(5)

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 517–530 (1982) 1982a

A Pattern Matching System

IAN SOMMERVILLE

Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, Scotland

SUMMARY

This paper describes a pattern matching system which has been implemented as a set of library procedures. The system provides a concise and consistent method of pattern definition and facilities for defining context sensitive pattern matching, defining repetitive patterns and defining alternatives. The operations available to the user allow him to identify if a substring matches a pattern, to extract that substring, to replace that substring and to associate a name with that substring. The system has applications in information retrieval, text manipulation and language processing.

KEY WORDS Pattern matching Context sensitive S-algol

INTRODUCTION

Pattern matching systems have been used in some form or another in context editors, translators, text analysis programs and artificial intelligence research. A pattern matching system consists of a program or set of programs providing facilities for defining patterns which syntactically specify the members of a set of strings. The system also provides associated operations which allow strings to be selected and manipulated if they match the specified pattern—that is if they satisfy the syntactic specification.

The notion of pattern matching is not new. As early as 1961, the language IPL-V[1] included some pattern matching facilities and a number of other languages developed since then have refined the concept. The most widely used pattern matching language is SNOBOL4[2] which has extensive pattern matching facilities, the power of which has prompted a good deal of research into pattern matching at both a practical and a theoretical level.[3–6] The most recent language developments include SL5[7] which expanded the pattern matching facilities of SNOBOL4 and provided good control structures, and ICON[8] which adopts a new approach to pattern matching. ICON does not have explicit patterns but has extended string processing capabilities which allow pattern matching operations to be easily implemented.

Apart from programming languages specifically designed for pattern matching operations, some pattern matching facilities have been provided in languages designed for artificial intelligence research[9] and pattern matchers which use regular expressions to define patterns have been implemented by Aho and others[10] and Richards[11].

The system which we have implemented has advantages over some other systems inasmuch as it allows context sensitive pattern matching and permits patterns to be treated as data objects. This has the advantage that patterns may be input, output, passed as parameters to procedures and returned as function values. The system also

0038–0644/82/060517–14$01.40
© 1982 by John Wiley & Sons, Ltd.

Received 12 September 1980
Revised 12 October 1981

provides a concise and consistent method of pattern definition and the implementation technique chosen involves no overhead whatsoever for the language user who does not need pattern matching.

The pattern matching system is implemented as a set of library procedures which are accessible from a programming language called S-algol[12]. The system was implemented in this way because we needed pattern matching capabilities integrated with our existing software use of a special pattern matching language was impossible.

A major design decision in any pattern matching system concerns the method of defining patterns. SNOBOL uses special built in pattern names and operations which construct patterns and a possible approach is to provide operations like these as library procedures. However, such an implementation makes it difficult to return patterns as procedure results and to provide facilities for the input and output of patterns. It can also become very verbose and introduce many new keywords for the user to learn. This we wished to avoid.

Accordingly, we decided that patterns would be defined as strings with some characters having a special meaning in certain contexts. For example, the pattern:

John#3Smith

matches the character string *John*, followed by any three characters, followed by *Smith*. The pattern:

(?10)%*firstten*

matches the characters in a string up to the 10th character and associates the name '*firstten*' with this substring.

The user of our system defines a pattern as a string and passes that string to a pattern building procedure which returns a pointer to a data structure representing that pattern. The user need not know the form of this structure—the pointer is effectively the 'value' of the pattern. The procedures which perform pattern matching operations all accept this pattern 'value' as a parameter.

In the remainder of this paper we present a 'top-down' description of our pattern matching system. Firstly, the operations available to the system user are described and this is followed by a number of examples illustrating the power of the system. These examples consist of short S-algol programs and illustrate pattern definitions and calls to interface procedures. This is followed by a full description of the pattern definition mechanism. This describes the basic pattern elements and how these elements can be combined to form more complex patterns. We also describe a novel feature of our system which constrains pattern matching to certain contexts and then go on to describe system extensions which allow repetitive patterns to be specified, which allow names to be associated with parts of the pattern and which allow a pattern delimiter rather than an explicit pattern to be declared. The final section summarizes the most important aspects of the pattern matching system and makes a brief assessment of the work we have done.

THE USER INTERFACE

Clearly, the user interface to a pattern matching system is made up of two parts:

(1) The pattern matching operations.
(2) The pattern definition mechanism.

Every pattern match involves applying a *subject* to an *object*. In our system, both the subject and the object are of type string but the subject string is interpreted by the pattern matching system as defining a pattern and the system attempts to find an occurrence of this pattern in the object string. Associated with the object string is an integer pointer called a *cursor* whose value indexes some character in the object string. The cursor has a minimum value of 1 and a maximum value of *rpos* where *rpos* is defined as *length*(*object*) + 1. When the cursor has the value *rpos*, it is said to index the special unprintable character '*end of string*'.

In this section we describe the operations available to the user of the pattern matching system. Following sections provide examples showing how the system may be used and a description of the pattern definition mechanism.

The most important operations which must be provided in any system are:

(1) Simple pattern matching.
(2) Replacement of a matched substring with some other string.
(3) Identification of the position in the object and the length of a matched substring.
(4) A means of accessing the substrings associated with user defined names.

Our pattern matching system provides these operations and some extensions to them by means of a set of library procedures. Because we dislike systems which depend on side effects, most of our library procedures return a structure rather than a single value. In all cases, the first element of that structure indicates the result of the pattern match but the remaining elements depend on the particular operation implemented.

In the notation below, the function name is specified followed by a description of its parameters. The arrow → means 'returns', and this is followed by a description of the structure returned by each function. As S-algol pointers may be associated with any structure, no structure name is necessary to define pointers.

The procedures available to the user are:

build.pattern(*string s*) → *pointer*

This procedure must be called to convert the string representation of the pattern to an internal representation. It parses the string and builds a linked list representing the pattern. The system returns a pointer to this representation but the user need never concern himself with its detailed structure.

match.pattern(*pointer subject*;*string object*) → *bool*

This procedure simply indicates whether a substring matching the subject exists in the object. Notice that the subject should be the pointer returned by the *build.pattern* function and the object should be a simple string—not a pattern. The object string and the subject pattern are compared. If the subject matches the object, *match.pattern* returns '*true*', otherwise '*false*'.

match.pattern.and.assign(*pointer subject*;*string object*) → *bool*,*pointer*

As described later, the user may associate names with parts of a defined pattern using an assignment specifier. If a match succeeds, these names are associated with those substrings in the object string which match the parts of the pattern. This procedure not only determines the result of the pattern match, it also associates the appropriate substrings with the names defined by the user. The result of the match is returned along with a pointer to a structure holding the names and matching substrings. The user need not concern himself with the details of this structure.

lookup(pointer namestructure;string name) → string

This procedure is used to find out what substring is associated with a user defined name. It takes the pointer to the data structure holding the name information returned by *match.pattern.and.assign* and the name defined in the pattern specification. It returns the substring (if any) associated with that name.

lookup.position(pointer namestructure;string name) → integer,integer

This procedure has parameters as above but instead of returning the string associated with the given name, *lookup.position* returns the position of that string in the object and its length.

replace(pointer subject;string object,replacement) → bool,string

The intention of this procedure is to create a new string consisting of the original object but with that part of the object matching the pattern replaced by another, user specified, string. It performs a pattern match operation and returns its result. If the match is successful, it also returns a string where the object substring matching the subject is replaced by the specified replacement string. If the match fails, the string returned is the original object.

extract(pointer subject;string object) →bool,string

This procedure extracts the substring in the object string which matches a specified pattern. It performs a pattern match and returns its result. If the match succeeds it also returns the object substring which matches the subject.

pattern.position(pointer subject;string object) → bool,integer,integer

This procedure is like extract above except that rather than return the matching substring, *pattern.position* returns its position in the object and its length.

A full description of the interface procedures and the structures returned by these procedures is available in the system reference manual.[13]

EXAMPLES

In this section we present a number of example programs illustrating the power of the pattern matching system. The mechanism of these examples is explained by included comments, where any characters following ! should be read as comment.

Example 1—blank replacement

```
! This program reads records from some input stream, replaces all
! sequences of one or more blanks in a record with a single blank
! and writes the resulting record to some output stream
! The pattern blanks is defined as any string of one or
! more blanks.
!   let introduces a name whose type is determined
!   by the type of the right side of the assignment
    let blanks = build.pattern("( )+")
!   process input stream till end of file
    while ~eof do
    {
!               get a line of input
                let rec : = readrecord
                repeat
!                       match pattern and replace strings
!                       of blanks
                        res: = replace(blanks,rec," ")
!               continue matching on same line until no
!               more streams of blanks found
                while res(result) do
                        rec : = res(newstr)
!               write out transformed line
                putrecord(rec)
    }
```

Example 2—selecting strings of a given length

```
!   This procedure reads records from an input stream, selects those
!   of length L and which also have a number at the beginning of the
!   record.
!   The length of the string L is passed as an integer and
!   converted to a string by the procedure int.to.string.
!   This length is then included in the pattern definition by
!   catenating it with the rest of the string defining
!   the pattern. The catenation operator is ++.
!   the character # is used to define patterns of a given length
!   so #n matches strings of n or more characters
!   ^ matches the null string at the beginning of the string
!   and $ the null string at the end so !^#n$! matches
!   strings of exactly n characters. Enclosing the pattern
!   in exclamation marks ensures that the implicit pattern
!   pointer is not moved by the match.
!   the pattern [0-9]+ matches one or more digits so the
!   entire pattern matches strings of length n which start with
!   one or more digits

    procedure get.records(int L)
    begin
                ! make up pattern ^!#L$!(^[0-9])+ by converting L to string
                let P =
                build.pattern("^!#" ++ int.to.string(L) ++ "$!(^[0-9])+")
```

```
        while ~ eof do
        {
                let rec     : = readrecord
                if match.  pattern(P, rec) do
                           putrecord(rec)
        }
end
```

Example 3—finding keywords

```
  !  This program scans records looking for the keyword "'computer"
  !  in column 25. It outputs those records in which computer
  !  is not followed by program or system or application
  !  the pattern ?25 means tab 25 — it moves an implicit pointer
  !  to column 25 in the record
  !  The pattern !~ (program|system|application)! is what is
  !  termed a negative qualifier. It only succeeds if the specified
  !  pattern fails to match. In this case, it succeeds if computer
  !  starting in col 25, is not followed by 'system',
  !  'program', or 'application'
     let keyword =
     build.pattern("?25computer! ~ (program|system|application)!")
     while ~eof do
     {
                let rec    : = readrecord
                if match . pattern(keyword,rec) do
                       putrecord(rec)
     }
```

Example 4—extract and identify tokens

```
  !  This is a simple recognizer which given some string ignores
  !  blanks at the beginning of the string and returns the first token
  !  in the string. Tokens may be real numbers or integers, bracket
  !  characters or arithmetic operators
     procedure scanner(string instr → string)
     begin !                integer is any string of one or more digits
             let integer = "([0—9])+"
             real is integer.integer
             let real = integer++"."++integer
  !          define characters. As (and) are special characters
  !          in the pattern matcher, they are preceded by the
  !          escape character \.
             let char = "(+|−|*|/|\)|\()"
  !          token is a concatenation of real, integer and char.
```

```
!          the first pattern in token ( )* skips over leading
!          blanks, the next pattern recognizes the token and
!          the operator "0 assigns the recognized pattern to
!          nextsym.
           let tok = "( )*(" + +integer + + "|" + +real + + "|" + +char
                         + + ")"0nextsym"
           let token = build.pattern(tok)
!          match pattern and assign matching substring to
!          'nextsym' if pattern matches successfully.
           let next = match.pattern.and.assign(token,instr)
!          if successful match, get the string matched by
!          nextsym and return it as the result of the
!          function scanner. Otherwise return the null string
           if next(matched) then
                  lookup(next(matchlist),nextsym)
           else""
   end
```

BASIC PATTERN ELEMENTS

Patterns are built out of basic pattern elements which individually match substrings in the object pattern. There are six basic pattern elements:

(1) Simple strings– these are simply string of characters such as *fred*, *a;b;c;*, *John Smith*. Blanks are considered part of simple strings. Simple strings are delimited either by the special 'end of string' character defined above or by one of the special characters below. Any of the special characters may be escaped using the character \, in which case the special character is simply considered to be part of the string.

(2) POS—this element is written ^⟨*n*⟩ where ⟨*n*⟩ is some positive or negative integer. If ⟨*n*⟩ is omitted ^1 is assumed by the pattern matching system. If ⟨*n*⟩ is negative, ^(*rpos-n*) is the value assumed by the system. POS matches the null string at the cursor position specified by ⟨*n*⟩. If the cursor is positioned elsewhere in the object, POS will fail to match. Therefore ^ will only match if the cursor is positioned at the beginning of the object, ^10 will match if the cursor is positioned at the tenth character in the object. ^−10 will match if the cursor is positioned 10 characters from the end of the object string and ^−0 will match if the cursor is positioned at *rpos* at the (virtual) 'end of string' character. The special character $ may be used to indicate this position rather than ^−0.

(3) LEN-This element is written #⟨*n*⟩ where ⟨*n*⟩ is some positive integer. It matches any string of characters of length ⟨*n*⟩. Therefore #1 will match any single character, #7 will match any 7 characters. LEN can only fail if there are fewer than the specified number of characters between the current cursor position and *rpos*.

(4) TAB—this element is written ?⟨*n*⟩ where ⟨*n*⟩ is any positive or negative integer. It matches any string from the existing cursor position to the specified position if the existing cursor value is less than ⟨*n*⟩. If ⟨*n*⟩ is negative, ?(*rpos-n*) is taken to be the specified tab. Therefore ?10 matches from the current cursor position to the 10th character, ?−16 matches from the current cursor position to the

16th character from the end of the object string. If the cursor value is greater than ⟨*n*⟩ when ? is encountered the match fails.

(5) ANY this element is written [⟨*string*⟩] and matches any *single* character specified in ⟨*string*⟩. Therefore [*abcde*] will match *a* or *b* or *c* or *d* or *e*. Ranges may also be specified—[*a z*] will match any lower case letter.

(6) NOTANY this element is written {<*string*>} and is the converse of ANY. It will match any single character *not* specified in the string. Again ranges such as {*A–K*} may be specified.

Any basic pattern may be enclosed in round brackets without changing its meaning—for example #8 is equivalent to (#8). Bracketing is essential when pattern specifiers(described below) are used.

The basic elements may be combined to form more complex patterns, may be qualified with another pattern, or may be enhanced using specifiers. We shall discuss each of these in turn.

PATTERN COMBINATION

The basic pattern elements discussed above are the building blocks out of which more complex patterns may be constructed. Patterns may be combined using catenation and alternation.

Catenation of patterns is specified simply by juxtaposition:

^*fred*

means the pattern POS followed by the simple string '*fred*'. In this case, ^*fred* will match objects whose first four characters are '*fred*'.

^*fred*$

means ^followed by '*fred*' followed by^—0. This pattern will match objects consisting of '*fred*' and no other characters.

fred#6{*a–z*}

will match '*fred*' followed by any 6 characters followed by any single character which is not a lower case letter.

Pattern alternatives may be specified by separating alternatives using a vertical bar | and enclosing all alternative patterns in round brackets. For example:

(*john*|*jim*|*jack*)

will match *john* or *jim* or *jack*.

(*fred*#6|*jim*#7|*william*#4)

will match '*fred*' followed by any 6 characters or '*jim*' followed by any 7 characters or '*william*' followed by any 4 characters.

The order of alternative specification is important. The system assumes that it should look for alternatives in the order specified so it scans the whole of the object string looking for the first alternative and only if that fails does it back up and look for the second alternative and so on. Therefore if an attempt was made to match the above pattern with the object:

william brown and fred wilson

the string '*fred wilso*' would match in spite of the fact that an apparently valid match '*william bro*' occurs in the object string before it. In this respect there is a difference between

[*a*–*z*] and (*a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*)

ANY will match the first character in the object string which is a lower case letter. The sequence of alternatives will match the character in the object string which is nearest the beginning of the alphabet.

The alternation facility includes a full backtracking capability so that if the first part of a pattern matches but a subsequent part fails, the system can back up and try alternatives of the first previously successful part in the hope that they will allow subsequent success. For example, suppose we have a subject:

(*br*|*b*)(*eak*|*ranch*)

and apply this to an object

branch

The system will first match '*br*' against the first 2 characters of '*branch*'. It will then attempt to match '*eak*' and then '*ranch*' against the following 4 characters '*anch*'. Failure will ensue which will cause the system to backtrack and match the second alternative of the first part of the pattern '*b*' against '*branch*'. This will succeed and the system will then try all the alternatives in the following part of the pattern. Success will ensue when '*ranch*' is matched against the object.

Similarly suppose we have a subject:

(*a*|*b*)*c*

and an object

bcad

The system will first match '*a*' in the object and then attempt to match '*c*' with '*d*'. This will fail so the pattern matcher will backtrack and match '*b*'. with the object followed by a successful match of '*c*' and '*c*'.

PATTERN QUALIFICATION

A serious deficiency in many existing pattern matching systems is the context independent nature of the pattern matching. If a subject pattern is applied to a sequence of objects, each object which contains a substring matching that pattern is deemed to contain a successful pattern match. The context of the matching substring in the object string is not taken into account by the pattern matching system. It is usually impossible to specify a pattern which will match only in a particular context or in a situation where a particular context does not exist. If context sensitive matching is required, it is usual to select all objects which successfully match some pattern and then apply some further pattern matching operation to them using the appropriate context as the subject pattern.

Our pattern matching system provides a construct for specifying context sensitive patterns by introducing the notion of pattern qualification. Any pattern may be qualified by either a positive or a negative qualifier. Qualifiers are simply pattern specifications enclosed in exclamation marks !!. A negative qualifier is a qualifier where the pattern is preceded by the special character ˜. Pattern qualifiers are not confined to simple patterns. They may include pattern combinations and the full range of additional specifiers described below.

If a qualified pattern is to match, the object string must contain a substring matching the pattern and the qualifier must be satisfied. Satisfying the qualifier means that the qualifying pattern must also match (or not match if the qualifier is negative) the object substring immediately succeeding the matched string. After a successful qualified pattern match, the cursor points at the beginning of the qualifier substring in the object. We illustrate the notion of qualification by example below:

John! *Smith*!

will match the substring '*John*' only if it is immediately followed by '*Smith*'.

John!˜*Smith*!

is an example of a pattern qualified by a negative qualifier. This pattern will match the substring '*John*' only if it is *not* immediately followed by ' *Smith*'.

[*a* *z*]!˜[*a*–*z*]!

will match any single lower case letter on its own. It will not match an object substring where there are two or more lower case letters in sequence.

ˆ !#20!

will match the null string at the beginning of objects whose length is 20 characters or more.

fred! (*smith*|*jones*)!

will match '*fred*' provided it is followed by '*smith*' or '*jones*'.

(*john*|*j.*)! ˜ ?20!

will match '*john*' or '*j.*' provided that ?20 fails that is provided the cursor is beyond character position 20 in the object.

PATTERN SPECIFIERS

As well as pattern combination and pattern qualification, other useful facilities in a pattern matching system are the ability to match up to but not including some specified string, the ability to refer to matching substrings using some user specified name, and the ability to match repetitions of a given pattern. These facilities are provided in our system by using what we call pattern *specifiers*.

A pattern specifier is simply a single character which may follow any bracketed pattern. The specifier characters only have a special meaning when they follow a closing bracket they need not be escaped elsewhere if they are part of a string. There are three types of specifier:

(1) The repetition specifier.
(2) The 'break' specifier.
(3) The assignment specifier.

The repetition specifier allows the user to specify that the pattern is made up of a number of repetitions of the bracketed pattern provided. This may be an explicit number such as 6 or it may be specified one or more repetitions or zero or more repetitions of the given pattern. An explicit number of repetitions is specified simply by following the bracketed pattern by the number. For example:

  *(john|fred)*4

will match 4 repetitions of *john* or *fred* or any combination of *john* and *fred*. The pattern:

  ([*a* *z*])10

will match any character string of length 10 whose components are lower case letters.

One or more repetitions of the pattern is specified by following the bracketed pattern with a plus sign(+) and zero or more repetitions by following the bracketed pattern with a star(*). For example:

  (*a*)+

will match *a*, *aa*, *aaa*, *aaaa*, *aaaaa*, etc —any sequence of one or more *a*s.

  (*a*)*

will match the null string or any sequence of one or more *a*s. The system always attempts to match as many repetitions as possible.

The 'break' specifier is derived from the BREAK function is SNOBOL4 but is more general. Rather than simply a single character, the 'break' specifier allows a delimiter pattern to be specified. The system will match any string, including the null string up to but not including the specified pattern. Break patterns are specified by following a bracketed pattern with an at(@) character. For example:

  (fred|joe)@

will match any string up to '*fred*' or '*joe*'.

  ([*a*–*z*])@

will match any string up to a lower case letter.

The assignment specifier allows the user to associate a name with a pattern. The substring which matches the specified pattern is associated with that name. A name may be any string of letters and it is a read-only name local to the pattern in which the name is used. Therefore the same name may be used in different pattern definitions. Assignment specifiers are written by following a bracketed pattern by the character % and following % with the required name. For example:

  (#10)%*lenten*

would associate the name *lenten* with the character string matched by #10.

  (anne|margaret|jane|helen)%*girlsname*

would associate the name *girlsname* with whichever of the alternatives matched. Name assignment is only valid if the pattern match succeeds. If the match fails, the effect of name assignment is undefined.

Naturally, more than one pattern specifier may be used, the only proviso being that the specifier character must immediately follow a closing bracket. Specifiers are evaluated in left to right order and if an assignment specifier is used, it must be the rightmost specifier. For example:

((*a*)10)%*astring*

matches a sequence of 10 *a*s and associates the name *astring* with that sequence.

(((*a*)10)@)%*toastring*

matches any string up to a sequence of 10 *a*s and associates the name '*toastring*' with the matched string.

The combination of these features means that a powerful pattern matching mechanism is available to the user. This can be illustrated with some further examples:

Example 1—pattern to match trailing blanks in a string

()*$

This pattern uses the repetition specifier to find a run of blanks and the $ pattern to ensure that these are at the end of the object string.

Example 2—pattern to match fixed fields of a record and assign names to these fields

(?10)%*f*1(?20)%*f*2(?35)%*f*3(?52)%*f*4

This pattern uses the tab basic pattern to break out the fields and the assignment specifier to name them. It will cause the name *f*1 to be assigned to the first 10 characters, *f*2 to the characters from 11 to 20 and so on.

Example 3—pattern to match all occurrences of '*computer*' **provided it is not followed by** '*science*' **or** '*systems*'

computer!~(*science*|*systems*)!

This pattern uses pattern qualification. It first matches all strings containing computer and then attempts to match '*science*' or '*systems*'. If this match succeeds, the whole match is deemed to have failed.

Example 4—match strings which start with one or more digits and which end with one or more '+' characters. Associate the name '*middle*' **with those characters between the last digit and the first '+'.**

^([0-9)+((+)@)%*middle*(+)+$

This pattern uses the repetition specifier to identify the first digits and final + characters. It uses the break specifier to establish those characters between the last digit and + and the assignment specifier to associate them with *middle*. The special character + only has a special meaning after ')' so it may be used without the escape character in the pattern definition.

CONCLUSIONS

A recent paper by Griswold[14] identifies the advantages and disadvantages of pattern matching as implemented in SNOBOL4. He argues that the complexity of pattern matching systems is fundamentally due to the fact that they are composed of 2 distinct languages—a basic language *L*, and a pattern matching language *P*. The basic language *L* provides control functions, declarations and program structuring facilities whereas *P* provides pattern definition and matching operations. These 2 languages are quite different and there is little facility for communication between them. In developing ICON, Griswold has tackled this problem by eliminating the pattern component *P* and extending the language component *L*.

Although the design of a special language is perhaps the most elegant solution to problems posed by the complexity of pattern matching systems, such a task is a major development effort. We required a pattern matching system which could be integrated with our existing software, and this precluded special language development or use of a language such as SNOBOL4. The approach which we have taken to the development of such a system is to recognize that there is a dichotomy between the basic language and the pattern matching language and to implement the pattern matching component via a library of procedures accessible from the basic language. This means that pattern matching is available to all users without any overhead for those users who choose not to use the system.

Procedural interfaces are never wholly satisfactory but we believe the convenience of pattern matching operations embedded in an existing programming language outweighs the slight clumsiness which is always a feature of systems based on procedure calls. An advantage of a procedural system compared with a special pattern matching language is that sophisticated users may extend the system by adding new pattern matching operations. The system facilities are open ended rather than fossilized in a language definition.

One of the disadvantages identified by Griswold in existing pattern matching systems is the necessity for using side effects and the difficulties this imposes on program structuring. In SNOBOL4 for example, a pattern matching operation sets a global variable indicating the success or otherwise of the match. It may also assign strings to global names if a value assignment operator is specified as part of a pattern definition. We agree with Griswold that side effects are undesirable so one of our fundamental design aims was that the system should never use side effects. A consequence of this is that a number of our pattern matching functions return structures containing values of different types rather than a single value.

The pattern definition technique relies heavily on the string handling facilities provided by S-algol but a similar technique could be used in any language where dynamic strings can be implemented. Defining patterns as strings means that they can be passed as parameters, returned as results, compared, input and output without incurring any extra overhead.

In summary therefore, the advantages of the pattern matching system which we have developed are as follows:

(1) It provides context sensitive pattern matching capabilities.
(2) It allows patterns to be treated as data objects.
(3) It provides a concise and elegant pattern definition mechanism.
(4) It can be integrated with existing software systems.

Our relatively limited experience of the system in use indicates that it will be a useful tool in the development of information retrieval systems, in language processing and in text manipulation.

REFERENCES

1. A. Newell (ed), *Information Processing Language-V Manual (Rand Corp)*, Prentice-Hall, New Jersey, 1961.
2. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, New Jersey, 1971.
3. J. F. Gimpel, 'A theory of discrete patterns and their implementation in SNOBOL4', *Comm. ACM*, **16**, 91–100 (1973).
4. J. F. Gimpel, 'Nonlinear pattern theory', *Acta Informatica*, **4**, 91–100 (1973).
5. R. B. K. Dewar and A. P. McCann, 'MACRO-SPITBOL– a SNOBOL4 compiler' *Softw. pract. exp.*, 7, 95–114 (1977).
6. R. E. Griswold, 'Extensible pattern matching in SNOBOL4', *Proc.ACM Annual Conf.*, 248–252 (1975).
7. R. E. Griswold and D. R. Hansen, 'An Overview of SL5', *SIGPLAN Notices*, **12**, (5) 40–50 (1977)
8. R. E. Griswold, D. R. Hansen, J. T. Korb, 'The Icon programming language: an overview', *SIGPLAN Notices*, **14** (4), 18–31 (1979).
9. D. G. Bobrow and B. Raphael, 'New programming languages for artificial intelligence research', *Comput. Surv.*, **6**, 153–174 (1974).
10. A. V. Aho, B. W. Kernigan and P. J. Weinberger, 'Awk- a Pattern scanning and processing language', *Softw, pract. exp.*, **9**, 267–279 (1979).
11. M. Richards, 'A compact function for regular expression pattern matching', *Softw, pract. exp.*, **9**, 527–534 (1979).
12. R. Morrison, *S-Algol Reference Manual*, Dept of Computational Sci, University of St Andrews, Scotland, June 1980.
13. I. Sommerville, *Pattern Matching in S-Algol*, Dept. of Computer Science, University of Strathclyde, Scotland, June 1980.
14. R. E. Griswold and D. R. Hansen, 'An alternative to the use of patterns in string processing', *ACM Trans. Prog. Lang. and Systems*, **2**, 153–172 (1980).