stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 126/79 DECEMBER

P. KLINT

AN OVERVIEW OF THE SUMMER PROGRAMMING LANGUAGE

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

1980 Mathematics subject classification: 68B05, 68G10

ACM-Computing Reviews-categories: 3.63, 4.22

An Overview of the SUMMER Programming Language *

by

Paul Klint

ABSTRACT

The language SUMMER is intended for the solution of problems in text processing and string manipulation. The language consists of a small kernel which supports success-directed evaluation, control structures, recovery caches and a data abstraction mechanism. It is shown how this kernel can be extended to support simultaneous pattern matching in arbitrary domains.

KEY WORDS & PHRASES: string manipulation, generalized pattern matching, recovery caches, success-directed evaluation

---

* This report is intended for publication elsewhere.

1. INTRODUCTION

The language SUMMER has been designed for the solution of problems in text processing and string manipulation. SUMMER consists of a relatively small kernel which has been extended in several directions. The kernel supports:

integers
reals
strings
classes
files
procedure and operator definitions
success-directed evaluation
control structures
recovery caches

and has been extended with

arrays (sequences of values)
tables (associative memories)
pattern matching
string synthesis

Pattern matching has been completely integrated with the success-directed expression evaluation mechanism. It will be shown that the operations in the kernel are sufficient to allow generalization of pattern matching in two directions:

- Simultaneous pattern matches can be expressed, which mutually affect each other.
- Pattern matching needs no longer be restricted to the string domain.

An attempt is made to describe most (novel) features of SUMMER and motivate their inclusion in the language. A simplified version of the pattern matching extension is discussed in some detail. Sections are included on related work and implementational issues.

2. SUCCESS-DIRECTED EVALUATION AND CONTROL STRUCTURES

The expression evaluation mechanism of SUMMER is somewhat unusual and needs special attention. Expressions consist of a juxtaposition of operators (like addition: "+" or string concatenation: "||") and operands (like the numeric constant "1Ø", the string constant "'abc'", the identifier "x" or the procedure call "p(1Ø,x)"). Some operations can only deliver a value, but others can potentially fail. If an expression fails, the evaluation of the expression in which that operation occurs is aborted immediately and failure is signalled to the construct in which the failing expression occurs. Such a failure is a transient entity and must be captured at the moment it occurs. Three cases arise:

a. The syntactically enclosing construct is capable of handling the failure itself. This is the case if the failing expression "E" occurs in contexts like:

```
if E then ... else ... fi
while E do ... od
E | ...      (logical "or" operator)
```

b. The syntactically enclosing construct is not capable of handling the failure itself, but is (perhaps dynamically) enclosed in a construct with that capability, like:

```
E & ...      (logical "and" operator)
return(E)    (value from a procedure)
```

In this way failure can be passed to the caller of the procedure in which the failing expression occurs (see below).

c. Neither of the above two cases applies. This results in abnormal program termination with the error message "Undetected failure". In

```
x := read(input); print(x);
```

the call to the read procedure may fail (on end of file). This failure will not be detected by the program itself and hence execution of the program will be aborted.

This expression evaluation scheme was designed to be concise and powerful, but at the same time an attempt was made to protect the programmer against undetected or unwanted failure.

Conciseness is obtained in two ways. First, by computing a value and a failure signal in the same expression. This allows, for example

```
while line := read(input) do ... od
```

instead of

```
while not eof(input)
do line := read(input);
   if io_errors(input) then ... fi;
   ...
od;
```

Second, by disregarding the source of failure and focussing attention on the absence of failure (i.e. success) during the evaluation of the expression. Consider:

```
if (read(input) || read(input) ≠ expected
then
   error('Bad input')
fi
```

where "expected" has the expected input string as value. Three sources of failure can be identified here: the two read operations and the inequality test. The programmer, however, is in most cases only interested in the fact that the input file does not conform to his expectations. The above formulation makes this more clear than

```
L1 := read(input);
if eof(input) then error('Bad input')
else
   L2 := read(input);
   if eof(input) | (L1 || L2 ≠ expected)
   then
      error('Bad input')
   fi
fi
```

In principle, this argument works in two directions: since the source of failure may be lost, the programmer may be misled about the actual source of failure. It is our experience that this seldom happens and in all cases where the distinction is important it can be expressed easily.

Protection is achieved by prohibiting undetected failure. This turns out to be a frequent source of run-time errors, which always corresponds to "forgotten" or "impossible" failure conditions. A direct consequence of this protection scheme is that one can write assertions (i.e. expressions which should never fail) in a program. A run-time error occurs if such an assertion is violated.

Another noteworthy consequence of this evaluation mechanism is its ability to let a procedure report failure to any procedure which called it (in)directly. This effect is obtained by adhering to the programming convention that procedures have the form E_1 & ... & E_n. If one of the expressions E_i fails, this failure is passed to the caller of the current procedure. If that calling procedure has the same form, it will not handle the failure itself but will pass it on to its caller. In this way, low-level procedures need not be aware of failure at all and high-level procedures can detect the failure and take appropriate measures. Some programming languages have special facilities for handling exceptions of this kind; in SUMMER they can be handled by the standard expression evaluation mechanism.

3. RECOVERY CACHES

For the solution of problems such as parsing languages with context-sensitive or non-LL(1) grammars and heuristic searching, it is often necessary to attempt a potential solution and to undo the effects of that attempt if it is not successful. Many schemes have been proposed for the formulation of such backtracking algorithms, but most involve either opaque control structures or provide unsatisfactory control over modifications of the program environment (i.e. global variables).

The recovery cache [1], which was invented to increase software reliability has been adapted to act as a device for monitoring environment modifications in backtrack-liable situations. Recovery caches are used both at the conceptual and at the implementational level. A cache consists of (name, value) pairs. The name part may refer to simple variables, array elements and class components (see section 4). When backtracking may be necessary a new cache is created and from that moment on all assignments to variables and input/output operations are monitored. Whenever an assignment is about to be made to a variable whose name does not yet occur in the cache, its name and value before the assignment are entered in the cache. Modifications of input/output streams are registered similarly. If the attempt is successful and no backtracking is necessary, the information in the cache is discarded but in case of failure, the information in the cache is used to restore the environment to the state as it was at the moment that the cache was created. Since recovery caches may be nested, "discarding" may mean: merging the information in the current cache with that in the previous cache. In this manner, the information in the previous cache is still sufficient to describe all modifications which were made since that cache was created. There are two exceptions to these rules:

- Input/output operations on the standard input/output stream are not recovered. In many situations it is not desired to recover these streams and in some cases the meaning of such a recovery may be non-obvious or confusing. In SUMMER these streams can be used to control and monitor the backtracking process interactively.

- The local variables of the procedure in which the cache was created are not recovered. In this way information about the reason of failure can survive the failure itself.

At the programming language level, caches are introduced by the construct

try E_1, E_2, ..., E_n until $E_\emptyset$ endtry .

In a first approximation this expression is equivalent to

$(E_1$ & $E_\emptyset)$ | ... | $(E_n$ & $E_\emptyset)$

Before the evaluation of each $(E_i$ & $E_\emptyset)$ starts, a new cache is created. If the evaluation of this subexpression succeeds, the cache is discarded and the whole expression succeeds. If the evaluation fails, the environment is restored from the cache and evaluation of $(E_{i+1}$ & $E_\emptyset)$ is attempted in the same manner. The whole expression fails if none of the subexpressions succeeds. Completely automatic backtracking is achieved by nested try constructs. This simple scheme is very well suited for the formulation of problems occurring in pattern matching as will be seen in section 5.

4. PROCEDURES, OPERATORS AND CLASSES

The remaining features of the SUMMER kernel are now summarized. Procedures have a fixed number of parameters, which are passed by value. Procedures may either fail or return zero or more values. Hence it is possible to return more then one result value.

An operator is defined by associating a user-defined operator symbol with a procedure with one or two parameters.

Classes are the only available data structuring mechanism and are a generalization of the SIMULA [2] class. A class declaration consists of:

- A class name and formal parameters. The class name is used as name for the creation procedure for objects belonging to this class. The formal parameters are used to provide initial values for that object.
- Fields, which are either used to store information related to the object (e.g. the real and imaginary parts of a "complex number" class object),

or information local to the class object (the stack pointer in a "stack" class object). Fetch and store access to fields can be controlled completely by associating fetch and store procedures with each field.

- Access procedures and operators defining the operations that can be performed on objects of this class.

The components of a class are accessed by means of the "dot" notation. The operators which are defined in a class can be used in infix notation. The type of the left operand of an operator is used to disambiguate overloaded operators, i.e. operators which are defined in more than one class.

One, final, concept must be introduced before we can turn our attention to some pattern matching applications. One of the advantages of string pattern matching languages is that they liberate programmers from the necessity to repeat a current subject string and cursor position in each pattern matching operation. In SUMMER an attempt is made to provide such a facility in general.

In sequences of the form:

```
a := S.x; b := S.y; c := S.z(10)
```

the prefix "S." could be factored out. Pascal uses the construct

```
with S do begin ... end
```

for this purpose. All field references that occur inside begin ... end are automatically prefixed with "S.". In this notation the example would read:

```
with S do begin a := x; b := y; c := z(10) end
```

But this is not sufficient for the applications we have in mind, where it is not unusual that many procedures operate on the same class object. This is illustrated by a set of parsing procedures that operate on one subject string. The Pascal approach has the disadvantage that this common class object must be passed as argument to all procedures (or must be assigned to a global variable) and that all procedure bodies must be surrounded by a with construct. This problem can be circumvented as follows. The construct (1)

---

(1) Inspired by the "scan S using E" construct in Icon [3].

scan S for E rof

declares a completely new variable each time the construct is encountered at run-time and assigns the class object S to that new variable. All occurrences of fields from the class to which S belongs are now prefixed with this new global variable in the same way as is done in Pascal. The scan construct is more general since it affects all expressions and procedures which can be evaluated directly or indirectly from the body of the scan construct. In Pascal this effect is restricted to the expressions which are statically enclosed in the body of the with construct. If the scan construct is used in a nested fashion, then the previous value of the new global variable is saved and restored properly on exit from the current scan construct. This also applies to the case that the scan construct is left prematurely by means of a return statement.

5. A PATTERN MATCHING EXTENSION

5.1. String Pattern Matching

Now we will show how a string pattern matching system can be build on top of the SUMMER kernel. Pattern matching is done on a string subject which is indexed by an integer cursor. For the sake of this discussion a very simple system will be defined, which only supports the following three functions:

lit(S): literally recognize the string S. If S occurs as substring in the subject at the current cursor position, then deliver S as value and move the cursor beyond S. Otherwise report failure.

break(S): recognize a string of characters not occurring in S followed by one terminating character which does occur in S. If such a string can be found starting at the current cursor position then deliver that string (without the terminating character) as value and move the cursor to the terminating character. Otherwise report failure.

span(S): recognize a non-empty string of characters all of which must occur in S. If such a string occurs as substring in the subject at the current cursor position, then deliver that string as value and move the cursor

beyond it. Otherwise report failure. (Span is added to allow more interesting examples, its implementation will not be shown here.)

The following class definition implements this pattern matcher:

```
class scan_string(subject)
begin var cursor := Ø;

proc lit(s)
( if cursor + size(s) > size(subject) |
     s ≠ substr(subject,cursor,cursor+size(s))
  then
     freturn          # failure return #
  else
     cursor := cursor + size(s);
     return(s)
  fi
);
proc break(s)
( var newcursor := cursor, result;
  for newcursor in [cursor : size(subject)]
  do
     for c in s do
       if c = subject[newcursor] then
          result := substr(subject,cursor,newcursor);
          cursor := newcursor;
          return(result)
       fi
     od
  od;
  freturn
);

proc span(s)    (# similar to break #);

end class scan_string;
```

The following example illustrates how identifiers starting with the letter "X" can be recognized:

```
proc X_identifier(s)
( var t := scan_string(s);
  t.lit('X') & (t.span(letgit) | t.lit(''))
)
```

(In all examples we assume that "letter", "digit" and "letgit" have appropriate values.) Note that the normal logical operators "&" and "|" are used for combination. Hence there will be no backtracking, reversal of effects or whatsoever.

This example can be written in a more concise form if we use the scan construct:

```
proc X_identifier(s)
 scan scan_string(s)
 for
    lit('X') & (span(letgit) | lit(''))
 rof
```

A final example may illustrate the use of the value delivered by the pattern matching procedures. The problem is to extract all letters from a given string. For example "a,b,c" gives "abc":

```
proc extract_letter(s)
( var result := '';
  scan scan_string(s) for
    while break(letter) & (result := result || span(letter))
    do # empty statement # od
  rof;
  return(result)
)
```

In SUMMER pattern matching and backtracking have been separated completely. It came as a shock to us that the vast majority of pattern matching problems, we had previously solved by means of implicit backtracking, could be solved without any backtracking at all! This suggests that the close interaction between pattern matching and backtracking, as can be found in many languages, should be reconsidered.

Now we will address the question how pattern matching with automatic backtracking can be obtained. Consider the expression:

```
(lit('ab') | lit('a')) & lit('bc')
```

In the pattern matcher developed above, the alternative lit('a') is discarded as soon as a subject string starting with "ab" is encountered. The string "abc" can not be recognized in this way. But if we rewrite this expression as

```
try lit('ab'), lit('a')
until
    lit('bc')
endtry
```

then the recovery cache mechanism restores the initial cursor value automatically and tries the second alternative if lit('bc') fails. No special attention needs to be given to the cursor: it is an ordinary variable which is saved and restored by the recovery cache mechanism!

5.2. Generalized Pattern Matching

In most pattern matching systems there is only one subject string involved in the pattern match. This restriction can be removed without introducing any new concepts as an example will show. The following (rather artificial) problem is to ensure that two strings S1 and S2 conform to the following rules:

a. S1 is of the form $c_1;c_2;\ldots;c_n;$ where c_i is a (perhaps empty) sequence of arbitrary characters other than the character ';'. Some examples are: 'a;b;', '2l!;7a;' and 'ab;cde;f;'.

b. For a given S1, S2 has the form $d_1d_2\ldots d_n$, and either $d_i = c_i$ or d_i = reverse(c_i) holds. Acceptable values for S2 with S1 equal to 'ab;cde;f;' are 'abcdef', 'abedcf', 'bacdef' and 'baedcf'.

The following program performs this check:

```
sl := scan_string(S1);
s2 := scan_string(S2);
scan sl for
   while (c := break(';')) & lit(';')
   do
      if not scan s2 for lit(c) | lit(reverse(c))
                     rof
      then
            error('check fails')
      fi
   od
rof;
if sl.cursor ≠ size(S1) | s2.cursor ≠ size(S2)
then
   error('check fails')
fi
```

Each scan_string object maintains its own cursor. Note how the cursor value of s2 survives each evaluation of the innermost scan construct. This allows the innermost pattern match to continue where it left of the previous time.

From the preceding paragraphs it will be clear that pattern matching as presented here, does not depend on the fact that strings are used as the basic unit of recognition. One can, for example, easily imagine pattern matching in an array of strings. The "cursor" must then be replaced by a pair of values to maintain the current position and basic scanning procedures like xlit, ylit, xspan and yspan must be defined. It may be expected that a system for the recognition of two-dimensional line-drawings, like ESP^3 [4], can be defined in a straightforward manner using the primitives from the SUMMER kernel.

6. IMPLEMENTATION

An implementation of SUMMER is near completion and runs under the UNIX (1) operating system [5]. This implementation consists of a two pass compiler (written in SUMMER) which transforms source programs into a rather high-level abstract machine code. This abstract machine code is then executed by an interpreter written in C [6]. Extensive facilities are provided for program profiling and symbolic debugging.

7. RELATED WORK

SUMMER is the successor of SPRING [7], a language which had the same design goals, but lacked the simplicity and generality achieved in SUMMER. Both languages were inspired by and profited from ideas in SNOBOL4 [8] and SL5 [9]. SUMMER was also influenced by Icon [3]. We had already formulated several ideas for the integration of pattern matching and expression evaluation, but the solution finally adopted in SUMMER was influenced by Icon. There are important differences too. For example, in Icon most pattern matching procedures deliver integer values corresponding to the position to which they can move the cursor. Next the cursor has to be moved explicitly. This operation delivers the substring between successive cursor positions as value. In SUMMER all pattern matching procedures deliver the recognized substring as value and move the cursor. In this way the cursor needs hardly ever be manipulated by the programmer. The pattern matching model in SUMMER is more general, since it allows simultaneous pattern matches and pattern matching in domains other than strings.

The evaluation model which prohibits undetected failure, the use of recovery caches and the separation of pattern matching and backtracking are new.

---

(1) UNIX is a Trademark of Bell Laboratories.

ACKNOWLEDGEMENT

Design and implementation of SUMMER were realized in close cooperation with Marleen Sint.

REFERENCES

[1] Randell, B., System structure for software fault tolerance, in Proceedings of an International conference on reliable software, SIGPLAN notices, 10(1975)6, 437-449.

[2] Dahl, O-J, Myhrhaug, B. & Nygaard, K., SIMULA Information, Common Base Language, Norwegian Computing Centre, S-22, 1970.

[3] Griswold, R.E. & Hanson, D.R., Reference Manual for the Icon Programming Language, TR 79-1, The University of Arizona, Tucson, Arizona, 1979.

[4] Shapiro, L.G., ESP^3: a language for the generation, recognition and manipulation of line drawings, (thesis), TR 74-04, University of Iowa, 1974.

[5] Ritchie, D.M. & Thompson, K., The UNIX time-sharing system, Communications of the ACM, 17(1974)7, 365-375.

[6] Kernighan, B.R. & Ritchie, D.M., The C Programming Language, Prentice-Hall, 1978.

[7] Klint, P., Pattern Matching in SPRING, in Van Vliet, J.C. (ed), Colloquium Capita Datastructuren, MC syllabus 37, 1978, 65-83.

[8] Griswold, R.E., Poage, J.F. & Polonsky, I.P., The SNOBOL4 programming language, Second edition, Prentice-Hall, Englewood Cliffs, N.J., 1971.

[9] Griswold, R.E. & Hanson, D.R., An overview of the SL5 programming language, SL5 project document S5LD1b, The University of Arizona, Tucson, Arizona, October 9, 1976.