CONFERENCE RECORD

SEVENTH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS, AND COMPUTERS

EDITED BY

Sydney R. Parker
Naval Postgraduate School
Monterey, California

PAPERS PRESENTED

TUESDAY THROUGH THURSDAY   NOVEMBER 27-29, 1973
ASILOMAR HOTEL & CONFERENCE GROUNDS – PACIFIC GROVE, CALIFORNIA

Sponsored by

NAVAL POSTGRADUATE SCHOOL   UNIVERSITY OF SAN i CLARA
Monterey, California   Santa Clara, California

With The Participation Of The

UNIVERSITY OF CALIFORNIA   UNIVERSITY OF CALIFORNIA   UNIVERSITY OF CALIFORNIA
Davis, California   Los Angeles, California   Santa Barbara, California

CALIFORNIA STATE UNIVERSITY   STANFORD UNIVERSITY   UNIVERSITY OF SOUTHERN CALIFORNIA
Sacramento, California   Stanford, California   Los Angeles, California

UNIVERSITY OF ARIZONA   WASHINGTON STATE UNIVERSITY
Tucson, Arizona   Pullman, Washington

In Cooperation With

IEEE SAN FRANCISCO SECTION, IEEE CIRCUITS AND SYSTEMS SOCIETY, AND
IEEE CONTROL SYSTEM SOCIETY

CONFERENCE COMMITTEE

Chairman:	J. Choma, Jr., Hewlett-Packard Company
Technical Program:	T.E. Fanshier, Lockheed Missiles and Space Company
	T.J. Higgins, University of Wisconsin
	L.P. Huelsman, University of Arizona
	L.P. McNamee, University of California, Los Angeles
	T. Mills, National Semiconductor Corporation
	J.G. Simes, California State University, Sacramento
	A. Willson, University of California, Los Angeles
Publication/Finance:	S.R. Parker, Naval Postgraduate School
Publicity:	S.G. Chan, Naval Postgraduate School

CONFERENCE RECORD

SEVENTH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS, AND COMPUTERS

EDITED BY

Sydney R. Parker
Naval Postgraduate School
Monterey, California

PAPERS PRESENTED

TUESDAY THROUGH THURSDAY NOVEMBER 27-29, 1973
ASILOMAR HOTEL & CONFERENCE GROUNDS — PACIFIC GROVE, CALIFORNIA

Sponsored by

NAVAL POSTGRADUATE SCHOOL UNIVERSITY OF SANTA CLARA
Monterey, California Santa Clara, California

With The Participation Of The

UNIVERSITY OF CALIFORNIA UNIVERSITY OF CALIFORNIA UNIVERSITY OF CALIFORNIA
Davis, California Los Angeles, California Santa Barbara, California

CALIFORNIA STATE UNIVERSITY STANFORD UNIVERSITY UNIVERSITY OF SOUTHERN CALIFORNIA
Sacramento, California Stanford, California Los Angeles, California

UNIVERSITY OF ARIZONA WASHINGTON STATE UNIVERSITY
Tucson, Arizona Pullman, Washington

In Cooperation With

IEEE SAN FRANCISCO SECTION, IEEE CIRCUITS AND SYSTEMS SOCIETY, AND
IEEE CONTROL SYSTEM SOCIETY

CONFERENCE COMMITTEE

Chairman: J. Choma, Jr., Hewlett-Packard Company
Technical Program: T.E. Fanshier, Lockheed Missiles and Space Company
T.J. Higgins, University of Wisconsin
L.P. Huelsman, University of Arizona
L.P. McNamee, University of California, Los Angeles
T. Mills, National Semiconductor Corporation
J.G. Simes, California State University, Sacramento
A. Willson, University of California, Los Angeles
Publication/Finance: S.R. Parker, Naval Postgraduate School
Publicity: S.G. Chan, Naval Postgraduate School

PATTERN MATCHING FOR STRUCTURED PROGRAMMING

IN PL/I

A.S. Pfeffer* and A.L. Furtado
Computer Science Department
Pontificia Universidade Catolica do Rio de Janeiro
and * Equipamentos Eletronicos, Rio de Janeiro

Abstract

A programming module for adding a simple string pattern-matching capability   to
PL/I is presented.
It is argued that pattern-matching is compatible with control structures that con-
form to the principles of structured programming [1].

1. AN OVERVIEW OF P/PL/I

P/PL/I is an extension to PL/I that adds to this
language a string pattern-matching capability.
Patterns are provided as an additional data type.
Three categories of patterns may be distinguished:

a. lexical patterns-these correspond to "prim-
itive" classes such as: letters, digits,
blanks, special characters, integers, char-
acter string constants (between single
quotes), identifiers;

b. syntactical patterns-classes defined by
concatenation and disjunction of other
classes, variables, and constants;

c. semantic patterns-classes defined by means
of predicates, i.e. procedures returning
'0'B or '1'B.

For lexical patterns the pattern-matching process
is very simple (implemented essentially as   a fi-
nite state automaton) [2]. The concept of semantic
patterns appears in [3].

The definition of a pattern may involve the pat -
tern concatenation (.), disjunction (|),assignment
(underscore), and restriction (::) operators.
Unless brackets are used concatenation has a higher
priority than disjunction. Assignment has the same
meaning as immediate assignment in SNOBOL 4, and
restriction is used to introduce a predicate as an
additional condition for membership in a given
class.
Right recursion is supported in the definition of
patterns as well as repetition factors.
Some examples of patterns are given below:

    <P> ::= 'A' | 'B' | 'C'

    <Q> ::= ',' . <# ID>
    <R> ::= 'D' | 'E' . <R>
    <S> ::= (3:*) P
    <T> ::= <P> Z
    <U> ::= <T>.'+'.Z
    <V> ::= <T> :: F(Z)

Clearly the notation resembles an extend BNF [4,2].
In the definition of <S> the repetition factor in -
dicates 3 or more occurrences of <P>. If the lower
bound were zero then <S> would also allow for the
non-occurrence of <P>.
In the last two patterns note that:

a. the "+" must be followed by exactly the same
substring that precedes it; so "B+B" belongs
to <U>,but "B + A" does not:

b. <V> is the same as <T> restricted by (i.e.,
satisfying in addition the requirements of)
the predicate F; in this example the value
of Z that appears as the argument of F would
have been determined when matching <T> (see
the definition of <T>).

The pattern-matching and replacement process can
now be described by:

    A // <P> → W(X)/ B

which reads: "if the contents of a string variable
A satisfy the requirements of pattern <P> then re-
port success and replace the contents of string
variable B by the value of the PL/I string expres-
sion W(X)".

In fact this is what happens when the total mode is
enabled; in the anchored and unanchored modes, de-
fined as in SNOBOL 4 [5,6], if a sub-string of   A

satisfies <P> then the match succeeds and the val-
ue of B becomes the value of A with the matching
substring replaced by W(X). We have written   W(X)
to stress the fact that assignments may have   oc-
curred during the matching phase to string   varia-
bles that may be present in the PL/I string expres-
sion.
The replacement action may be omitted. In this
case we are only concerned with the success   or
failure of the match (and possibly assignments spe-
cified in the patterns).
A special variable #S is set to '0'B or '1'B de-
pending on the outcome of the matching phase.
Among the other special variables the cursor   var-
iables are particularly important. They   indicate
the position of the character in A that is under
examination at a given moment in the matching
phase.

2. SOME CHARACTERISTICS OF THE IMPLEMENTATION

The implementation uses the standard PL/I pre-proc-
essor to convert P/PL/I into valid PL/I.
The two main statements are the pattern definition
statement and the pattern-matching statement. The
former is shown for the patterns <T> and <R>   of
the preceding section:

   #PAT(<T>) ::= <P>_Z);

   #PAT(#RECURS(<R>::= 'D' | 'E' . <R>));

#PAT is declared as a pre-processor function, and
so is #MAT below, which is used for the pattern-
matching statement:

   #MAT(M//<T> + ','|| Z|| ',' //N);

The replacement part may be omitted:

   #MAT(M // <T>);

In either case the cursor variable for this state-
ment will be reset to 1 before the statement is
executed. However by writing:
   #MAT(M, #// <T>);

the cursor variable will keeps its previous value
which will indicate the position of the first char-
acter in M to be examined. Also the user has   ex-
plicit control over the process; if one writes:

   #M @ = 5;

   #MAT(M, #//<T>);

the fifth character of M will be the starting point
for the matching. The final value of the cursor
variable is 1 plus the lenth of the string if the
matching failed, or 1 plus the position of the last
character in the string that matched the pattern
if the matching succeded.
The pre-processor translates the #MAT statement
into a call to a sub-routine. A pattern-matching
expression that returns '0'B or '1'B (besides set-
ting #S), and can be used in any context where a
bit - valued expression is expected, can be   spec-
ified by using #MAT#. The pre-processor   will

translate #MAT# into a function call.
In the present implementation all variables appear-
ing in a pattern-matching statement must be   de-
clared in a #DCL statement.
Besides handling P/PL/I statements, the pre-proces-
sor program appends to the generated PL/I text   a
number of run-time routines that will perform the
pattern-matching process. It also introduces   a
binary tree [7] where the structure of the patterns
will be represented, and a number of auxiliary ta-
bles (for a graph representation of patterns see
[8]).
As one could expect, if pattern <A> appears in the
definition of pattern <B>, then the tree   corre -
sponding to <A> is a sub-tree of the tree   repre-
senting <B>.
Edges in the tree represent either a concatenation,
being traversed as long as the matching succeeds,
or disjunction, being taken in case of failure.The
matching algorithm takes care of backtracking and
right recursion (left recursion is not supported
in the present implementation).
The auxiliary tables supply additional information
including address and length of string variables
involved, bounds of the repetition factors, entry
points of user-defined predicates, etc.
The pre-processor program and the run-time routines
and tables constitute the P/PL/I module. The mod-
ule runs in an IBM 370/165 under a supervisor.
Other modules (for graph-processing [9], list
processing, and additional control statements) are
also implemented, and through the supervisor sev-
eral modules can be appended to the same PL/I pro-
gram.

A simple P/PL/I program is given below. A   number
of cards containing one or more PL/I-like   state-
ments are read in. The purpose of the program   is
to detect all 'simple arithmetic statements' that
assign values to any variable included in a   given
list, and print out the whole program with gener-
ated comments indicating the variables to which
such an assignment is being made. Here a   simple
arithmetic statement consists of a single scalar
variable followed by an equal sign followed by an
arithmetic expression containing only additions
and subtractions of variables and integer constants;
there are no blanks inside the statements.

```
Ex: PROC OPTIONS(MAIN);
      DCL TAB(100) CHAR(31) VAR;
      DCL V CHAR(31) VAR;
      DCL (i,NT,NC) BIN FIXED;
      #DCL (CARD CHAR(80));
      #PAT (<SAS>::=<VAR>. '=' .<SAEXP>. ';');
      #PAT (<VAR>::= <#ID>_V:: INCL(V)>;
      #PAT (<SAEXP>::= <TERM>.<TRMS>);
      #PAT (<TERM>::= <#ID> |<#NB>);
      #PAT (<TRMS>::= (0:*)<TRMS1>);
      #PAT (<TRMS1>::= ('+' | '-'). <TERM>);
      DCL INCL ENTRY(CHAR(*)VAR)RETURNS(BIT(1));
      INCL: PROC(X) RETURNS(BIT(1));
            DCL X CHAR(*) VAR;
            DCL I BIN FIXED;
            DO I = 1 TO NT;
                  IF TAB(I) = X
                  THEN RETURN('1'B);
            END;
```

```
            RETURN('O'B);
END INCL;
# UNANCHOR;
        GET LIST(NT,(TAB(I) DO I = 1 TO NT));
        GET LIST(NC);
        DO I = 1 TO NC;
                GET SKIP EDIT(CARD)(A(80));
                # CARD @ = 1;
                DO WHILE(#MAT# (CARD,#//<SAS>));
                PUT SKIP LIST('/* ASSIGNMENT TO',
                        V, '*/');
                END;
                PUT SKIP LIST(CARD);
        END;
END EX;
```

3. PATTERN-MATCHING AND STRUCTURED PROGRAMMING

Since one goal in the present implementation was to conform to the principles of structured programming [1] special care was devoted to the choice of control structures to be used in connection with pattern-matching.

The starting point is the realization that a pattern is a definition of a set of strings. In the simplest case a finite set is defined by ennumerating its members (extensional definition); if string variables are involved the set may still be finite but will be time-varying; in the more complex cases possibly infinite sets are specified algorithmically (intensional definition).

Now if x is a string and P a pattern, the pattern-matching process consits of checking whether x∈P, which in PL/I terms would be an operation returning a bit string value.

It has been shown [10] that the two constructs

    a. IF p THEN f1 ELSE f2;

    b. DO WHILE(p); f; END;

together with the simple sequential execution of statements are sufficient to describe any flowchartable program. In both constructs pattern - matching expressions could occupy the place of p or be part of it (the variable #S could also appear in the rame context).

This should be contrasted to the success/failure exits used for example in SNOBOL, which are equivalent to GO TO's.

Side effects are allowed to occur if assignments within a pattern are specified or if a replacement action takes place. The latter is equivalent to:

    IF p THEN perform replacement;
    ELSE;

and the assignment can be described similarly. Note also that a text of the form:

$$DO\ WHILE\ (\#MAT\#\ (\sigma//\ell_1 \rightarrow r_1//\sigma)|$$
$$\#MAT\#\ (\sigma//\ell_2 \rightarrow r_2//\sigma)|$$
$$\cdots\cdots\cdots\cdots\cdots|$$
$$\#MAT\#\ (\sigma//\ell_n \rightarrow r_n//\sigma));$$
$$END;$$

represents a straightforward implementation of Markov algorithms [11]. If the pattern-matching succeds for any of the n alternatives the corresponding replacement is executed and the first alternative is tried again; otherwise the next alternative is tried, unless the failing alternative is the n-th one in which case the DO group is exited (warning: this worksonly with the optimiying compiler).

The desirability of including Markov algorithms in programming languages is indicated in [12].

An even simpler scheme may be implemented by:

$$\#S = '1'B;$$
$$DO\ WHILE(\#S);$$
$$\#MAT(\sigma//\ell_1 \rightarrow r_1//\sigma);$$
$$\#MAT(\sigma//\ell_2 \rightarrow r_2//\sigma);$$
$$\cdots\cdots\cdots\cdots\cdots$$
$$\#MAT(\sigma//\ell_n \rightarrow r_n//\sigma);$$
$$END;$$

Here the next sequential rule will be tried regardless of the success or failure at the current rule, and after the n-th rule has been tried the set of rules will be tried again starting from the first one if any of them has succeded.

4. FURTHER WORK

We expect to profit from the comments of a number of users for extending the module in useful ways. Attempts will also be made to increase its efficiency.

An aspect to deserve special consideration is the inclusion of left-recursive patterns.

The supervisor shall also be modified in order to make the P/PL/I module and the FORMAC [13] preprocessor compatible, with a view to applications in symbolic mathematics.

REFERENCES

1. Dahl, O., Dijkstra, E., and Hoare, C. - "Structured Programming" - Academic Press, 1972.

2. Cocke, J. and Schwartz, J. - "Programming Languages and their Compilers" - Courant Institute of Mathematical Sciences, 1970.

3. Fateman, R.- "The user level semantic matching capability in MACSYMA" - Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation, 1971.

4. Naur, P. (ed.) - "Revised Report on the Algorithmic Language ALGOL 60", in Programming Systems and Languages" - Rosen, S. (ed.) - McGraw-Hill, 1967.

5. Griswold, R. et al - "The SNOBOL 4 Programming Language" - Prentice Hall, 1971.

6. Griswold, R. - "The macro implementation of SNOBOL 4: a case study of machine-independent software development" - W.H. Freeman, 1972.

7. Wegner, P. - "The structure of SNOBOL" -Cornell University - T.R. 68-9.

8. Gimpel, J. - "A theory of discrete patterns and their implementation in SNOBOL 4" - CACM vol. 16, Feb. 1973.

9. Santos, C. and Furtado, A. - "G/PL/I - extending PL/I for graph processing" - Proceedings of the Computer and Information Sciences Symposium, 1972.

10. Mills, H. - "Mathematical Foundations for Structured Programming" - IBM Corporation,1972.

11. Galler, B. and Perlis, A. - "A view of Programming Languages" - Addison-Wesley, 1969.

12. Cheatham, T. - "The recent evolution of programming languages" - Proceedings of the IFIP Congress 1971, vol. 1 - North-Holland, 1972.

13. Tobey, R. - "PL/I FORMAC interpreter - user's reference manual" - 360D - 03.3.004 - IBM Corporation, 1967.