Understanding Pattern Matching —
A Cinematic Display of String Scanning

Ralph E. Griswold

TR 83-14a

October 22, 1983; Revised February 28, 1984

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.
Understanding Pattern Matching —
A Cinematic Display of String Scanning

1. Introduction

String pattern matching in the style of SNOBOL4 is easy to understand in a general, intuitive way. This accounts for the ease with which it can be learned and used. However, the actual process by which pattern matching takes place is generally poorly understood. Consequently, the implementation of pattern matching traditionally has been ad hoc and generalizations and extensions to it have been inhibited.

There have been numerous approaches to describing pattern matching, including “bead diagrams” [1], cursor-position transformations [2], formal algebraic models [3], denotational semantics [4, 5], axiomatic semantics [6], as well as implementation models [7-10].

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.

The report describes a program that produces a “cinematic” display of pattern matching in which the user can watch the process as it takes place, step by step, and observe both the details and the dynamics of the process.

This program supports Icon string scanning and SNOBOL4-style pattern matching. It adds a new dimension to Icon with unanchored string scanning, in which the scan need not start at the beginning of the subject. A number of extensions are provided to the standard SNOBOL4 repertoire. In addition, all of the control structures of Icon can be used in conjunction with pattern matching. Thus this program can be used for the experimental development of new pattern-matching facilities.

The internals of this program are also described, showing how pattern matching is implemented.

The specific focus for this material is string scanning in Icon rather than pattern matching in SNOBOL4. Icon string scanning is somewhat more general than SNOBOL4 pattern matching, and SNOBOL4 pattern matching is easy to model in Icon [11-13]. The reader should be familiar with SNOBOL4 [1] and Icon [14] in order to understand the material that follows.

2. String Scanning States

A string scanning expression has the form

\[ expr_1 \mathbin{?} expr_2 \]

where \( expr_1 \) provides a subject that is processed by \( expr_2 \). The order of evaluation can be expressed in terms of a state diagram:
In this diagram, \( E \) is the initial state, corresponding to the initiation of evaluation of the scanning expression, \( e_1 \) and \( e_2 \) represent the evaluation of \( expr_1 \) and \( expr_2 \), respectively, and \( s \) and \( f \) indicate the success and failure of evaluation, respectively. \( F \) indicates a terminal state in which the scanning expression has failed, while \( S \) indicates a quasi-terminal state in which the scanning expression has succeeded and produced a value. \( S \) is a state of suspension. The dashed arrow indicates that the scanning expression may be resumed by an enclosing expression in order to produce another result. Note that \( e_1 \) and \( e_2 \) can be resumed to produce additional results.

Consider the following simple example:

"abc" \ ? move(2)

The evaluation of \( expr_1 \) succeeds and produces \( abc \). The evaluation of \( expr_2 \) also succeeds and produces \( ab \). The state sequence is \( E-e_1-e_2-S \).

On the other hand, the expression

"abc" \ ? move(4)

fails, with the state sequence \( E-e_1-e_2-e_1-F \). Note that when \( e_2 \) fails, \( e_1 \) is resumed. It then fails, since "abc" can produce only one result.

These two examples represent the commonest situations in pattern matching: the success or failure of \( expr_1 \). There are many other possibilities. For example, \( expr_1 \) may fail initially, as in

("ABC" \ => "abc") \ ? move(2)

which has the state sequence \( E-e_1-F \). A more interesting situation occurs if \( expr_1 \) can produce more than one result, as in

("abc" | "defgh") \ ? move(4)

The state sequence here is \( E-e_1-e_2-e_1-e_2-S \), since the resumption of \( expr_1 \) produces \( defgh \), which is matched by \( move(4) \).

The significance of an expression that encloses a scanning expression is illustrated by

("abc" \ ? move(2)) \ ===> "de"

Here the left argument of the comparison is the same as for the first example in this section. The state sequence is \( E-e_1-e_2-S-e_2-e_1-F \), since the comparison of \( ab \) with \( de \) fails, causing \( expr_2 \) to be resumed, ultimately leading to the failure of the entire expression.

Failure of a scanning expression does not necessarily mean that the expression does no useful computation. Consider

\[
\text{every write(\text{"abc" } ? \text{move(1 to 3)})}
\]

This expression writes \( a \), \( ab \), and \( abc \). Its state sequence is \( E-e_1-e_2-S-e_2-S-e_2-e_1-F \).
3. A Cinematic Display

State sequences only describe part of string scanning. What is lacking is the subject, the position in it at which matching is taking place, and the value that is produced. This information, together with the state, is referred to as the status of string scanning. While this information can be presented in a linear or tabular form, the dynamics of the process are easier to grasp if the information is displayed pictorially, with the status changing as string scanning process takes place.

The program Cinema executes an Icon program and produces a display of string scanning. The display is two-dimensional and screen-oriented, with windows as shown below:

```
      scanning status
         
        program text
         
         input/output
```

The text of the program itself is displayed on the left side of the screen. Because of the limited space available, program lines are truncated at 50 characters and only the first 24 lines are shown.

The top right portion of the screen is a status window, which contains five pieces of information:

1. the state of expression evaluation
2. the subject of scanning
3. the initial cursor position
4. the scanning cursor position
5. the portion of the subject that has been matched.

The initial cursor position is always 1 in Icon, but may change in SNOBOL4 in the unanchored mode of pattern matching (See Section 4.1). The initial cursor position is included here to allow Cinema to be used for studying SNOBOL4 pattern matching.

The portion of the subject that has been matched is between the initial and scanning cursor positions, and is highlighted in the status window*. The initial cursor is shown as \( \downarrow \), while the scanning cursor is shown as \( \wedge \). For example, in the expression

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

the status when state S is reached is shown as

---

*The method used for highlighting is dependent on terminal characteristics. In this report, it is shown as underlining, which is the highlighting method used for the DataMedia 3045.
Note that the cursors are displayed to the left of corresponding characters in the subject. There is no practical way to display them between the characters of the subject on a terminal screen. For example, the screen at the completion of

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

is

```
procedure main()
"abc" ? move(2)
end
```

The lower right portion of the screen is reserved for user input and output. For example, the screen at the completion of

```
"abc" ? write(move(2))
```

is

```
procedure main()
"abc" ? write(move(2))
end
```
Since there may be many scanning operators in a program, it is important to be able to determine the operator that is currently being evaluated. This is done by highlighting the active scanning operator in the program display on the left side of the screen. Consider

"test" ? (move(2) ? tab(uplo('aeiou')))\[7.5cm\]

Evaluating $expr_1$ for the left scanning operator involves evaluating the right scanning operator. At the completion of $move(2)$, the display is

Note that in nested scanning such as this, there is a status window for each scanning operator that is active or suspended.

When Cinema is running, the screen changes as string scanning progresses, providing a "motion picture" of the dynamically changing status. This gives an overall view of the dynamics of string scanning and is particularly useful for observing backtracking and the combinatorial nature of many scanning expressions.

To study a particular aspect of pattern matching, however, a "slow motion" single-step mode is provided. If the value of identifier Single is 1, the display stops every time the state changes and proceeds only after a carriage return by the user. The value of Single can be changed during program execution, as in

\[
every \ expr_1 \ ? \ (\text{Single} \leftarrow 1, \ expr_2, \ \text{Single} \leftarrow 0, \ expr_3)
\]

which single steps during the evaluation of $expr_2$ but not during the evaluation of $expr_3$.

4. SNOBOL4 Pattern Matching

In earlier work [13], SNOBOL4 patterns were implemented using Icon procedures. For example, the SNOBOL4 pattern ARB is implemented by an Icon procedure Arb(), and so on. Initial uppercase letters are used to distinguish these procedures from the actual patterns of SNOBOL4. Using these procedures, a SNOBOL4 pattern-matching statement such as

\[
s \ ? \ \text{LEN(3) BAL $ OUTPUT}
\]

can be cast in Icon as
s ? Len(3) || write(Bal())

The complete collection of pattern-matching procedures is contained in the Icon program library [15]. These procedures are available in Cinema. A library manual page that describes these procedures is included as Appendix A to this report and a listing of the relevant procedures is given in Appendix B.

4.1 The Unanchored Mode

In the unanchored mode of pattern matching in SNOBOL4, if evaluation of expr₁ fails, the initial cursor position is incremented and expr₂ is evaluated again. This adds another state to the state diagram for pattern matching:

```
E  --s-- e₁  --s-- e₂  --s-- S
    |      |      |
    f      f      f

F
```

Evaluation in state m fails if the initial cursor position is at the end of the subject.

The unanchored mode of pattern matching is set by assigning the value 0 to Anchor*. For example,

```
Anchor := 0
s ? Len(3) || write(Bal())
```

performs the previous pattern match in the unanchored mode. The unanchored mode also can be used in conjunction with Icon string scanning, as in

```
Anchor := 0
every "abc" ? write(move(2))
```

which writes ab and bc.

The default mode is anchored (which is different from the default in SNOBOL4). The value of Anchor is tested at the first evaluation of expr₁ and remains in effect for subsequent resumptions of expr₂. If its value is changed, the change does not take effect until the next evaluation of expr₂.

5. Running Cinema

Cinema is run by

```
Cinema [ options ] file
```

where file is the name of an Icon program (ending in .icon).

Cinema translates, links, and executes file, producing the cinematic display described in the preceding sections.

*This is slightly different from the technique used if the procedures are used outside of Cinema. The pattern-matching procedures in the library can be used without modification, however.
The available options are:
  -u Set Anchor to 0 initially (unanchored mode)
  -s Set Single to 1 initially (single-step mode)

The defaults are anchored and not single stepped.

6. Suggested Exercises

The following short programs are suggested as exercises. None of them performs any significant computation and some of them are pathological, but they test understanding of the pattern-matching process. The reader should examine each program first to determine what it does and then run it under Cinema. It is instructive to run the programs in both the anchored and unanchored modes.

Program 1:

```plaintext
procedure main()
  s := "This is a test"
  s ? (tab(1 to 10) & tab(any('aeiou')))
end
```

Program 2:

```plaintext
procedure main()
  s := "This is a test"
  s ? (tab(1 to 10) & tab(upto('aeiou')))
end
```

Program 3:

```plaintext
procedure main()
  s := "This is a test"
  every s ? (tab(1 to 10) & tab(any('aeiou')))
end
```

Program 4:

```plaintext
procedure main()
  s := "This is a test"
  every s ? (tab(1 to 10) & tab(upto('aeiou')))
end
```

Program 5:

```plaintext
procedure main()
  s := "This is a test"
  every s ? tab(1 to 4) ?
    tab(1 to 4)
end
```
Program 6:

procedure main()
  s := "This is a test"
  every s ? (tab(upto('aeiou')) & move(1 to 3))
end

Program 7:

procedure main()
  s := "This is a test"
  every s ? (tab(upto('aeiou')) ? move(1 to 3))
end

Program 8:

procedure main()
  s := "This is a test"
  every s ? Break('aeiou')
end

Program 9:

procedure main()
  s := "This is a test"
  every s ? Breakx('aeiou')
end

Program 10:

procedure main()
  s := "This is a test"
  every s ? Arb()
end

Program 11:

procedure main()
  s := "This is a test"
  every (s ? Arb()) ? Arb()
end

Program 12:

procedure main()
  s := "(x+y)*z"
  every s ? Bal()
end

Program 13:

procedure main()
  s := "(x+y)*z"
  every (s ? Bal()) ? Bal()
end
Program 14:

```plaintext
procedure main()
    s := "(x+y)*z"
    every s ? (Bal()) ? Bal()
end
```

7. The Implementation of Cinema

7.1 String Scanning

There are two parts to the implementation of string scanning: the scanning control structure itself and scanning operations that apply to the subject at the scanning cursor position. These two parts are treated separately in the following sections.

7.1.1 The Scanning Control Structure

The expression

```plaintext
expr₁ ? expr₂
```

is a control structure and its evaluation differs from that of functions and operations. In particular, &subject is set to the value produced by the evaluation of expr₁ before expr₂ is evaluated. Consequently, the scanning control structure cannot be modeled simply by a procedure call. Instead, the scanning control structure is implemented as a programmer-defined control operation [16]. A preprocessor converts all instances of

```plaintext
expr₁ ? expr₂
```

into

```plaintext
Scan(create expr₁, create expr₂)
```

Consequently, when Scan is invoked, expr₁ and expr₂ are not evaluated before the procedure gains control.

In order to understand string scanning, it is instructive to look at a simple model first. This model shows how the state diagram for string scanning is reflected in a procedure, but it does not save the values of &subject and &pos. Therefore it can be used for simple string scanning but not for nested scanning expressions. The procedure Scan for this simple model is:

```plaintext
procedure Scan(e₁, e₂)
    local value
    while &subject := @e₁ do {  # state E
        &pos := 1  # state e₁
        while value := @e₂ do  # state e₂
            suspend value
            e₂ := \e₂
        fail  # state S
    }
end
```

where Scan is called as shown above. Thus, e₁ is a co-expression for expr₁. This co-expression is repeatedly activated to produce new values for &subject. For each new value of &subject, &pos is set to 1. This is redundant, since assignment to &subject in Icon automatically sets &pos to 1, but it is included for clarity. Next e₂, the co-expression for expr₂, is refreshed. This is unnecessary the first time through the loop, but is required for subsequent iterations. In the inner loop, the co-expression for expr₂ is repeatedly activated to perform the scanning. The procedure suspends for each value produced. When activation of the co-expression for expr₂ fails, the outer loop continues by activating the co-expression for expr₁. When this loop terminates, Scan fails.
In order to allow nested scanning, it is necessary to add code to this procedure to save and restore the values of &subject and &pos at appropriate places. This requires a thorough understanding of string scanning and what may occur in complex nested scanning expressions. The procedure is:

```plaintext
procedure Scan(e1, e2)
    local nsb, value
    local sb, pos
    local sb2, pos2, xpos
    while nsb := @e1 do {
        sb1 := &subject
        pos1 := pos2 := &pos
        &subject := nsb
        &pos := 1
        repeat {
            sb2 := sb1
            pos2 := pos1
            value := @e2 | break
            xpos := &pos
            &subject := sb2
            pos2 := xpos
            &pos := xpos
            suspend value
            &subject := sb2
            &pos := pos2
            e2 := &e2
        }
        &subject := sb1
        &pos := pos1
    }
    fail
end
```

It is important to note that the evaluation of \texttt{expr} may change &subject and &pos outside the scanning expression. This occurs in situations such as

```plaintext
text ? (tab(many(wchar)) | write(tab(upto(vowel))))
```

where \texttt{expr} for the right scanning expression is obtained by scanning the subject in the left scanning expression. On the other hand, evaluation of \texttt{expr} must not change &subject or &pos in an outer scanning expression. Consequently, &subject and &pos must be saved before \texttt{e2} is activated and restored after it returns.

A dodge is necessary in saving and restoring &pos in the inner loop, since assignment to &subject automatically sets &pos to 1. The local identifier xpos is used as an alternate value for &pos.

Introducing the unanchored mode adds another loop. The general version of Scan that supports unanchored pattern matching follows. Additions for handling the initial cursor are marked by #s.
global Anchor

procedure Scan(e1,e2)
    local nsubject, value
    local subject1, pos1
    local subject2, pos2, xpos
    while nsubject := @e1 do {
        subject1 := &subject
        pos1 := &pos
        &subject := nsubject
        every &pos := 1 to maxpos() do {
            repeat {
                subject2 := subject1
                pos2 := pos1
                value := @e2 | break
                xpos := &pos
                &subject := subject2
                pos2 := xpos
                &pos := xpos
                suspend value
                &subject := subject2
                &pos := pos2
            }
            e2 := &e2
        }
        &subject := subject1
        &pos := pos1
    }
    fail
end

procedure maxpos()
    return if Anchor == 0 then *&subject + 1 else 1
end

General object comparison is used for testing the value of Anchor in maxpos. This allows Anchor not to be set at all by programs that operate in the anchored mode.

7.1.2 Displaying Scanning

In order to display scanning, it is necessary to add calls to procedures that maintain the windows of the display to the scanning procedure. There are eight procedures involved:

decr() decrement display level
incr() increment display level
init() initialize the display
newwin() create a new status window
state(s) write the state s in the current status window
snapshot() write &subject, &pos, and the initial cursor position in the current status window and highlight the portion of &subject between initial cursor position and &pos
mark(loc) highlight the scanning operator at the location loc
unmark(loc) remove highlighting from the scanning operator at the location loc

The location of the current scanning operator is given by the global identifier Loc, which is a list containing the row and column positions of the operator in the program. This information is provided by the preprocessor, which translates

\[ expr_1 \ ? \ expr_2 \]

into

\[ \{ \text{Loc} := [i,j]; \text{Scan}(\text{create } expr_1, \text{create } expr_2) \} \]

where \( i \) and \( j \) are the column and line numbers. Thus, when Scan is called, Loc has the required position information.

The identifier ipos, whose value is the initial cursor position, is added for use by the display procedures.

The procedure Scan with calls to the display procedures follows. Additions for handling the display are marked by #.

```plaintext
global Anchor

global Loc, ipos

procedure Scan(e1, e2)
local nssubject, value
local subject1, pos2, ipos1
local subject2, pos2, ipos2
local loc, ipos2
initial {
    init()
    ipos := 1
}
incr()
loc := Loc
newwin()
mark(loc)
state("E")
repeat {
    state("e1")
    unmark(loc)
    nssubject := @e1 | break
    mark(loc)
    subject1 := &subject
    pos1 := &pos
    ipos1 := ipos
    &subject := nssubject
    snapshot()
    state("m")
```

-12-
every &pos := i pos := 1 to maxpos() do { #
snapshot() #
repeat {
  state("e2") #
  unmark(loc) #
  subject2 := subject1
  pos2 := pos1
  i pos2 := i pos1
  value := @e2 | break
  mark(loc) #
  state("S") #
  snapshot() #
  decr() #
  x pos := &pos
  &subject := subject2
  pos2 := x pos
  i pos2 := i pos
  &pos := x pos
  unmark(loc) #
  suspend value
  mark(loc) #
  &subject := subject2
  &pos := pos2
  i pos := i pos2
  incr() #
}
e2 := ^e2
&subject := subject1
&pos := pos1
i pos := i pos1
}
state("F") #
decr() #
unmark(loc) #
fail
end

Note the preponderance of programs lines related to the display.

The display procedures themselves have no direct relation to string scanning, but they are listed in Appendix C for reference. The Cinema program itself is listed in Appendix D.

7.2 Scanning Operations

The built-in scanning operations do not participate in the display and can be used without modification in Cinema. Appendix A illustrates programmer-defined scanning operations cast in the style of SNOBOL4. See [12] for examples of other programmer-defined scanning operations. Most of these are simple and use the built-in matching functions of Icon. Thus Len(i) is simply

    procedure Len(i)
    suspend move(i)
    end

The screen display is not affected by the evaluation of a scanning operation such as move(i). Instead, move(i) changes the value of &pos and this is reflected in the display when Scan gets control again — only Scan updates the display. This normally provides enough detail, but the display procedures also can be called
from programmer-defined scanning operations. The two relevant procedures are \texttt{state(s)} and \texttt{snapshot()}. 

For example, calls to display procedures could be added to \texttt{Len(i)} as follows:

\begin{verbatim}
procedure Len(i)
    local s
    s := State
    suspend 2(state("Len"), move(i), snapshot(), state(s))
    state(s)
end
\end{verbatim}

The state display is changed to \texttt{Len} and if \texttt{move(i)} succeeds, \texttt{snapshot()} reflects its effect on \&pos. The global identifier \texttt{State} contains the last displayed state. It is saved in the procedure above so that it can be restore before \texttt{Len(i)} returns.

Display states are truncated at three characters because of the limited space in the status window.

Acknowledgement

Dave Hanson, Bill Mitchell, and Steve Wampler made a number of helpful suggestions on the presentation of the material in this report.

References


NAME
patterns – SNOBOL4-style pattern matching

DESCRIPTION
These procedures are adapted from TR 80-25 and TR 81-6. They provide procedural equivalents for most SNOBOL4 patterns and some extensions.

SYNOPSIS

<table>
<thead>
<tr>
<th>Any(s)</th>
<th>ANY(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arb()</td>
<td>ARB</td>
</tr>
<tr>
<td>Arbno(p)</td>
<td>ARBNO(P)</td>
</tr>
<tr>
<td>Arbx(i)</td>
<td>ARB(I)</td>
</tr>
<tr>
<td>Bal()</td>
<td>BAL</td>
</tr>
<tr>
<td>Break(s)</td>
<td>BREAK(S)</td>
</tr>
<tr>
<td>Breakx(s)</td>
<td>BREAKX(S)</td>
</tr>
<tr>
<td>Cat(p1, p2)</td>
<td>P1 P2</td>
</tr>
<tr>
<td>Discard(p)</td>
<td>/P</td>
</tr>
<tr>
<td>Exog(s)</td>
<td>\S</td>
</tr>
<tr>
<td>Find(s)</td>
<td>FIND(S)</td>
</tr>
<tr>
<td>Len(i)</td>
<td>LEN(I)</td>
</tr>
<tr>
<td>Limit(p, i)</td>
<td>P \ i</td>
</tr>
<tr>
<td>Locate(p)</td>
<td>LOCATE(P)</td>
</tr>
<tr>
<td>Marb()</td>
<td>longest-first ARB</td>
</tr>
<tr>
<td>Notany(s)</td>
<td>NOTANY(S)</td>
</tr>
<tr>
<td>Pos(i)</td>
<td>POS(I)</td>
</tr>
<tr>
<td>Replace(p, s)</td>
<td>P = S</td>
</tr>
<tr>
<td>Rpos(i)</td>
<td>RPOS(I)</td>
</tr>
<tr>
<td>Rtab(i)</td>
<td>RTAB(I)</td>
</tr>
<tr>
<td>Span(s)</td>
<td>SPAN(S)</td>
</tr>
<tr>
<td>String(s)</td>
<td>S</td>
</tr>
<tr>
<td>Succeed()</td>
<td>SUCCEED</td>
</tr>
<tr>
<td>Tab(i)</td>
<td>TAB(I)</td>
</tr>
<tr>
<td>Xform(f, p)</td>
<td>F(P)</td>
</tr>
</tbody>
</table>

In addition to the procedures above, the following expressions can be used:

\[
\begin{align*}
p1() & | p2() \\
v & <= p() \\
v & := p() \\
fail & \\
= & s \\
p1() & || p2()
\end{align*}
\]

Using this system, most SNOBOL4 patterns can be satisfactorily transliterated into ICON procedures and expressions. For example, the pattern

\[
\text{SPAN}("0123456789") \ $ \ N \ "H" \ \text{LEN}(\star N) \ $ \ \text{LITERAL}
\]

can be transliterated into
(n <- Span('0123456789')) || ="H" || (literal <- Len(n))

Concatenation of components is necessary to preserve the pattern-matching properties of SNOBOL4. See the documents listed below for details and limitations.

CAVEATS
Simulating SNOBOL4 pattern matching using the procedures above is inefficient.

SEE ALSO
procedure Any(s)  # ANY(S)
    suspend tab(any(s))
end

procedure Arb()  # ARB
    suspend tab(&pos to *subject + 1)
end

procedure Arbno(p)  # ARBNO(P)
    suspend "" | (p() || Arbno(p))
end

procedure Arbx(i)  # ARBX(I)
    suspend tab(&pos to *subject + 1 by i)
end

procedure Bal()  # BAL
    suspend Bbal() || Arbno(Bbal)
end

procedure Bbal()  # used by Bal()
    suspend (="" || Arbno(Bbal) || =")") || Notany(""()"")
end

procedure Break(s)  # BREAK(S)
    suspend tab(upto(s) \ 1)
end

procedure Breakx(s)  # BREAKX(S)
    suspend tab(upto(s))
end

procedure Cat(p1, p2)  # P1 P2
    suspend p1() || p2()
end

procedure Discard(p)  # /P
    suspend p() & ""
end

procedure Exog(s)  # \S
    suspend s
end

procedure Find(s)  # FIND(S)
    suspend tab(find(s) + 1)
end
procedure Len(i)
  suspend move(i)
end  # LEN(I)

procedure Limit(p, i)
  local j
  j := &pos
  suspend p() \ i
  &pos := j
end  # P \ i

procedure Locate(p)
  suspend Arb() & p()
end  # LOCATE(P)

procedure Marb()
  suspend tab(+&subject + 1 to &pos by -1)
end  # max-first ARB

procedure Notany(s)
  suspend tab(any(-s))
end  # NOTANY(S)

procedure Pos(i)
  suspend pos(i + 1) & ""
end  # POS(I)

procedure Replace(p, s)
  suspend p() & s
end  # P = S

procedure Rpos(i)
  suspend pos(-i) & ""
end  # RPOS(I)

procedure Rtab(i)
  suspend tab(-i)
end  # RTAB(I)

procedure Span(s)
  suspend tab(many(s))
end  # SPAN(S)

procedure String(s)
  suspend =s
end  # S

procedure Succeed()
  suspend |""
end  # SUCCEED
procedure Tab(i) # TAB(I)
    suspend tab(i + 1)
end

procedure Xform(f, p) # F(P)
    suspend f(p())
end
Appendix C — Display Procedures for the DataMedia 3045

The implementation of the display procedures is dependent on terminal characteristics. Examples of the procedures for the DataMedia 3045 follow. On the DataMedia 3045, the underscore is non-destructive and is used for highlighting both the portion of &subject that is currently matched (see snapshot()) and also the current scanning operation (see mark(loc) and unmark(loc)).

No attempt has been made to optimize cursor motion.

global row, col, cm, ce, scol, dcol, bar, slevel, Single, State

# decr() decrements the status window display level.
#
procedure decr()
    slevel := 1
    return
end

# incr() increments the status window display level.
#
procedure incr()
    slevel := 1
    return
end

# init() initializes variables used by the display procedures.
#
procedure init()
    row := &cset[33*:24]  # row offsets for cursor position
    col := &cset[33*:80]  # column offsets for cursor position
    cm := "\^[Y"               # cursor motion character
    ce := "\^[K"               # clear line character
    dcol := 51              # screen division column
    scol := dcol + 4        # column for state information
    bar := repl("-", 80 - dcol)  # divide screen
    every xy(dcol - 1, 1 to 24, ",")  # mark off user i/o window
    xy(dcol, 19, bar)           # initial screen slevel
    slevel := 0
    if \useL then Anchor := 0  # set Anchor for -u
    if \ssetL then Single := 1  # set single stepping for -s
    return
end

# mark(loc) highlights the ? symbol at coordinates given by loc.
#
procedure mark(loc)
    xy(loc[1], loc[2], ",")
    xy(dcol, 20)
    return
end
# min(i, j) returns the minimum of i and j.
#
procedure min(i, j)
  return if i < j then i else j
end

# newwin() sets up a new scanning window.
#
procedure newwin()
  every xy(scol, 3 * slevel − (2 | 1), ce)
   xy(dcol, 3 * slevel, bar)
   return
end

# snapshot() provides a snapshot of the state of scanning.
#
procedure snapshot()
  xy(scol, 3 * slevel − 2, ce) # clear line and write subject
  xy(scol, 3 * slevel − 2, image(&subject))
  xy(scol, 3 * slevel − 1, ce) # clear line for ipos & &pos
  xy(scol + ipos, 3 * slevel − 1, "|")
  xy(scol + &pos, 3 * slevel − 1, "^")
  if &pos <= ipos then # highlight nonempty string
    xy(scol + min(&pos, ipos), 3 * slevel − 2, repl("","", abs(&pos − ipos)))
  every xy(dcol, 24 to 20 by −1, ce) # clear input/output window
  xy(dcol, 24) # reposition cursor
  return
end

# state(s) updates the state identification.
#
procedure state(s)
  State := s
  xy(dcol, 3 * slevel − 2, left(s, 3))
  xy(dcol, 20, ce)
  if Single === 1 then read() # single stepping
  return
end

# unmark(loc) removes highlighting from ? symbol at specified coordinates.
#
procedure unmark(loc)
  xy(loc[1] + 1, loc[2], "\b \b?")
  xy(dcol, 20)
  return
end
# xy(x, y, s) moves screen cursor to (x, y) and writes s.
# Note (x,y) coordinates out of range of the screen produce no output.
#
procedure xy(x, y, s)
    writes(cm, col[x], row[y], s)
    return
end
Appendix D — The Cinema Driver

The Cinema program proper takes options and a program name on the command line. An option causes a corresponding dummy procedure to be linked with the program. This in turn causes the corresponding variable name to be nonnull, which is tested for in init() (see Appendix C). For example, the -s option causes sset.u1 to be linked with the program. sset.u1 contains a dummy procedure sset_(.), which causes this identifier to be global and nonnull, allowing Single to be set by init().

Once the command line has been parsed, the program is preprocessed, translated, and linked with library routines. If this is successful, the screen is cleared (the clear code is terminal dependent) and the program is written at the left of the screen. The program is then executed. Subsequent screen display comes from procedures called by Scan; see Appendix C.

The various files are at site-dependent locations.

```plaintext
procedure main(x)
  local file, in, base, s, switch
  switch := ""
  # procedures used as switches
  every s := |x do
    if s[1] == "-" then switch := "" | |
    case s[2] of |
      "u": "uset.u1"
      "s": "sset.u1"
    default: stop("usage: [-u -s] file")
  |
  else file := s
  # assume it is the program
  if /file then stop("no file specification")
  file ? |
    # parse file name
    while tab(upto("/") + 1)
      (base := tab(find(".icn")) &
       pos(-4)) | stop("illegal file specification")
  |
  if system("Ptran -s " | | file | | |
    " | icont = -s -o " | |
    base | | pscan.u1" | | switch) -= 0
  then stop("translation failed")
  writes("\^[M"]
  in := open(file)
  every 1 to 24 do
    write(trim(left(read(in), 50))) | break
  system(base)
  # execute the program
end
```

Ptran is the preprocessor. The scanning, display, and pattern-matching procedures are contained in pscan.icn. The programs uses.icn and sset.icn are simply

```plaintext
procedure uses_()
end
```

and

```plaintext
procedure sset_()
end
```