usually the specific position is not itself of interest, but rather what is there and what is around it. This is analogous to climbing to the top of a hill to get a better view of the surrounding countryside. The geographical coordinates of the top of the hill probably are not important or even known; the important thing is that it is a local high point. Similarly, in analyzing a sentence, certain words and their relative positions may be important, but their specific locations in the sentence usually are not.

String scanning is based on a pair of implicit variables that form a *scanning environment*: {subject,position}. The subject normally (but not necessarily) remains fixed during a string scanning operation, while the position usually changes as the result of the analysis of the subject. For example, the act of locating a particular substring moves the position to where this substring begins.

One consequence of this model is that string analysis operations usually can be written without any explicit reference to the subject or position. This reduces clerical detail and tedious arithmetic computations that are necessary in lower-level forms of string analysis. It is no more necessary to increment a pointer to get to a desired substring than it is to count footsteps getting to the top of a hill.

The form of a string scanning expression in Icon is

$expr_1$? $expr_2$

where the evaluation of $expr_1$ produces the subject and $expr_2$ does the scanning. For convenience, the expression that does the scanning ($expr_2$ here) is called the *analysis expression* although, as subsequent examples illustrate, it is not limited to purely analytic computations.

The position in string scanning initially is 1, which is at the beginning of the subject. For example, in

```
term ? expr
```

if the value of term is "a*(b − c)", the scanning environment in which *expr* operates initially is {"a*(b − c)",1}.

The idea of scanning is illustrated by the function move(i), which increments the position in the subject by i and returns the substring of the subject between the previous position and the new one. For example,

```
text ? while move(1) do
  write(move(1))
```

writes the even-numbered characters in text. The function move fails (and does not change the position) if the new position would be outside the subject. This causes the loop above to terminate when the end of the subject is reached.

Note that this analysis expression does not depend on the value of text. The variable text itself is not mentioned in the analysis expression, and all changes to the position are implicit. This abstraction can be seen more clearly if the analysis expression is encapsulated in a procedure:

```
procedure write_even()
  while move(1) do
    write(move(1))
end
```

This procedure then can be called wherever it is needed, as in

```
text ? write_even()
```

3. STRINGS IN ICON

Before describing string scanning in more detail, it is necessary to understand how Icon treats strings. Strings in Icon are sequences of characters. Characters themselves are not values in Icon and strings are not arrays of characters. Instead, strings are values in their own right. Strings may be arbitrary long and can be represented by literals within quotation marks or produced by a variety of operations.

The concatenation of two strings produces a new string, as in

```
noun_ph := "The giant condor"
verb_ph := "seeks its nest"
sentence := noun_ph || " " || verb_ph || "."
```

which concatenates four strings to produce the string "The giant condor seeks its nest.", which is assigned to sentence. It is worth noting that storage management in Icon is automatic and it is not necessary to specify how long a string may be.

Positions in strings are between characters, starting at 1, which is the position to the left of the first character:

```
  c o n d o r
 ↑ ↑ ↑ ↑ ↑ ↑ ↑
 1 2 3 4 5 6 7
```

For convenience in referring to characters with respect to the right end of a string, there are corresponding nonpositive position specifications:

```
   c  o  n  d  o  r
 ↑  ↑  ↑  ↑  ↑  ↑  ↑
−6 −5 −4 −3 −2 −1  0
```

In Icon notation, s[i:j] is the substring of s between positions i and j. Substrings are specified by position pairs. For example, if the value of bird is "condor", the value of bird[2:4] is "on". Positive and non-positive specifications can be mixed and the order of the positions are irrelevant: positions −3 and 2 specify the same substring as 2 and 4, as do −3 and −5.

4. EXPRESSION EVALUATION

Icon has an expression evaluation mechanism with a number of unusual characteristics that are centrally important in string scanning.

Unlike most programming languages, the evaluation of an expression in Icon may fail to produce a value. This is illustrated by move(i), as described above, which fails if the resulting position would be out of the range of the subject. Similarly, a substring specification fails if a position is out of range. Failure, not Boolean values, drives control structures in Icon. As suggested above, the while-do control structure terminates if its control expression fails. Similarly, if an expression that produces the argument of a function fails, the function is not called. Thus, write(move(1)) does not write anything if move(1) fails.

An expression that does not produce a value is said to *fail*, while one that produces a value is said to *succeed*. It is important to understand that failure is a normal aspect of expression evaluation in Icon, not the indication of an error. Failure is used to check bounds, terminate loops, and generally as a mechanism for controlling program flow.

Some Icon expressions, called *generators*, are capable of producing more than one result. This is a natural concept in string analysis. The location of substrings is an example: The substring "o" occurs at two positions in "condor"; 2 and 5. The function find(s) performs this operation, generating the positions, from left to right, at which s occurs in the subject.

A generator produces one value at a time, suspending evaluation every time it produces a value so that it can be resumed if another value is needed. If only one value is needed from a generator, only one is produced. For example if the value of bird is "condor",

```
bird ? write(find("o"))
```

just writes 2, even though there is another position at which "o" occurs in bird.

Icon has several generators. For example, i to j by k generates the integers from i to j in increments of k, and !s generates all the one-character substrings of s.

The need for multiple values from generators arises from *goal-directed evaluation*, in which suspended generators are resumed if they are contained in an expression that otherwise would fail. This is illustrated by

```
"condor" ? write(find("o") = 5)
```

The first value produced by find is 2, which is not equal to 5. The failure of the comparison operation (=) causes find be resumed. It next produces 5, and the comparison succeeds. Comparison operations produce the value of their right operand, so 5 is written.

Icon has many generative expressions. Some, like find, are used in string analysis. One generator of general usefulness is the *alteration* control structure

$expr_1 \mid expr_2$

Alternation generates the values of $expr_1$ followed by the values of $expr_2$. Thus,

```
find("o") | find("or")
```

generates 2, 5, and 5. Alternation can be used to provide a sequence of arguments to find, as in

```
find("o" | "or")
```

which generates the same results as the expression above. Another example is

```
line ? move(10 | 5)
```

In this example, move(10) is attempted first. If line is not long enough, this fails and move(5) is attempted.

As mentioned above, if the evaluation of an argument expression fails, the operation that needs the argument is not evaluated. This provides a way of performing a computation only if several expressions mutually succeed. For example, in

comp($expr_1$, $expr_2$)

comp is only called if both $expr_1$ and $expr_2$ succeed. The *conjunction* operation

$expr_1$ & $expr_2$

provides another form of this kind of evaluation. This expression succeeds (and produces the value of $expr_2$) only if both $expr_1$ and $expr_2$ succeed.

The need to test for mutual success occurs frequently in string scanning. An eample is

```
bird ? (move(3) & find("o"))
```

which succeeds only if bird contains "o" after position 3.

As indicated above, a generator suspends evaluation when it produces a value. The concept of suspension is important; it implies that the state of a computation is preserved (such as the position in the subject where the last substring was found) so that the computation can be resumed (such as to search for the next position).

Procedures in Icon also can suspend and be resumed. This allows programmer-defined generators in addition to the built-in ones. Suspension from a procedure (as opposed to returning, in which case the procedure call cannot be resumed) is done with

suspend *expr*

The suspend expression suspends with the values generated by *expr*. Each time it suspends, a value is delivered to the calling site. If the call of the procedure is resumed, the next value generated by *expr* is delivered, and so on. An example is:

```
procedure lc_vowel()
  suspend !"aeiou"
  fail
end
```

When the argument of suspend produces no more values, evaluation continues with the next line. The expression fail causes the procedure call to return with no value (the resumption at this point does not produce a result). Consequently, the values that lc_vowel() generates are "a", "e", "i", "o", and "u". For example, in

```
sentence ? write(find(lc_vowel()))
```

the value written (if any) is the first position, in sentence, of the first lowercase vowel, in alphabetical order.

5. MORE ON STRING SCANNING

Matching Functions

The function move(i) mentioned above is called a *matching function*, since it returns the portion of the subject that is "matched" as a result of changing the position. There is another matching function, tab(i), which moves to position i in the subject (if possible) and, like move(i), returns the substring of the subject between the previous and new positions.

The argument of tab often is provided by a string analysis function. For example,

```
sentence ? write(tab(find(lc_vowel())))
```

writes the initial substring of sentence up to the first lowercase vowel.

The argument of tab also may be given as a non-positive specification with respect to the right end of the subject. For example, tab(0) sets the position at the right end of the subject.

Locating String Positions

There are several operations in addition to find that produce positions in the subject, depending on the characters it contains.

The operation =s succeeds and matches s (moves the position past it) if s occurs at the current position in the subject. For example,

```
bird ? ="eagle"
```

succeeds if the value of bird is (or begins with) "eagle".

The function upto(s) generates the positions in the subject at which characters in s occur. For example, if the value of the subject is "condor", upto("on") generates 2, 3, and 5. Note the difference between find and upto; the former generates positions of substrings while the latter generates positions of characters in a set.

The function many(s) matches the longest possible substring containing characters in s. In particular, it fails if the character at the current position in the subject is not contained in s.

It is important to note that functions like find and upto produce positions, but the specific values of such positions generally are not of interest; they usually serve only to provide arguments to matching functions. For example, if the value of letters consists of the upper- and lowercase letters,

```
sentence ? while tab(upto(letters)) do
  write(tab(many(letters)))
```

writes the "words" in sentence. The expression tab(upto(letters)) matches up to the next letter, while tab(many(letters)) matches the word.

On occasion it is necessary to know if the current position has a specific value. The function pos(i) succeeds if the position is i but fails otherwise. For example, pos(0) succeeds if the position is at the end of the subject. Note that positions 1 and 0 identify the beginning and end of the subject. As such, the specific numbers are less important than the identification of extreme points.

Control Backtracking

Goal-directed evaluation, which may resume suspended generators, causes *control backtracking*. That is, if an expression fails and there is a suspended generator, goal-directed evaluation causes evaluation to go back the previous expression that suspended.

Expressions are evaluated from left to right, while suspended generators are resumed in a last-in, first-out fashion. Consider, for example, three expressions in conjunction:

$expr_1$ & $expr_2$ & $expr_3$

Suppose $expr_1$ and $expr_2$ are generators that succeed, but that $expr_3$ fails. When $expr_1$ suspends, it is pushed on a suspension stack. When $expr_2$ suspends, its state is pushed on the suspension stack, so the suspension stack has the form

$\leftarrow expr_1 \leftarrow expr_2$

When $expr_3$ fails, the state information for $expr_2$ is removed from the top of the suspension stack and $expr_2$ is resumed with it. If it produces another result and suspends, $expr_3$ is evaluated again. Since $expr_3$ failed previously, it can succeed now only if it depends on some external factor (such as a side effect of $expr_2$ or the time of day). Assuming it fails again, $expr_2$, which is again on the top of the suspension stack, is resumed again. This continues until $expr_2$ has no more results. When it fails to produce a result, $expr_1$, which now is on the top of the suspension stack, is resumed, and so on. The effect is cross-product evaluation with a depth-first 'search' for alternatives.

Control structures, by their nature, interfere with control backtracking. They provide the mechanisms necessary to prevent an entire Icon program from being mutually evaluated. They do this by clearing the suspension stack of suspended generator state information. For example, in

if find(s) then $expr_2$ else $expr_3$

If find(s) succeeds, it suspends, but its suspended generator is discarded and $expr_2$ is evaluated. In fact, if the suspended generator for find(s) were not discarded, failure of $expr_2$ would cause find(s) to be resumed. The effect would not correspond to the expected semantics of if-then-else. Similarly, in

while $expr_1$ do $expr_2$

If $expr_1$ or $expr_2$ suspend, their suspended state information is discarded. See [9] for a more detailed description of the issues involved.

Data Backtracking

Matching functions also perform data backtracking. When a matching function produces a result, it suspends, even though it cannot produce another result (there is only one way to set the position to a specific value). If a subsequent expression fails, the suspended matching function is resumed, at which point it restores the position to the value it had prior to the initial evaluation of the matching function. Thus, a matching function reverses the change it made to the position if that change did not lead to subsequent success in the expression of which it is a part.

Data backtracking of the position in the subject assures that alternative matches start in the same place. For example, in

```
line ? (tab(10) & find("or")) | move(1)
```

if tab(10) is successful but find("or") fails, the resumption of tab(10) restores the position to its previous value (at the beginning of the subject) and move(1) starts at the same position as tab(10) did.

The Result of Scanning

The result of the scanning expression

$expr_1$? $expr_2$

is the result of the analysis expression $expr_2$. For example, the result of

```
text ? (tab(upto(letters)) & tab(many(letters)))
```

is the result of tab(many(letters)), provided that tab(upto(letters)) succeeds; otherwise the scanning expression fails.

If the analysis expression is a generator, the scanning expression in which it appears generates its results. For example,

```
line ? tab(1 to 10)
```

generates the initial substrings of line of length 0 through 9 (fewer if line is not at least 9 characters long).

Synthesizing Strings

As mentioned earlier, $expr_2$ in

$expr_1$? $expr_2$

is referred to as the *analysis expression* only as a matter of convenience. In fact, $expr_2$ may contain any kind of computation. In particular, it often performs string synthesis as well as analysis. For example, the following procedure deletes everything but letters from a string

```
procedure only_letters(s)
  result := ""              # start with empty string
  s ? while tab(upto(letters)) do
    result := result || tab(many(letters))
  return result
end
```

Scanning Keywords

Although the scanning environment is implicit and it usually is not necessary to make explicit reference to the subject and position, these values are available through the keywords &subject and &pos. For example,

```
&pos := 1
```

sets the scanning position to 1. This assignment is equivalent to tab(1).

It also is possible to change the subject, as in

```
&subject := read()
```

which assigns the next line to the subject. Whenever the subject is changed, the position is set to 1.

It usually is undesirable, as a matter of programming practice, to refer to the subject and position explicitly. However, the ability to reference the subject and position explicitly is necessary when writing matching procedures to augment the built-in repertoire of matching functions. Consider, for example, a procedure arb, which matches zero or more characters:

```
procedure arb()
  local pos
  start := &pos
  suspend &subject[
    start : &pos := start to *&subject + 1 by 1
    ]
  &pos := start
  fail
end
```

The argument of suspend produces a substring starting at the original value of &pos. The end of this substring (&pos), which starts at the original value of &pos (producing the empty string first), increases each time arb is resumed. If arb is resumed repeatedly, the matched substring increases until the end of the subject is reached, in which case the substring operation fails and evaluation continues after the suspend expression. At this point &pos is restored to its original value (performing data backtracking in the manner of matching functions) and arb fails.

Thus, arb() matches zero or more characters in order of length. For example,

```
sentence ? (arb() & ="cats" & arb() ="dogs")
```

succeeds if sentence contains the substring "cats" followed by the substring "dogs".

Procedures like this follow a simple model: saving &pos, generating substrings of &subject obtained by changing &pos, restoring &pos when there are no more results, and finally failing. For example, arb can be given a parameter that causes the matched substring to be incremented by a specified amount:

```
procedure arbn(i)
  local pos
  start := &pos
  suspend &subject[
    start : &pos := start to *&subject + 1 by i
    ]
  &pos := start
  fail
end
```

For example, arbn(3) matches strings whose length is an even multiple of 3.

Similarly, reversing the order in which positions are generated provides a procedure that matches the longest possible string first and works backward to shorter ones:

```
procedure maxarb()
  local pos
  start := &pos
  suspend &subject[
    start : &pos := *&subject + 1 to start by -1
    ]
  &pos := start             # redundant here
  fail
end
```

6. MATCHING EXPRESSIONS

As described above, the term 'matching' refers to an expression that returns the portion of the subject between the positions before and after evaluation. The examples of matching shown so far also perform data backtracking on the position so that alternative matching expressions start at the same position: if they fail they leave the position unchanged.

The two aspects of evaluation – (1) producing a portion of the subject between successive positions and (2) performing data backtracking on the position – are logically separable. The former is useful for assuring that results come from a 'matched' portion of the subject. The latter is important in assuring that alternative matches are performed independently.

Expressions that perform data backtracking are referred to as *state-maintaining*. In terms of programming methodology the independence of the alternative matches is of primary importance. It effectively assures that matching is free of side effects with respect to the matching process.

Because of control backtracking in goal-directed evaluation, state-maintaining expressions can be combined in operations without interfering with the independence of alternative matches. For example, if $expr_1$ and $expr_2$ are state-maintaining expressions, then

$expr_1 \mid expr_2$

and

$expr_1 \parallel expr_2$

are state-maintaining.

The criterion for writing state-maintaining expressions is simple: An expression is state-maintaining if it does not contain control structures that interfere with control backtracking. For example,

```
while tab(upto(letters)) do
  tab(many(letters))
```

is not state-maintaining, since suspended generators are discarded as described in Section 5. This, of course, is exactly what is wanted in this situation.

It is worth noting that the criteria for composing state-maintaining *matching* expressions are more stringent than those for composing arbitrary state-maintaining expressions. For example, if $expr_1$ and $expr_2$ are state-maintaining matching expressions,

$expr_1$ & $expr_2$

is state-maintaining. It is not, in general, a matching expression: its result is the result of $expr_2$, which is not necessarily the entire portion of the subject matched by both $expr_1$ and $expr_2$.

7. PATTERN MATCHING

There are two ways of viewing a matching function such as move(i): (1) as a function that moves the position to i and returns the matched substring, or (2) as a 'pattern' that matches strings that are i characters long.

The first view is concerned by the *process* by which scanning is performed. The second view is concerned with the characterization of a set of strings (in technical terms, a language).

In writing analysis expressions, the first view usually is taken. That is, Icon programmers tend to focus on the process by which the position is changed. The second view, however, provides a higher level of abstraction; it is more declarative, specifying 'what' rather than 'how'. The declarative view is analogous to the concept of 'pattern' in SNOBOL4 [10] and is an essential component of languages like Prolog [11, 12].

Although the built-in matching and string analysis functions of Icon are designed to specify how matching is done, the higher-level, declarative 'pattern' view can be used to advantage in programming. For example, arbn(i) can be thought of in a natural way as a pattern that matches all strings whose lengths are even multiples of n. Note that although the process by which these strings are produced is not important, the *order* in which matches are attempted is important. The order resolves the ambiguities that are inherent in pattern matching. In this sense, patterns are part way between declarative and imperative.

Since a pattern conceptually characterizes a set of strings (usually infinite), it is analogous to a grammar that provides a finite and structured characterization of a set of strings that comprise a language. Consider a simple context-free phrase-structure grammar that describes the skeletons of a kind of nested parenthesized lists:

$$\mathcal{P} \rightarrow [\,] \mid [\mathcal{L}] \mid [\mathcal{P}]$$
$$\mathcal{L} \rightarrow , \mid , \mathcal{P} \mid , \mathcal{L}$$

$\mathcal{P}$ and $\mathcal{L}$ are nonterminal symbols, vertical bars separate alternative productions, and all other right-hand-side symbols are terminal. Examples of strings in the language for $\mathcal{P}$ are [,[]], [,,,], and [[]].

Such a grammar can be viewed in several ways. From one view, it is a specification for a recognizer for the corresponding language. From another view, it is prescription for generating strings in the language.

There is an isomorphism between this grammar and Icon matching procedures that illuminates some of the concepts of pattern matching mentioned above. In this isomorphism, nonterminal symbols of the left side correspond to matching procedures that recognize strings in the language. These procedures are obtained using the following correspondences between right-hand-side symbols in the grammar and Icon matching expressions:

- terminal symbols correspond to literal matching expressions
- nonterminal symbols correspond to calls of matching procedures
- sequences of symbols correspond to the concatenation of matching expressions
- alternation corresponds to alternation

A corresponding matching procedure consists of a suspend expression whose argument is given by the correspondences above.

For the grammar given above, the matching procedures are:

```
procedure P()
  suspend (
    ="[]" |
    (="[" || L() || ="]") |
    (="[" || P() || ="]")
    )
  fail
end

procedure L()
  suspend (
    ="," |
    (="," || P()) |
    (="," || L())
  fail
end
```

Thus, P() is a 'pattern' that matches strings in the language for $\mathcal{P}$. For example.

```
string ? (P() & pos (0))
```

succeeds if string is in the language for $\mathcal{P}$ but fails otherwise. (The expression pos(0) assures that P() matches all of string; otherwise it would match any initial substring from $\mathcal{P}$.) This form of matching only works, of course, if the grammar is free of left recusion.

This model of pattern matching illustrates the nature of goal-directed evaluation mentioned earlier: recursive-descent, depth-first search with backtracking. This method is not efficient for this kind of problem; its value lies in its illustration of the close correspondence between grammars and patterns. Such a 'recognizer' also is of limited interest. However, the model is easily extended to parsers and translation. It also is possible to express context-sensitivity by providing arguments to matching procedures. See [7] for a discussion of these possibilities.

8. THE MAINTENANCE OF SCANNING ENVIRONMENTS

Scanning expressions are on a par with all other expressions in Icon. Consequently, scanning expressions can occur in conjunction, as in

$$(expr_1 \,\&\, expr_2) \;?\; (expr_3 \;?\; expr_4)$$

Scanning expressions also can be nested, as in

$$(expr_1 \;?\; (expr_2 \;?\; expr_3))$$

Procedures called in analysis expressions also can contain scanning expressions, as in

```
expr1 ? p()
```

where **p** itself contains scanning expressions. This situation amounts to dynamic nesting, and it occurs more frequently in practice than the static form of nesting shown above.

In order for such constructions to behave in a reasonable way, Icon maintains multiple scanning environments. Scanning environments (and **&subject** and **&pos**) are global with respect to procedure calls, but they are local to scanning expressions.

Icon begins execution with the scanning environment: {" ", 1} (an empty, zero-length subject). When a scanning expression is evaluated, it saves the current scanning expression and creates a new one. If the scanning expression fails, it restores the previously saved scanning environment. If the scanning expression suspends, its scanning environment is saved (since it may be resumed) and the previous scanning environment is restored. A scanning environment remains in existence until its corresponding analysis expression fails or until it is no longer possible to resume it. This occurs as the result of control structures that discard suspended generators.

Because of the possibility of scanning expressions in conjunction as well as nested scanning expressions, the structure connecting saved scanning environments is best thought of as a tree, not a stack. (The same is true of suspended generators.)

In general, the tree of scanning environments is rooted in the scanning environment associated with the initiation of program execution as described above. There are two ways that the tree of scanning environments can grow. One is horizontally, as in expressions such as

$$(expr_1 \;?\; expr_2) \,\&\, (expr_3 \;?\; expr_4) \,\&\, \ldots$$

The other is vertically, as in expressions such as

$$(expr_1 \;?\; (expr_2 \;?\; (expr_3 \;?\; (expr_4 \ldots))))$$

In horizontal growth of the scanning environment tree, an analysis expression is suspended during the evaluation of a subsequent expression. In vertical growth, before an analysis expression completes evaluation, a scanning expression that is nested within it is evaluated. Vertical growth usually appears in programs in the form of matching procedures that themselves contain scanning expressions, as mentioned above.

As an example, consider the evaluation of the following expression:

```
("abc" ? move(2 | 1)) & ("defg" ? (tab(4) ?
move(1 | 2)))
```

Assuming that there is no other surrounding expression, the evaluation of

```
"abc" ? move(2 | 1)
```

causes the scanning environment tree to become

$$\begin{tikzcd}
\text{\{" ",1\}} \arrow[d] \\
\text{\{"abc",3\}}
\end{tikzcd}$$

When

```
"abc" ? tab(2 | 1)
```

suspends,

```
"defg" ? (tab(4) ? move(1 | 2))
```

is evaluated. The tree of scanning environments grows horizontally. After the evaluation of **tab(4)**, the tree is:

$$\begin{tikzcd}
\text{\{" ",1\}} \arrow[d] & \\
\text{\{"abc",3\}} & \text{\{"defg",4\}}
\end{tikzcd}$$

Evaluation of the nested scanning expression then causes the tree of scanning environments to grow vertically:

$$\begin{tikzcd}
\text{\{" ",1\}} \arrow[d] & \\
\text{\{"abc",3\}} & \text{\{"defg",4\}} \arrow[d] \\
 & \text{\{"def",2\}}
\end{tikzcd}$$

If the expression above appears in a context that causes it to be resumed, as in

```
(("abc" ? move(2 | 1)) & ("defg" ? (tab(4) ?
move(1 | 2)))) & expr
```

where *expr* fails, then the expression **move(1 | 2)** is resumed and the last scanning environment is changed:

$$\begin{tikzcd}
\text{\{" ",1\}} \arrow[d] & \\
\text{\{"abc",3\}} & \text{\{"defg",4\}} \arrow[d] \\
 & \text{\{"def",3\}}
\end{tikzcd}$$

Further resumption produces no new result for this expression, resumption of **tab(4)** produces no new result, the second scanning expression in the mutual evaluation produces no new result, and **move(2 | 1)** in the first scanning expression in the mutual is resumed. At this point, the scanning environment tree again has the form

$$\begin{tikzcd}
\text{\{" ",1\}} \arrow[d] \\
\text{\{"abc",3\}}
\end{tikzcd}$$

Note that two scanning environments were discarded as the result of the failure of scanning expressions.

The second result for **move(2 | 1)** changes this environment to

$$\begin{tikzcd}
\text{\{" ",1\}} \arrow[d] \\
\text{\{"abc",2\}}
\end{tikzcd}$$

At this point, the second scanning expression in the mutual evaluation is evaluated again, and the tree of scanning environments grows again in a fashion similar to that illustrated above. The tree of scanning environments reverts to a single root node only when all alternatives in the mutual evaluation have been produced.

Note that all the nodes along the right edge of the tree of scanning environments correspond to expressions whose evaluation is incomplete and are 'active', while all other nodes correspond to inactive expressions that may produce another result if they are resumed because of failure of expressions corresponding to nodes to their right.

9. A MODEL OF SCANNING EXPRESSIONS

The internal mechanism for maintaining scanning environments can be better understood by modelling the scanning expression in terms of Icon procedures.

It is not possible to model

$expr_1$? $expr_2$

with a single procedure, such as Scan($expr_1$, $expr_2$), since in a procedure call, all arguments are evaluated before the procedure is called. In a string scanning expression, however, scanning environments must be manipulated between the time of $expr_1$ is evaluated and $expr_2$ is evaluated. The interposition of this manipulation can be modelled using two procedures, in which the scanning expression is mapped as follows:

$expr_1$? $expr_2 \rightarrow$ Escan(Bscan($expr_1$), $expr_2$)

The order of evaluation is easier to see if the nested procedure call is recast in suffix form, so that expressions are written from left-to-right in the order they are evaluated:

(($expr_1$)Bscan, $expr_2$)Escan

Thus, $expr_1$ is evaluated first, providing the subject and the argument to Bscan, which manipulates scanning environments. Next the analysis function $expr_2$ is evaluated. The value returned by Bscan (which contains information about scanning environments) and the result of the analysis expression are arguments to Escan, which also manipulates scanning environments.

In this model, an Icon record type is used to represent scanning environments:

```
record Envir(subject,pos)
```

In Icon notation, if E is an Envir record, E.subject is the subject of scanning and E.pos is the position in the subject. The procedures are:

```
procedure Bscan(e1)
  local OuterEnvir
  OuterEnvir := Envir(&subject,&pos)
  &subject := e1
  &pos := 1
  suspend OuterEnvir
  &subject := OuterEnvir.subject
  &pos := OuterEnvir.pos
  fail
end

procedure Escan(OuterEnvir,e2)
  local InnerEnvir
  InnerEnvir := Envir(&subject,&pos)
  &subject := OuterEnvir.subject
  &pos := OuterEnvir.pos
  suspend e2
  OuterEnvir.subject := &subject
  OuterEnvir.pos := &pos
  &subject := InnerEnvir.subject
  &pos := InnerEnvir.pos
  fail
end
```

Bscan saves the current subject and position in OuterEnvir and sets &subject to the value provided by its argument. It suspends, returning OuterEnvir, which is needed by Escan. By suspending, Bscan assures that it will be resumed in the case of subsequent failure – in which case it restores the subject and position from OuterEnvir.

Escan is called with the record provided by Bscan and the result of the analysis expression. Escan creates another record to save the current subject and position, restores the subject and position from OuterEnvir, and then suspends with the result of the analysis expression (which is the result of the entire scanning expression). If Escan is resumed, it restores the subject and position to the values they had when the analysis expression completed. It also updates the outer scanning environment in case a subsequent expression, operating in the context of the outer scanning environment, modified that subject or position. Escan then fails, causing the analysis expression to be resumed (if it suspended). If the analysis expression did not suspend, or if it now fails, the suspended Bscan is resumed. This corresponds to failure of the entire scanning expression, so the previous scanning environment is restored before Bscan fails.

Note that $expr_1$? $expr_2$ is, itself, state-maintaining. It performs data backtracking on both the subject and the position.

These procedures show how scanning environment are maintained internally in Icon. They do not account for all possibilities. For example, a suspend expression may occur in a scanning expression that is inside a procedure. In this case the current scanning environment is saved and the previous one is restored. If the procedure call is subsequently resumed, the process is reversed. Icon also has control structures that cause scanning expressions to be terminated in the middle of analysis. See [13, 14] for more detailed descriptions.

10. CONCLUSIONS

String Scanning and Pattern Matching

The relationship between string scanning and pattern matching is an important one. As mentioned above, Icon focusses on the process of matching – string scanning is inherently imperative in nature. By contrast, SNOBOL4 patterns are inherently declarative in nature. Icon, however, allows declarative characterizations. The procedure arb( ) in Section 5 is a direct analog of a SNOBOL4 pattern ARB. Both do the same thing. However, SNOBOL4 patterns are not purely declarative. They embody information about the order of matching as described in Section 7. This information is largely hidden in SNOBOL4. SNOBOL4 has built-in matching functions, but there is no way to access them directly.

Declarative characterizations are limiting in what they can describe in the absence of imperative mechanisms. SNOBOL4, for example, provides no mechanisms for providing programmer-defined matching procedures. A SNOBOL4 programmer must get by with the built-in matching functions as they are embodied in patterns. In addition, SNOBOL4, with its declarative characterization of pattern matching, has no direct, natural way to perform other kinds of computation during pattern matching. In string scanning, on the other hand, all types of computation can be performed as needed during the matching process.

Although Prolog [11] is not designed primarily for string manipulation, its evaluation mechanism resembles Icon's in many ways. In particular, goal-directed evaluation and generators have similar manifestations in Icon and Prolog. There are differences between the two languages that are very significant, however. While Prolog is declarative in nature, Icon is imperative.The order of matching in Prolog is largely hidden from the Prolog programmer and there is little in the way of mechanisms to control it ('cut' being the notable exception). In Icon, on the other hand, the order of matching is strictly defined and there are many (conventional and unconventional) control structures. Both approaches have their merits.

The Concepts in String Scanning

The success of string scanning in Icon lies in the concept of a scanning environment in which a subject string being analysed is implicit, as is the position in the subject at which attention is focussed. Matching functions provide ways of changing the position and also of producing portions of the subject that are of interest. While specific positions produced by string analysis functions are needed to drive the matching process as arguments of matching function, these positions usually are not accessed explicitly.

Generators and control backtracking provide the mechanism for searching for alternatives in combinations of matches. Data backtracking assures the independence of alternative matches. The maintenance of scanning environments in a state-maintaining fashion allows multiple scanning expressions to be used in combination.

Other Possibilities

Taken together, all of these features provide a simple and powerful mechanism for analyzing strings. These concepts, however, are not restricted to string analysis.

For example, a string synthesis facility can be designed along lines that are similar to those of string scanning. Just as a subject string is implicit in string analysis, an object string could be implicit in string synthesis. In this case, the scanning environment might have the form {subject,sposition,object,oposition}. Here sposition is the position in the subject as before, while oposition provides an insertion position in the object string being synthesized. In such a facility, synthesis functions are analogous to matching functions. Experience with string analysis suggests that data backtracking for string synthesis (object and oposition) should be concomitant with data backtracking for analysis. The model of string scanning given in Section 9 can be easily adapted to this view of string synthesis. There are many other possibilities, including *transformation* of a subject string by analysis and synthesis functions working in concert. Such possibilities probably are best explored by modelling them in terms of Icon procedures as shown in Section 9.

Most of the fundamental aspects of scanning in Icon are not tied to strings, *per se*. The concepts of scanning environments, generators, goal-directed evaluation, control backtracking, and data backtracking could equally well be applied to the scanning of structures such as lists and trees. This possibility already has been explored with limited success [15]. The inherent problem with structure scanning is that structures are fundamentally more complex than strings. Strings are simple – they are just sequences of characters with no internal structure. The portion of a string between two positions is itself a string, so that the concept of matching is well-defined. In a structure, however, the portion between two positions (such as the root and a leaf) is not so easily interpreted as a substructure of the same kind. For structures that contain loops, such as general directed graphs, the situation is considerably more complicated. Nonetheless, the value of the concepts from string scanning suggest that it is worth exploring a facility for structure scanning based on the same underlying ideas.

Acknowledgements

The origins of string scanning in Icon lie in SNOBOL4 [10]. They are molded by the SL5 programming language [16, 17]. Many persons have participated in the design and implementation of string scanning in Icon. Dave Hanson, Tim Korb, Cary Coutant, and Steve Wampler were major contributors. Ken Walker made significant contributions to the method for maintaining scanning environments and provided the material on the tree of scanning environments and the procedures for modelling string scanning.

The work described in the paper was supported by National Science Foundation Grants MCS-8101916, DCR-8401831, and DCR-8502015.

REFERENCES

1. V. H. Yngve, "A Programming Language for Mechanical Translation", *Mechanical Translation* 5, **1**, 25–41 (1958).
2. C. Y. Lee and others, *A Language for Symbolic Communication*, MM 62-3344-4, Bell Telephone Laboratories, 1962.
3. D. J. Farber, R. E. Griswold and I. P. Polonsky, "SNOBOL, A String Manipulation Language", *J. ACM 11*, **1**, 21–30 (Jan. 1964).
4. J. F. Gimpel, "A Theory of Discrete Patterns and Their Implementation in SNOBOL4", *Comm. ACM 16*, **2**, 91–100 (Feb. 1973).
5. P. Klint, An Overview of the SUMMER Programming Language, in *Conference Record of the Seventh Annual ACM Symp. on Prin. of Programming Languages*, 1980.
6. *Proceedings of the Workshop on Pattern-Directed Inference Systems, SIGART Newsletter*, June 1977.
7. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
8. R. E. Griswold and D. R. Hanson, "An Alternative to

the Use of Patterns in String Processing", *ACM Trans. Prog. Lang. and Systems 2*, **2**, 153–172 (1980).
9. J. O'Bagy, *The Implementation of Generators and Goal-Directed Evaluation in Icon*, Doctoral Dissertation, The University of Arizona, 1988.
10. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1971.
11. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
12. L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.
13. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton Univesity Press, 1986.
14. R. E. Griswold, *Supplementary Information for the Implementation of Version 7 of Icon*, The Univ. of Arizona Icon Project Document IPD51, 1988.
15. A. J. Anderson and R. E. Griswold, *Unifying List and String Processing in Icon*, The Univ. of Arizona Tech. Rep. 83–4, 1983.
16. R. E. Griswold and D. R. Hanson, "The SL5 Procedure Mechanism", *Comm. ACM 21*, **5**, 392–400 (May 1978).
17. R. E. Griswold, "String Analysis and Synthesis in SL5", *Proceedings of the ACM Annual Conference*, 1976, 410–414.

Announcements

3–7 September 1990

MONTREUX, SWITZERLAND

Eurographics' 90

Images

Synthesis, Analysis and Interaction

The EUROGRAPHICS Association is 10 years old in September 1990. For the last 10 years, EUROGRAPHICS has served the European research community in computer graphics and its applications, through the annual event, journal, workshop programme and other activities.

In the past, EUROGRAPHICS conferences have concentrated in the main on topics traditionally associated with computer graphics and human computer interaction. EUROGRAPHICS '90 will continue to address such topics.

For EUROGRAPHICS '90, a new theme of the conference will be the relationship between image synthesis (traditionally the domain of computer graphics) and image processing and computer vision.

It is now clear that there is overlap between image synthesis and image analysis in both techniques and applications. For example, as computer graphics is used more and more in the visualization of scientific and engineering computations, so it looks likely that image processing techniques will be used to help develop an interpretation of the experimental results.

Tutorials, state of the art reports and invited papers will address the relationship between graphics and image analysis, at both introductory and advanced levels.

Tutorials

3–4 September 1990

The first two days of the event will be devoted to the tutorial programme. Tutorials will be given by leading international experts and will cover a wide range of topics offering an excellent opportunity for professional development in computer graphics, image processing and related areas. The programme includes both introductory and advanced tutorials.

Each tutorial will occupy one full day. Lecture notes will be provided for attendees.

Preliminary List of Topics

● Introduction to Image Processing
● X and NeWs Environments
● Introduction to Ray Tracing and Radiosity
● Image Reconstruction
● Superworkstations for Graphics
● Human Visual Perception
● Intelligent CAD Systems
● Free-Form Surfaces and CSG
● Graphics and Distributed Environments
● Scientific Data Visualization
● Computer Vision
● Traditional Animation: A Fresh Look
● Computer Graphics for Software Engineering

State of the Art Reports

5–7 September 1990

In parallel with the conference paper session, a series of 1½ hour reports on topics of wide current interest in key fields will be given by leading experts in the fields. These will serve to keep attendees abreast of the state of the art in these fields and will highlight recent significant advances.

Preliminary List of Topics

● Standardization in Graphics and Image Processing:
Present and Future
● Advanced Rendering
● Object Oriented Design in Action
● Digital Typography
● Simulation of Natural Phenomena
● Advanced Mathematics and Computer Graphics
● Interactive Graphics and Video Discs
● Graphics-Education.

Conference

5–7 September 1990

Papers selected by the International Programme Committee will present the most relevant and recent developments in Computer Graphics. The Conference Proceedings will be published by North-Holland.

List of Topics

● Graphics Hardware
● Superworkstations
● Hypersystems
● Graphics and Parallelism
● Distributed Graphics
● Visualization Techniques
● Animation and Simulation
● Image Processing
● Sampling Theory
● Unwarping
● Image Filtering
● Image Representation
● Computational Geometry
● Modelling
● Standards
● Exchange of Product Data
● Graphics for CAD, CAM, CAE
● Human-Computer Interaction
● Human Factors
● Tool kits for UIMS and WMs
● Presentation Graphics
● Graphics in the Office
● Graphics in Publication and Documentation
● Page Description Languages
● Novel Graphics Applications
● Graphics as an Experimental Tool
● Graphics in Education
● Integration of Graphics and Data Bases
● Colour
● Multi Media Graphics

Video and Film Competition

There will be a competition of computer-generated videos and films, with prizes awarded for the best entries based on creativity and technical excellence. Submissions are invited for scientific and technical applications, art and real-time generated sequences. Entries will be shown during the conference.

Slide Competition

A competition will also be held for artistic images and scientific and technical images submitted on 35mm slides. Prizes will be awarded for the best entries, and slides will be shown during the conference.

The closing date for submission to both competitions will be June 15, 1990. Entries should be sent to the Conference Secretariat. Rules for the competition will be sent to people who apply to the Conference Secretariat.

Montreux

Pearl of the Swiss Riviera

Montreux, Pearl of the Swiss Riviera, has been chosen as the venue for Eurographics '90. The Congress will be held in the Maison des Congrès, a fully equipped convention centre built on the shores of Lake Geneva.

Sheltered by mountains, nestled in hills, surrounded by forests and famous vineyards, Montreux can seem far away from the world – but it's an hour by car or train from Geneva (direct train connection between Geneva airport and Montreux).

For relaxing or working, Montreux is ideal, with a mild climate all year round. Sports, nightlife and history are here; a casino, a jazz festival and the famous medieval Chillon Castle. It has a 9,6 km lake side promenade where palm trees and tropical flowers will amaze visitors. Montreux ranks amongst one of the most sunny areas of Switzerland, and is protected from the cold winds by the Rochers de Naye (2,042 m.) stretching up behind Montreux from where there is a breathtaking view of the whole Lemanic and Alpine region.

For further information please contact:

Eurographics '90, Conference Secretariat, Paleo Arts et Spectacles, Case postale 177, CH-1260 Nyon, Switzerland. Tel. (41) 22 62 13 33. Telex 419 834. Telefax (41) 22 62 13 34.