SYNTACTIC RULES EMBEDDED IN PL/I

Joachim von Peschke
Telefunken Computer GmbH, D 7750 Konstanz
West Germany

Though there are already many translator writing systems [1] , we shall propose yet another one. The philosophy we shall follow is best outlined by a reference to Garwick [2] : "One should exploit a formal syntax as far as possible, at the same time letting the programmer maintain complete control". This aim can easily be achieved by embedding in PL/I syntactic rules similar to BNF.

More specific: For implementing translators (and also for other purposes) a PL/I-program may be useful that acts as a preprocessor: it transforms a character-string or stream-file, called source-text, that may contain besides PL/I-statements also syntactic rules (productions, categories) of a general kind, into a character-string or stream-file, called program-text, that contains only PL/I-statements, i.e. with the rules being expanded into legal PL/I-statements; during execution of the program-text left-right top-down analysis will be performed by recursive descent [3] .

The source-text must have the following form (notation and unexplained variables as known from PL/I-manuals; ignore + in column 1):

```
        source-text ::=
          { pl1-statement | rule }...
        rule ::=
            rule-identifier ::={simple-spec [,]
                               [| redundant  { simple-spec [,] }... ]... ;
+         | {rule-identifier :}... PROC [parameter-list]
+                                        [RECURSIVE] RETURNS(BIT(1));
+                                  [parameter-declaration]...
+                                  group ;
+                                  END-statement
        group ::=
          <[|] [( { iterate-spec[( LOOKUP-spec] | LOOKUP-spec} ] alternatives >
        alternatives ::=
          redundant alternative [ | redundant alternative] ...
        redundant ::=
          [|]...
        alternative ::=
            {simple-spec [,]}...
          | ;[part-statement
             |position-statement | execute-statement | pl1-statement ]...
        simple-spec ::=
            terminal
          | reference
          | group
+         | ¬ simple-spec
        terminal ::=
             char-string-const-except-nullstring
+         | bit-string-const-except-nullstring
+         | [+|-]arithmetic-constant
        reference ::=
            rule-identifier [argument-list]
+         | function-reference
+         | variable-reference
```

```
        part-spec ::=
            terminal
          | reference
          | group
+         | ( expression )
+         | ¬ part-spec
        part-statement ::=
          [label-prefix] = part-spec [ , [name-spec] ] { ; | ELSE pl1-group }
        iterate-spec ::=
          { )
+         | integer )
+         | { DO | BEGIN } [ TO [ = 1  TO ] expression | identifier = 1 TO expression ] )
+         }[separator]
        separator ::=
          * [ ( LOOKUP-spec] alternatives *
+       LOOKUP-spec ::=
+         LOOKUP [ CHAR] )
+       name-spec ::=
+         expression
+       position-statement ::=
+         [label-prefix] , variable-reference
+       execute-statement ::=
+         [label-prefix] . ;
```

Rules for comments and mandatory or optional use of blanks as in PL/I.

Additional terminology: A specification is a simple-spec or a part-spec. An option is a specification that begins with <| An alternative that contains no specification is called pseudo-alternative. The terms left-option, main-specification and right-specification will be defined later.

The meaning of a source-text is found for most cases as follows: if the rules don't contain PL/I-statement, comma, argument-list , label-prefix, and constructs form lines marked above by +, the grammar as specified by those syntactic rules may be called a phrase structure grammar, and for it an equivalent in the well known usual syntax notation exists; in order to get the equivalent delete the symbols equal and semicolon and the constructs comment and redundant, write identifiers with lower case letters instead of upper case letters (and - instead of _), delete the quotes of char-string-constants and underline the string (in most cases not necessary); substitute group brackets as follows:

```
< ALTERNATIVES >              by          { alternatives }
<| ALTERNATIVES >             by          [ alternatives ]
<() ALTERNATIVES >            by          { alternatives } ...
<|() ALTERNATIVES >           by          [ alternatives ] ...
<()* ALTERNATIVES-1 * ALTERNATIVES-2 >                          by
                      { alternatives-2 [ alternatives-1         alternatives-2 ] ... }
<|()* ALTERNATIVES-1 * ALTERNATIVES-2 >                         by
                      [ alternatives-2 [ alternatives-1         alternatives-2 ] ... ]
```

(Remark: These small deviations from the usual notation are necessary in order to remain within the 56-character-set of PL/I, to simplify the preprocessor (so that it needs only 1 pass and no recourse to syntactic analysis of PL/I-statements). Note that a part-statement in its simplest form is only another representation of a specification, but is capable of extensions. See example 1 below.)

Some additional points must be observed:

A) The program-text, generated for a source-text, contains no automatic provision for left-recursion and back-up (back-up is meant relative to the sequence of tokens (atoms, symbols); these are found by other means, i.e. by lexical analysis, see later), therefore, if a phrase structure grammar is specified it must be restricted accordingly. Elimination of left-recursion in favour of iteration raises in practice no difficulty. Back-up may be handled by means that are outside the possiblities of phrase structure grammars as defined above (example 10). (If we had not singled out these special cases, they would slow down the execution of any grammar considerably.)

B) A pseudo-alternative is always regarded as non-matching (examples later).

C) An alternative consisting only of (1 or more) options is regarded as non-matching if not at least 1 of its options matches (that is {[a] [b]} ist taken to mean [a b | a | b]). If the largest group contained in a rule is optional, also all references of this rule must be written as options (so that the preprocessor can handle an optional rule as non-optional rules). (These conventions avoid an option within an iterative group to cause unlimited cycling when the reading position is not advanced.)

D) The alternatives of a rule are normally tested in the order in which they are written, with the first matching one terminating the test; thus the question of uniqueness of the grammar does not arise (one may have for instance two alternatives, the former being a special case of the latter, see the definition of iterate-spec above). (See later the exception with LOOKUP.)

E) The source-text must not contain some identifiers, all of which end with a break-character (they are reserved for use by the preprocessor; note that ELSE is not reserved).

The expansion of a rule into procedural form, i.e. into PL/I-statements, will be performed by the preprocessor in agreement with our conventions essentially as follows: If the specification is a reference (especially a reference of a rule)

```
IF reference THEN ...
```

will be generated; if it is another expression (especially a char-string-constant)

```
IF TERMINAL_(expression) THEN ...
```

will be generated. The ELSE-path for completion of the IF-statement depends on the position of the specification within the sequence that makes up the alternative; let us distinguish between

left-options,
the main-specification (i.e. the first non-optional specification),
and all following specifications as right-specifications,

so that we can say: options are completed by

```
ELSE;
```

If no left-option occurs, the main-specification is completed by

```
ELSE GOTO alternative-did-not-match;
```

(i.e. got to the begin of the next alternative or if no next one exists to the end of the group), otherwise by

```
ELSE DO; IF at-least-1-left-option-matched
THEN CALL backup;
GOTO alternative-did-not-match; END;
```

where backup stands for TBACKUP_ if the expansion contains TERMINAL_, for GBACKUP_ if the specification is a group, and for BACKUP_ otherwise. The ELSE-path for a main-statement after the second * of a group with separator tests also if the control variable of the group equals 1. A non-optional rigth-sepcification is completed by

```
ELSE DO; CALL backup;
         GOTO alternative-did-not-match; END;
```

There will be a simpler expansion if the preprocessor is informed that the subroutine TBACKUP_ does not return control (i.e. if it is not an application like example B in the last chapter below: then the GOTO after TBACKUP_ will be suppressed and TERMINAL_ will be combined with TBACKUP_ to a call of TERMINAL_ _. A rule with ::= not followed by a group, is completed by group brackets; similarly, the parts before and after the second * of a group with separator are completed to groups if necessary. An opening group bracket is expanded to a PROC-statement or a DO-statement of the kind

```
DO [v = 1 BY 1];
```

At the end of each alternative a suitable GOTO- or RETURN-statement is included.

In order to provide maximum flexibility, syntactic analysis is not done entirely by the code generated from the syntactic rules; rather it relies partly on an environment to be supplied by the user. The environment consists of those PL/I-statements in the source-text that declare (among others) the function TERMINAL_ which may have the description

```
ENTRY(CHAR(*)) RETURNS(BIT(1))
```

and serves (after suitable initialization) to compare a terminal (especially a char-sting-constant) with a token in the input-stream. This function will not cope with back-up, i.e. successive calls of TERMINAL_ will use the same token, termed current token, until the first match between terminal and token occurs; thereafter for the next call the next token is used. The environment must also contain the parameterless backup-subroutines (see example 9) and a function TERMINAL_ _ which works like

```
TERMINAL_ _: PROC(STRING) RETURNS(BIT(1));
             DCL STRING CHAR(*);
             IF TERMINAL_(STRING) THEN RETURN('1'B);
             CALL TBACKUP_; STOP;
END TERMINAL_ _;
```

Among the facilities that exceed phrase structure grammars are the following ones:

1) A rule may have parameters and (correspondingly) a specification may have arguments (examples 3 and 4). Terminals may also be constants of other type than CHAR (they have the same form as data in a stream for PL/I-list-directed I/O, example 2). A part-statement may be labeled (restriction: the classification into left-options, main-specification and right-specifications must not be disturbed by jumps) and it may contain a specification for comparison with any scalar expression (example 6B). A ¬ before a group means that each specification within a group will be prefixed by the bit-string-operator ¬ .

2) PL/I-statements may be included in a rule for execution either in the case a specification matches or in the case of no match (ELSE) (example 5). A GOTO leading out of an ELSE-path will in general have as destination the begin of the next alternative or the begin of a pseudo-alternative added at the end of the group. PL/I-statements are copied to the program-text without inspection. If no ELSE-path is specified one will be supplied automatically as already explained.

3) Repetition may be controlled by a variable (example 6C). The construct

(iterate-spec

is expanded as follows:

( )	to	DO v	= 1 BY 1;
(DO identifier)	to	DO identifier	= 1 BY 1;
(integer)	to	DO v	= 1 TO integer;
(DO TO expression)	to	DO v	= 1 TO expression;
(DO identifier = 1 TO expression)	to	DO identifier	= 1 TO expression;

with v supplied automatically. Iterative groups of the first two kinds try to match as often as possible, whereas the other ones are regarded as successful only after they matched the specified number of times (regardless of what may follow). By specifying BEGIN a new scope for declarations will be introduced, i.e. a BEGIN-statement will be included before the DO-group. (Note that an optional iterative group is taken to mean the same as a non-iterative option that contains a non-optional iterative group.)

4) The LOOKUP-spec serves for optimization; it indicates that each alternative on that group-level begins with

{ terminal | ;=terminal[,[name-spec]]; }

(note: no ELSE-path), all these terminals have the same data type, and may be tested in any order (whereas normally alternatives are tested in the order in which they are written). The last alternative may be a pseudo-alternative. The program-text generated (near the end of the group) will be of the kind

```
DCL char-string-array(...) STATIC INIT (...)
                           ALIGNED VAR CHAR(...);
GOTO label(LOOKUP_(char-string-array));
```

Similar expansions for other data types (especially integers). The environment must contain a declaration for the entry LOOKUP_ that compares the current token with the terminals (example 9). If LOOKUP CHAR is specified it is assumed that the tokens (and terminals) will be 1 character long; then the generated program-text will be of the kind

```
DCL INDEX BUILTIN;
GOTO label(INDEX(CURRENT_TOKEN,char-string-constant));
```

with the entry CURRENT_TOKEN contained in the environment (example 9). (Remark: it is up to the PL/I-compiler to produce optimal code for the GOTO. Note some similarity with transition tables.) The expansion of a right-specification in a group with LOOKUP will contain an ELSE-path with a STOP-statement instead of a GOTO-statement.

5) An execute-statement is expanded to

```
CALL EXEC_;
```

A position-statement is expanded to

```
CALL POSITION_(variable-reference);
```

If a specification is followed by a comma, the following CALL-statement is included in the THEN-path of the expansion: if the specification is a group:

```
CALL GNAME_(position, name-spec);
or   CALL GSAVE_(position);
```

if the expansion of the specification contains TERMINAL_ (or TERMINAL_ _):

```
CALL TNAME_(position,namespec);
or   CALL TSAVE_(position);
```

otherwise

```
CALL NAME_(position, name-spec);
or   CALL SAVE_(position);
```

where position is a bit-variable or bit-constant of value '1'B if the specification in the alternative is the first matching one with a comma following. The environment must contain suitable entries. (See example 6) (Note that also for an option only the THEN-path is concerned and the ELSE-path remains empty.)

Examples

1) A rule that may usually be written

a ::= b { c D / e } [f]

may now be written with quotes and angular brackets as follows:

```
A ::= B <C 'D'|
      |E   > <|F> ;
```

Note that stacking of alternatives is still possible. An argument-list may be specified as follows:

```
ARGUMENT_LIST ::= '(' <( )*',' EXPRESSION > ')' ;
```

2) Since also integers may be used as terminals, the lexical analysis performed by the procedure TERMINAL_ may associate 1 or more integers (representing lexical categories) with a token and test for an integer. In order to improve readability, a specification of the form (variable) may be used instead of integer (but see the restriction with LOOKUP).

3) If an arithmetic expression is to be parsed without regard of operator priority, a rule of the following kind may be used:

```
EXPR ::=<( )*OPERATOR*   OPERAND | '(' EXPR   ')' > ;
```

Prioritites are taken into account (without need for back-up) by calling EXPR(0), where

```
EXPR: PROC(PRIOMIN) RECURSIVE RETURNS(BIT(1));
    <  < OPERAND  |  '(' EXPR(0)   ')' >
       <|( )  OPERATOR(PRIOMIN,PRIO)  EXPR(PRIO) >
    > ;
END EXPR;
```

OPERATOR is the lexical category of the infix operators with (non-negative) priority greater than its first parameter; OPERATOR assigns the priority of the current token to its second parameter.

4) A specification may do some unorthodox additional tests; for instance

MEMORY_(rule-identifier-1, rule-identifier-2)

may call rule2 only if the last but one procedure activation is one of rule1 (may be used to combine two rules that differ only slightly, depending on left context). Other dependencies on earlier parts of the analysis may be specified by

BOTH_(expression, rule-identifer)

(see also example 10). The specification

LOOKAHEAD_(array-of-terminals, rule-identifier)

will call the specified rule only after successful comparison of the current token with 1 or more terminals in the same way as TERMINAL_ would do but without giving permission for the reading position to advance to the next token after success (may be used for optimizing rules that can start only with the specified terminals but need many activations before calling TERMINAL_ in the ordinary way . (In special cases a set of integer terminals may be compressed in a bit string to perform the operation & on it instead of using an array.) Also many other specifications depending on right context may be useful as well as a specification that matches any current token (see COMIT and SNOBOL for examples). Note that we have based our language on a very simple algorithm; the more complex kinds of scanning and replacement operations may be performed by refinement; this is possible because of a careful design of the interface between the environment and the control structure represented by the rules.

5) Executable PL/I-statements may be embedded in rules for whatever facility is needed, for instance to build an intermediate parse tree (using the list processing facilities of PL/I) that may or may not reflect the structure of the parse; or they may store identifiers in a symbol table or deliver code; they may aid syntactic analysis by counting parentheses, or by checking if a list has more elements than permitted or by other error checking (for instance an error message in a pseudo-alternative at the end of a group). A GOTO-statement may be used to get a more compact syntactic description; for instance an alternative of the kind

```
a | b { a | c }
```

may be substituted by

```
[ b[ c ]] a
```

if after c a GOTO-statement is included that abnormally terminates 2 groups and skips the specification a (may be plus some adjustment of the tree). A GOTO-statement may select an alternative by using a label-array that is initialized by a prefix before the first statement of each alternative of the same group; the selection may depend on earlier parts of the analysis, for instance on data types associated with identifiers or constants. For declarative PL/I-statements within a rule see example 6C.

6) A part-statement may not only test tokens; in the case of a successful test it may additionally perform an action. Since this action is a CALL to a user-defined procdure in the environment the user may choose what is to be done with the last token. In a very simple case he may choose to substitute the last token by another string (of equal length) and never use it again, but more often he will wish to keep the result of analysis easily accessible for a longer time, i.e. to store a copy of the last token in a separate variable or to store the current reading position in a variable of type FIXED BIN. Or he may proceed in building a parse tree that reflects the structure of the rule activations and that has tokens and names of rules stored in the nodes (in order to get the name of the rule a function would be needed that returns a string similar to that printed by the SNAP-option in on-units), or with the result of the name-spec stored in the nodes; but the name-spec may as well provide a variable in which to store a pointer (of any scope and storage-class) to the last part of the tree or an entry to a code generating routine. Note that these actions could as well be done by explicit PL/I-statements.

Examples:

```
A) /* REORDERING OF ITEMS */
   = B ,ITEM1;   = C ,ITEM2;   PUT EDIT( ITEM2 || ITEM1  ) (A);
B)    =COMPARATIVE,ITEM;  ='AND';  =(ITEM);
c)    DCL  (A,B)(3) PTR INIT((3)NULL);
     <(DO  J = 1 TO 3);  =X,A(J);  =Y,B(J);

      then some actions using A and B  >
```

7) A position-statement may be regarded as a part-statement without test; it may be used to store the reading position or a pointer to the last part of a tree.

8) An execute-statement may be used to execute a subroutine or to evaluate a function, with entry and parameters taken from the last parts of a tree (these parts may be regarded as a stack) and then updating the tree. Or it may deliver the text so far accumulated in the last parts of a tree.

9) Primitive example of an environment; assumption for this example: each token 1 char long.

```
BACKUP_: TBACKUP_: GBACKUP_: PROC;
  ON CONDITION(SNAP) SNAP; SIGNAL CONDITION(SNAP);
  STOP;
END BACKUP_;
DCL TOKEN_ CHAR(1) STATIC, VALID_FOR_TEST_ BIT(1) INIT('0'B) STATIC;
CURRENT_TOKEN_: PROC RETURNS(CHAR(1));
  IF  VALID_FOR_TEST_ THEN DO; GET EDIT(TOKEN_)(A(1));
                               VALID_FOR_TEST_='1'B; END;

  RETURN(TOKEN_);
END CURRENT_TOKEN_;
TERMINAL_: PROC(STRING) RETURNS(BIT(1));
  DCL STRING CHAR(1);
  IF CURRENT_TOKEN_ = STRING THEN DO; VALID_FOR_TEST_ = '0'B;
                                      RETURN('1'B); END;
  RETURN('0'B);
END TERMINAL_;
LOOKUP_: PROC(STRINGS) RETURNS(FIXED BIN);
  DCL  STRING(*) ALIGNED VAR CHAR(*), HBOUND BUILTIN;
  TOKEN_ = CURRENT_TOKEN;
  DO I = 1 TO HBOUND(STRINGS);
    IF TOKEN_ = STRINGS(I) THEN RETURN(I);
  END;
  RETURN(0);
END LOOKUP_;
```

10) Now let us see how the well known methods of handling back-up fit in our framework. If efficiency is of importance, back-up should be avoided; this may be done in simple cases by merging left parts common to several alternatives [4,5], i.e for instance

```
a b | a c
a c | a
{ref :=}... espression;
```

are substituted by

```
a {b | c}
a [c]
ref {:= ref}... [espression-tail];
```

See also the syntax of iterate-spec as defined above. In more complex cases, for instance if the above changes of the syntactic description would lead to a language structure regarded as unsuited for semantic description, or if in a language without reserved identifiers an assignment-statement has to be distinguished from statements of other type, then a (single, simple) prepass may be considered that gathers information, so that decisions in the main pass can be made early enough by a specification of the kind

```
BOTH_(bit-variable, rule-identifier)
```

that tests for the rule only if the variable, set in the preprass, equals '1'B. (If a prepass is used it would reasonably do some additional work, namely change the source into a more machine-oriented form and attach to each token 1 or more classifying integers, so that the main pass can work with integers instead of strings.) A more general but less efficient approach is the following one. That part of the grammar in which back-up is needed, is arranged such that it has its own scope. The specifications in this scope are supplemented in general by a comma. The environment for this scope contains a CONTROLLED variable whose generations give for each active alternative the reading position on entry to the alternative. A generation is allocated and freed by the save-routines introduced by specifications followed by a comma. The backup-routines reset the reading position according to the last generation and free it (as a preparatory step for printing error messages, the backup-routines may also store the rightmost reading position ever reached). In general, a non-optinal right-statement will have no ELSE-path, so that one is supplied automatically as explained above, i.e. one that calls a backup-routine and then jumps to the next alternative. But those part-statements that always must match, are supplied by the user with

```
ELSE CALL ERROR(error-number);
```

so that in case of an error a message can be printed and the analysis of that part of the grammar can be reinitialized (including the CONTROLLED variable). If a tree is used, reading positions may be saved in the tree so that no CONTROLLED variable is needed (storing reading positions may be useful also for other purposes).

11) The language that is input to our preprocessor, i.e. PL/I extended by a variant of BNF, is a means for compact but easy to read formalization of many problems with complex logic; this is due to its possibility of separating the program control structure from the operations, the first being concentrated in "rules" and the second making up an "environment", i.e. the interpretation of the rules. Thus this language gives a framework that is general enough not only for writing translators for formal languages, but also for other purposes, for instance for computational linguistics (note some similarity with the approach of Woods [6], based on LISP, and his critique concerning transformational grammars), as macro language (compile-time facility) and for other kinds of symbol manipulation and pattern matching. It may also be used for merely defining a language, for instance the more complex features of PL/I; this saves the user the bother of learning an additional language (like the metalanguage VDL that is designed only for the purpose of defining and cannot be used for programming as well).

Some problems do not fit well in our framework, for instance (1) enumerating all strings (up to a fixed length) that can be tested successfully by a given rule, or (2) quasi-parallel execution of the alternatives, i.e. immediately after successful execution of one alternative another one of the same group must be selected and executed, possibly with some comparison of results on re-unification of the paths at the end of the group. For these cases the straightforward solution would require something like co-routines, but there are no suitable primitives in PL/I for use in the expansion.

In most cases it will be reasonable to define a new language as extension of one for which a compiler is already available, so that only a preprocessor has to be implemented. This poses constraints on the language design. Otherwise one could introduce new elements by systematic revision instead of merely adding them. The starting-point for such a revision may be the following observation: Like PL/I-FORMAT-statements whose meaning changes according to their use with GET- or PUT-statements, it is possible to use rules, by merely changing the environment, not only for syntactic analysis but also for instance for

A) concatenating a string (or stream-file) out of severalstrings; then the procedure TERMINAL_ would do the concatenation and return always '1'B;
B) performing Boolean operations; sequencing of specifications in an alternative is interpreted as the operation & in the following environment:

```
TERMINAL_:PROC(STRING) RETURNS(BIT(1));
          DCL STRING BIT(1); RETURN(STRING); END;

BACKUP_: TBACKUP_ : GBACKUP_: PROC; END;
```

C) allocating (and initializing) storage to the components of an aggregate (the rule is then an explicit representation of a tree).

Note that a compiler for such a language should avoid redundancies if the entries of the environment are INTERNAL and of the simple kinds mentioned under A and B above. A new systematic language would also provide a case-statement, i.e something like

IF scalar-expression = group { ; | ELSE pl1-group }

This would be in line with what is usually called goto-less or structured programming. The main advantage of a language that is more systematic would be a simpler description of its semantics (may be by prose rather than by a formal metalanguage) and the possibility to construct suitable hardware.

References

1. J.Feldman, D.Gries: Translator Writing Systems,
   Comm.ACM 11(1968), p. 77-113
2. J.V. Garwick: GARGOYLE, A Language for Compiler Writing,
   Comm.ACM 7(1964), p. 16
3. R.M.McClure: An Appraisal of Compiler Technology,
   Spring Joint Computer Conference 1972, p. 1-9
4. D.Cohen: A List Structure Form of Grammars for Syntactic Analysis,
   Comp.Surveys 2(1970), p. 65-82
5. J.M.Foster: A Syntax Improving Program,
   Comp.J. 11(1968), p. 31-34
6. W.A. Woods: Transition Network Grammars for Natural Language Analysis,
   Comm.ACM 13(1970), p. 591-606.