A Pattern-Matching Laboratory; Part I — An Animated Display of String Pattern Matching*

Kenneth Walker
Ralph E. Griswold

TR 86-1

January 2, 1986

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

A Pattern-Matching Laboratory; Part I — An Animated Display of String Pattern Matching

1. Introduction

As discussed in [1], string pattern matching in the style of SNOBOL4 and Icon is easy to understand in a general, intuitive way. This accounts for the ease with which it can be learned and used. However, the process by which pattern matching takes place is generally poorly understood. Consequently, the implementation of pattern matching traditionally has been *ad hoc*; generalizations and extensions to it have been limited by the lack of a general and coherent implementation model.

There have been numerous approaches to describing pattern matching, including "bead diagrams" [2], cursor-position transformations [3], formal algebraic models [4], denotational semantics [5, 6], axiomatic semantics [7], as well as implementation models [8-11]. These approaches have been useful in explicating pattern matching, but none of them has been entirely successful in providing the programmer or implementor with a clear understanding of the pattern-matching process.

These considerations have motivated the development of a pattern-matching "laboratory" that is designed to serve as a programming environment for the design and analysis of pattern-matching facilities. This report describes the first step in the development of the laboratory — a general model for pattern matching and a tool that provides an animated display of the string pattern matching.

This tool is based on Icon's string scanning facility, but includes as well most of SNOBOL4's pattern-matching repertoire. The reader should be familiar with Icon [12]. Some knowledge of SNOBOL4 [2] is helpful also.

2. A Model of String Scanning

The Icon string-scanning expression has the form

$$expr_1 \ ? \ expr_2$$

where $expr_1$ is the *subject expression* that provides a subject that is the focus of attention for the evaluation of the *matching expression* $expr_2$. During scanning, the keyword &subject contains the value being scanned and &pos is the position in &subject where matching expressions apply.

The string-scanning expression is a control structure. This is in contrast to operations and function calls, in which all arguments are evaluated before the operation or function is performed, since between the time that $expr_1$ is evaluated and $expr_2$ is evaluated, the two "state variables" &subject and &pos are changed. The value of $expr_1$ provides the value of &subject and &pos is set to one, indicating that scanning begins at the beginning of &subject. After these variables are set, $expr_2$ is evaluated; it typically examines the value of &subject and changes the value of &pos in the process. &subject and &pos are global variables; their values constitute an environment for scanning. The outcome of the scanning expression is the outcome of $expr_2$.

Even though $expr_1$? $expr_2$ is a control structure, it can be modeled using procedures. A naive model is

$$expr_1 \ ? \ expr_2 \rightarrow \text{Bscan}(expr_1) \ \& \ expr_2$$

The procedure Bscan intervenes in the evaluation process and sets the values of &subject and &pos before $expr_2$ is evaluated. In this naive model, the operation of Bscan corresponds to the following procedure:

```
procedure Bscan(e1)
   &subject := e1
   &pos := 1
   return
end
```

This model illustrates how simple the string-scanning expression is — any actual pattern matching takes place during the evaluation of $expr_2$ and the only function of the string-scanning expression itself is to set the values of the global variables in the scanning environment.

This naive model, which is equivalent to the mutual evaluation

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

does not account for the fact that string-scanning expressions can be nested or that several can occur in mutual evaluation so that several scanning environments can exist simultaneously at any point in program execution. In order for string scanning to behave in a useful and coherent way, the values of &subject and &pos are saved prior to assigning new values to them in a string-scanning expression, and they are restored when a string-scanning expression is complete. Furthermore, a string-scanning expression can produce a result but be resumed to produce another, so it is necessary to reset &subject and &pos if a scanning expression is resumed. To accomplish this, a more general model is needed:

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

The operation of these functions is illustrated by the following procedures in which records of type ScanEnvir hold the values of &subject and &pos for scanning environments:

```
record ScanEnvir(subject, pos)

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

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

In this formulation, $expr_1$ is evaluated first and provides the argument to Bscan that is used for the new value of &subject. In Bscan, the current values of &subject and &pos are saved in OuterEnvir before the new ones are set. Bscan then suspends with OuterEnvir, which is passed on to Escan. However, $expr_2$ is evaluated first and may change the values of the state variables before Escan is called. If evaluation of $expr_2$ succeeds, Escan is called with two arguments: the outer scanning environment that was in effect before Bscan was called, and the result produced by the evaluation of $expr_2$.

Escan saves in InnerEnvir the scanning environment as it was left by the evaluation of $expr_2$, restores the outer environment, and suspends with the result produced by the evaluation of $expr_2$.

If the scanning expression occurs in a context in which it is resumed, the evaluation of Escan picks up after the suspend expression, and &subject and &pos are restored to the values they had when $expr_2$ produced its previous result. If $expr_2$ produces another result, as in

```
expr1 ? (&pos := 1 | 2)
```

Escan is *called* again; the situation is the same as it was when $expr_2$ produced its previous result.

If $expr_2$ does not produce another result, Bscan is resumed and picks up evaluation after its suspend expression. It restores the outer scanning environment and fails. At this point, $expr_1$ is resumed. If it produces another result, as in

```
(s1 | s2) ? expr2
```

Bscan is called again and the process described above is repeated.

3. Matching Functions

As illustrated above, the evaluation of a matching expression may change the values of the state variables in the current scanning environment. While this may be done by explicit assignment to &subject and &pos, changes are usually confined to &pos and normally occur as a side effects of evaluating the matching functions: tab(i), which sets &pos to i, and move(i), which adds i to &pos.

These matching functions obey a protocol that involves data backtracking of &pos. In this protocol, which effectively defines "matching", a function changes the value of &pos only if it is successful and, if resumed, restores &pos to the value it had when the function was called. Thus, a matching function has no net effect on the scanning environment unless it "matches".

This protocol is easily cast in a form that allows matching procedures to be formulated:

```
procedure p( ... )
   suspend &pos <- new position
   fail
end
```

If a call of such a procedure produces a result and is resumed, the reversible assignment operation automatically restores the previous value of &pos.

In addition to the data backtracking protocol, matching expressions in Icon also conform to the convention of returning the substring of &subject between the values of &pos before and after they match, thus returning the "matched" substring.

This convention can be added to the backtracking formulation above as follows:

```
procedure p( ... )
   suspend &subject[.&pos:&pos <- new position ]
   fail
end
```

The first instance of &pos is explicitly dereferenced so that its value is obtained before a new value is assigned to it.

Using this model, the functions tab(i) and move(i) can be written as procedures as follows:

```
procedure tab(i)
   suspend &subject[.&pos:&pos <- i]
   fail
end
```

and

```
procedure move(i)
   suspend &subject[.&pos:&pos <- &pos + i]
   fail
end
```

4. Trees of Scanning Environments

Prior to the execution of an Icon program, the values of &subject and &pos are established as if the following expressions were evaluated:

```
&subject := ""
&pos := 1
```

This situation may be viewed as starting program execution with an expression of the form

```
"" ? main()
```

Thus, there is always at least one scanning environment in existence during the course of the execution of an Icon program. Additional scanning environments come into existence as scanning expressions are evaluated. A scanning environment remains in existence until its matching expression fails or until it is no longer possible to resume it. It may become impossible to resume an expression because control backtracking is prevented, which occurs at well-defined places. Examples are:

- After the evaluation of the control expression in a control structure, such as for $expr_1$ in

  if $expr_1$ then $expr_2$

- After the evaluation passes from one expression in a compound expression to another, as for $expr_1$ in

  { $expr_1$; $expr_2$ }

- On return from a procedure call, as for $expr_1$ in

  return $expr_1$

Expressions that occur in such contexts are called *bounded expressions*. In particular, a bounded expression can produce at most one result.

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

($expr_1$? $expr_2$, $expr_3$? $expr_4$, ...)

and vertically, as in expressions such as

($expr_1$? ($expr_2$? ($expr_3$? ($expr_4$ ...))))

In horizontal growth of the scanning environment tree, a scanning expression completes and becomes dormant during the evaluation of another expression in the current bounded expression. In vertical growth, before a scanning expression completes, another 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 in

$expr_1$? p()

where p as the form

```
procedure p()
        :
expr2 ? expr3
        :
end
```

Let a scanning environment be represented formally as a pair

```
<&subject, &pos>
```

Then the tree of scanning environments is rooted in

```
<"", 1>
```

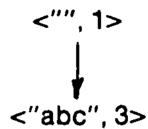
As an example, consider the evaluation of the following expression:

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

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

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

the scanning environment tree becomes



When

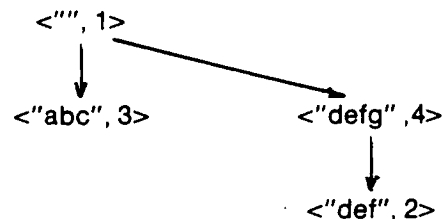
```
"abc" ? tab(2 | 1)
```

suspends, its scanning environment becomes dormant, and

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

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

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



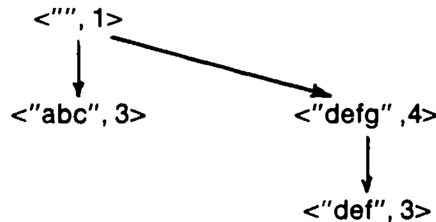
If the expression

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

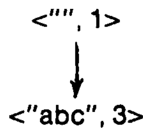
constitutes a complete bounded expression, its evaluation is complete at this point, and the tree of scanning environments reverts to a single root node. However, if the expression above appears in a context that causes it to be resumed, as in

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

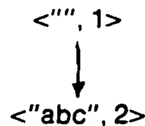
then the expression move(1 | 2) is resumed and the last scanning environment is changed:



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 evaluation is resumed. At this point, the scanning environment tree again has the form



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



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 dormant expressions that may produce another result if they are resumed because of failure of expressions corresponding to nodes to their right.

5. The Animated Display

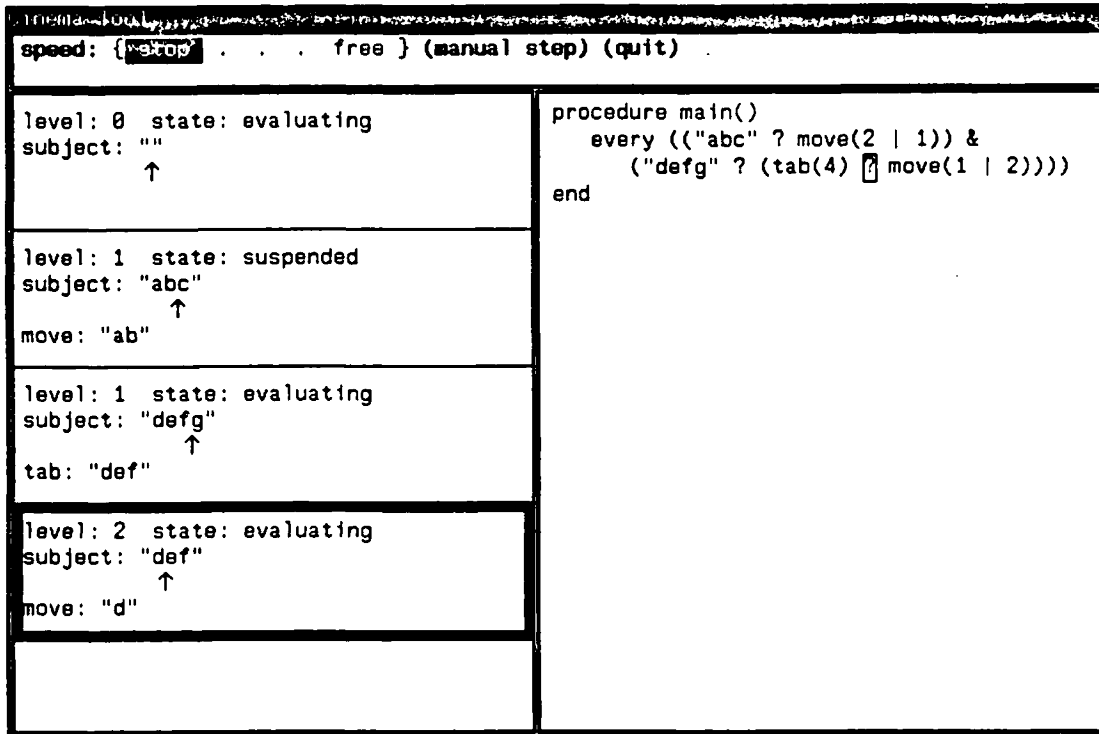
The program Cinema runs Icon programs and provides a display of string scanning that shows the program with scanning expressions highlighted, their scanning environments, and the results produced by matching functions. The user may step through scanning events, one by one, or allow the program to run without interruption, providing an animated display of string scanning. A history of display commands can be saved so that the display can be repeated without re-running the program.

In addition to Icon's built-in matching functions, a library of matching procedures corresponding to SNOBOL4 patterns is available [13] and the user can write additional matching procedures.

Cinema runs on Sun Workstations using Suntools [14] for window management. The Cinema display is controlled by using standard mouse functions.

5.1 Overview

A typical display is:



The top portion of the display is an options menu that allows the user to control Cinema. The right portion contains the program, while scanning environments appear at the left.

A sample Cinema session, showing various aspects of the display, is given in the appendix.

5.2 The Program Display

The program is displayed as it appears in the source file. Because of the limited size of the display, portions of the program may not appear in the window. The usual Suntool window functions can be used to adjust the position and size of the window during the display.

The scanning expression that is currently being evaluated is indicated by outlining its scanning operator, as illustrated in the example above.

5.3 Scanning Environments

Scanning environments are shown starting with the root environment at the top, and with the currently active environment outlined.

Four pieces of information are displayed for each scanning environment:

(1) the level

(2) the state of the scanning expression

(3) the value of &subject

(4) the value of &pos shown as an arrow

(5) the name of the last matching procedure to produce a result together with that result.

There may be more than one window at a given level, corresponding to scanning environments at the same level in the tree.

As string scanning progresses, the values in the current scanning environment may change. Each such change corresponds to an event in the scanning process. The possible events are:

(1) beginning the evaluation of a scanning expression

(2) a change in the value of &subject or &pos

(3) return of a result by a matching expression

(4) suspension of a scanning expression

(5) resumption of a scanning expression

(6) failure of a scanning expression

(7) removal of a scanning environment as a result of the completion of the bounded expression in which it occurs.

5.4 The Options Menu

The options menu consists of three components:

(1) a speed control

(2) a manual/step option

(3) a quit option

The speed control introduces a delay in the operation of Cinema. It ranges from infinite delay (stopped) to free running, in which case the speed is limited by the speed of program execution (which is slowed by driv-ing the display). The intermediate speeds correspond to introducing delays of 2, 1, and 0.5 seconds, respec-tively, between the display of events. Note that a change in the value of &pos during the evaluation of a matching function constitutes an event in addition to the one for the function's result.

The speed control initially is in the stopped position. The display can be started by selecting another speed setting or by selecting the manual option, which runs the program until the next event in a scanning process.

5.5 Running Cinema

Cinema is run as follows:

Cinema [*options*] *name* [*arguments*]

where *name* is an Icon program file without its .icn suffix. The available options are:

-c Compile the program to be displayed. In the absence of this option, the program is not compiled but is assumed to have been compiled previously.

-s Save a history of the display commands so the the display can be re-run without running the program. The history file is *name*.ste. In the absence of this option, the display commands are not saved.

-n Do not display the scanning events. In the absence of this option, scanning events are displayed. Cinema must be run under Suntools if the scanning events are displayed.

-h Run the display from commands previously saved by the -s option.

For example,

```
Cinema -c -s convert
```

compiles and executes the program in convert.icn, displaying the results and saving a history of the display commands.

Cinema creates a window for the display in front of the current window. When the cursor is positioned in the Cinema window, the display is controlled as described above, except that when the cursor is over the left

subwindow, the left mouse button is redefined. In this subwindow, placing the cursory over the display of an environment and pushing the left mouse button causes the corresponding scanning operation to be highlighted in the program.

Standard input to a program being run by Cinema is entered with the cursor positioned in the window that initiated Cinema. Standard output and error output are written to this window also.

6. User-Defined Matching Procedures

As described in Section 3, it is easy to write matching procedures that obey the same protocol and conventions as built-in matching functions. While the built-in matching functions have been modified to provide the display of the scanning events they produce, it is necessary to use the function Display in user-defined matching procedures. This function has the form

```
Display(s, label)
```

where s is the string produced by the matching procedure and label is an optional label that is displayed to the left of s in the scanning environment window. Display returns the value of s.

The function Display is used at the point in a user-defined matching procedure at which the result is produced. For example, a matching procedure Arb that models the SNOBOL4 pattern ARB and matches successively longer strings at the current position could be written as

```
procedure Arb()
   suspend Display(&subject[.&pos:&pos <- &pos to *&subject + 1], "Arb: ")
   fail
end
```

7. The Implementation

Cinema preprocesses the user's Icon program to produce a corresponding program that is run with a library of Icon procedures to support the display of scanning. An augmented Icon run-time system utilizing Icon's personalized interpreter system [15] provides the necessary functions to capture the events in the scan-ning process and to communicate with Cinema. Cinema is written in C so that it can use the Suntools window system.

7.1 Organization

Cinema starts by creating two pipes for communication between it and the Icon program. A child process is forked and the file descriptors of the pipes are placed in environment variables. The Icon program then is executed.

If the user has specified the creation of a history file, the name of the file is sent over the pipe from Cinema to the Icon program; otherwise an empty string is sent.

If the user has requested a display, the string "y" is sent over the pipe (the Icon program sends the display commands back); otherwise the string "n" is sent.

Whenever Cinema wants another display command from the program, it sends a synchronization signal, newline, to the program and then does a read from the other pipe. When the Icon program is ready to send a display command, it waits for a synchronization signal so that it does not get ahead of the display.

When displaying from a history file, Cinema reads from the file instead of from the pipe to the program. Synchronization signals are written to a dummy file.

7.2 Preprocessing

The Icon program is preprocessed for two reasons: to convert scanning expressions into nested procedure calls according to the model described in Section 2 and to enclose bounded expressions in procedure calls so that scanning environments can be removed when evaluation of a bounded expression is complete. The preprocessor was constructed using the variant translator system described in [16]. For a scanning expression,

the following translation is performed:

$$\tau(expr_1 \ ?\ expr_2) = \text{Escan}(\text{Bscan}(\tau(expr_1), col, line), \tau(expr_2))$$

where $\tau(expr)$ denotes the translations performed by the preprocessor. A similar translation is performed for augmented scanning expressions. In this translation, *col* and *line* are the column and line numbers of the ? operation in the source program. These extra arguments are used by Bscan to determine the location of the current scanning operator, so that it can be outlined.

Cinema maintains a tree of scanning environments. Procedures for removing scanning environments when the evaluation of a bounded expression is complete are provided as illustrated for the if-then expression:

$$\tau(\text{if } expr_1 \text{ then } expr_2) = \text{if Clear(Level()}, \tau(expr_1)) \text{ then } \tau(expr_2)$$

The procedure Level is evaluated first and increments the level of nesting in bounded expressions. When evaluation of the bounded expression $expr_1$ is complete, Clear removes any scanning environments created in the bounded expression and decrements the level of nesting in bounded expressions. Note that if $expr_1$ fails, any scanning expressions in it must also have failed and been removed. In this case, Level is resumed and decrements the level of nesting in bounded expressions. These procedures have no affect on the display if they do not enclose scanning expressions.

The overall result of preprocessing is illustrated by the following procedure

```
procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end
```

whose preprocessed counterpart is

```
procedure check(s)
   Clear(Level(), if Clear(Level(), Escan(Bscan(s, 9, 2), any(&ucase)))
      then return Clear(Level(), count(s))
      else return Clear(Level(), 0))
end
```

7.3 Library Procedures

In addition to the procedures Bscan, Escan, Clear, and Level mentioned above, there are a number of other procedures that are included in the library that is loaded with the preprocessed user program. These include Display mentioned in Section 6 as well as procedures called by Bscan and Escan to control the display.

In order to display the results produced by the matching functions tab and move, these functions are overloaded by Icon procedures that perform the same operations but also call Display. The string matching operation =s is converted into a corresponding procedure by the preprocessor.

7.4 Capturing Scanning Events

The procedures Bscan, Escan, and Clear handle events associated with the creation of scanning environments, their suspension or failure, and removal on the completion of a bounded expression. The results produced by matching procedures are handled by Display.

The two other scanning events that have to be captured are the assignments to &pos and &subject. These events are handled in the personalized interpreter by modifications to the routines that handle keyword assignments.

Acknowledgements

John Placer designed and implemented an earlier version of a cinematic display of string scanning, including the use of the history feature.

Dave Hanson and Janalee O'Bagy provided several suggestions about various aspects of Cinema and provided helpful advice on the presentation of the material in this report.

References

1. R. E. Griswold, *Understanding Pattern Matching — A Cinematic Display of String Scanning*, The Univ. of Arizona Tech. Rep. 83-14a, Feb. 1984.
2. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1971.
3. J. F. Gimpel, "A Theory of Discrete Patterns and Their Implementation in SNOBOL4", *Comm. ACM 16*, 2 (Feb. 1973), 91-100.
4. A. C. Fleck, "Formal Models for String Patterns", in *Current Trends in Programming Methodology; Data Structuring*, vol. IV, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978, 216-240.
5. R. D. Tennent, "Mathematical Semantics of SNOBOL4", *Proceedings of the ACM SIGACT News-SIGLAN Symposium on the Principles of Programming Languages*, 1973, 95-107.
6. A. De Bruin, *Operational and Denotational Semantics Describing the Matching Process in SNOBOL4*, Afdeling Informatica, Mathematisch Centrum, Amsterdam, 1980.
7. M. M. Siegel, *Proving Properties of SNOBOL4 Patterns*, Doctoral Dissertation, Cornell University, 1980.
8. W. M. Waite, *Implementing Software for Non-Numeric Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.
9. F. C. Druseikis and J. N. Doyle, "A Procedural Approach to Pattern Matching in SNOBOL4", *Proceedings of the ACM Annual Conference*, 1974, 311-317.
10. J. N. Doyle, *A Generalized Facility for the Analysis and Synthesis of Strings and a Procedure-Based Model of an Implementation*, SNOBOL4 Project Document S4D38, University of Arizona, Feb. 1975.
11. P. Emanuelson, *Performance Enhancement in a Well-Structured Pattern Matcher Through Partial Evaluation*, PhD Thesis, Linkoping University, 1980.
12. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
13. R. E. Griswold, *The Icon Program Library; Version 5.10*, The Univ. of Arizona Tech. Rep. 85-18, Aug. 1985.
14. Sun Micorsystems, Inc., *Command Reference Manual for the Sun Workstation; Suntools(1)*, 1985.
15. R. E. Griswold and W. H. Mitchell, *Personalized Interpreters for Version 5.10 of Icon*, The Univ. of Arizona Tech. Rep. 85-17, Aug. 1985.
16. R. E. Griswold, *The Construction of Variant Translators for Icon*, The Univ. of Arizona Tech. Rep. 83-19a, June 1984.

Appendix — A Sample Cinema Session

The display produced by Cinema depends for its effectiveness on user control and in some situations on the visual impression of progressing rapidly through a number of events. These effects cannot be captured in a printed document, but some understanding of the nature of the display can be acquired from the following example, which shows most of the scanning events that occur during the execution of a simple program.

The program counts the number of lowercase vowels in words that begin with an uppercase letter:

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The main procedure reads lines, finds words, and calls check to process them. The procedure check tests for an initial uppercase letter and calls count if there is one. The procedure count in turn counts the vowels. This program is more complicated than need be for the computation performed, but it is designed specifically to illustrate various aspects of Cinema. In addition, the definitions of "words" and "vowels" are deliberately naive to avoid complications that are unrelated to Cinema.

The displays that follow show the operation of the program for the input line

```
(Count George out.)
```

The initial display, prior to input, shows only the root scanning environment:

```
speed: { stop . . . free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The next scanning event occurs when the line has been read and the scanning expression in the main procedure is evaluated:

```
speed: { stop . . . free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
          ↑

procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The value of &pos is set to 2 by the first tab, resulting in the display:

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
          ↑                             total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
           ↑                            write(total)
                                     end

                                     procedure check(s)
                                        if s ? any(&ucase) then return count(s)
                                        else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

Next tab returns the matched substring:

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
          ↑                             total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
           ↑                            write(total)
tab: "("                             end

                                     procedure check(s)
                                        if s ? any(&ucase) then return count(s)
                                        else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

Note that there are two events involved in the evaluation of a matching function: the setting of &pos and the return of the matched substring. In the remainder of this appendix only the display for the return of the

matched substring is shown. Next the second tab matches the first word:

```
Cinema-tool
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
         ↑                              total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
                ↑                       write(total)
tab: "Count"                         end

                                     procedure check(s)
                                        if s ? any(&ucase) then return count(s)
                                        else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

At this point, check is called and its scanning expression produces a scanning environment at level 2 with the subject "Count":

```
Cinema-tool
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
         ↑                              total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
                ↑                       write(total)
tab: "Count"                         end

level: 2  state: evaluating          procedure check(s)
subject: "Count"                        if s ? any(&ucase) then return count(s)
         ↑                              else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

An uppercase letter is found and this scanning expression suspends. The scanning expression in the main

procedure that resulted in the call to check is now "current", even though evaluation is still taking place in check. This situation is displayed as follows:

```
speed: { stop  .  .  .  free } (manual step) (quit)
```

```
level: 0  state: evaluating
subject: ""
         ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                ↑
tab: "Count"

level: 2  state: suspended
subject: "Count"
         ↑
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

In check, the bounded expression in the control clause of the if-then expression then is exited and the suspended scanning environment in check is removed:

```
speed: { stop  .  .  .  free } (manual step) (quit)
```

```
level: 0  state: evaluating
subject: ""
         ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                ↑
tab: "Count"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The procedure count then is called:

```
speed: { stop . . . free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                ↑
tab: "Count"

level: 2  state: evaluating
subject: "Count"
          ↑
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

Note that its scanning environment is at level 2. The scanning expression in count now tabs up to a vowel:

```
speed: { stop . . . free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                ↑
tab: "Count"

level: 2  state: evaluating
subject: "Count"
           ↑
tab: "C"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

Next a substring of vowels is matched:

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                ↑
tab: "Count"

level: 2  state: evaluating
subject: "Count"
             ↑
tab: "ou"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

Since there are no more vowels in the word, the scanning expression fails:

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                ↑
tab: "Count"

level: 2  state: failing
subject: "Count"
             ↑
tab: "ou"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

Control returns to the main procedure where the beginning of the next word is found (the two intermediate events are not shown):

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                 ↑
tab: " "
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The next word is matched:

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                        ↑
tab: "George"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

At this point, check processes the word as before. Skipping the three intermediate displays, count is called:

```
Cinema Tool
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
         ↑                              total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
                 ↑                      write(total)
tab: "George"                        end

level: 2  state: evaluating          procedure check(s)
subject: "George"                       if s ? any(&ucase) then return count(s)
         ↑                              else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

The first vowel is found as before:

```
Cinema Tool
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
         ↑                              total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
                 ↑                      write(total)
tab: "George"                        end

level: 2  state: evaluating          procedure check(s)
subject: "George"                       if s ? any(&ucase) then return count(s)
          ↑                             else return 0
tab: "G"                             end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

The vowels are matched:

```
Cinema tool
speed: {stop . . . free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                        ↑
tab: "George"

level: 2  state: evaluating
subject: "George"
             ↑
tab: "eo"

procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The next vowel then is found:

```
Cinema tool
speed: {stop . . . free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                        ↑
tab: "George"

level: 2  state: evaluating
subject: "George"
               ↑
tab: "rg"

procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

This vowel is matched and control is again returned to the main procedure (intermediate events are not shown):

```
speed: { stop .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
         ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                         ↑
tab: "George"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

The last word is located and matched:

```
speed: { stop .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
         ↑

level: 1  state: evaluating
subject: "(Count George out.)"
                             ↑
tab: "out"
```

```
procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```

Since this word does not begin with an uppercase letter, check fails:

```
Cinema-Tool
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
          ↑                             total := 0
                                        while line := read() do
level: 1  state: evaluating                line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
                           ↑            write(total)
tab: "out"                           end

level: 2  state: failing             procedure check(s)
subject: "out"                          if s ? any(&ucase) then return count(s)
          ↑                             else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

The scanning expression in the main procedure fails, since there are no more words:

```
Cinema-Tool
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating          procedure main()
subject: ""                             let := &lcase ++ &ucase
          ↑                             total := 0
                                        while line := read() do
level: 1  state: failing                   line ? while tab(upto(let)) do
subject: "(Count George out.)"                total +:= check(tab(many(let)))
                           ↑            write(total)
tab: "out"                           end

                                     procedure check(s)
                                        if s ? any(&ucase) then return count(s)
                                        else return 0
                                     end

                                     procedure count(s)
                                        static lvowel
                                        initial lvowel := 'aeiou'
                                        i := 0
                                        s ? while tab(upto(lvowel)) do
                                           i +:= *tab(many(lvowel))
                                        return i
                                     end
```

At this point, the display is the same as it was at the beginning of program execution, with only the root scanning environment remaining. If there is no more input, the main procedure fails and Cinema completes.

```
speed: { stop  .  .  .  free } (manual step) (quit)

level: 0  state: evaluating
subject: ""
          ↑

procedure main()
   let := &lcase ++ &ucase
   total := 0
   while line := read() do
      line ? while tab(upto(let)) do
         total +:= check(tab(many(let)))
   write(total)
end

procedure check(s)
   if s ? any(&ucase) then return count(s)
   else return 0
end

procedure count(s)
   static lvowel
   initial lvowel := 'aeiou'
   i := 0
   s ? while tab(upto(lvowel)) do
      i +:= *tab(many(lvowel))
   return i
end
```