MEMORANDUM
RM-5611-ARPA
OCTOBER 1968

AD 677268

# APAREL—A PARSE-REQUEST LANGUAGE

R. M. Balzer and D. J. Farber



PREPARED FOR:

ADVANCED RESEARCH PROJECTS AGENCY

The RAND Corporation
SANTA MONICA • CALIFORNIA

ARPA ORDER NO. 189-1

MEMORANDUM
RM-5611-ARPA
OCTOBER 1968

# APAREL—A PARSE-REQUEST LANGUAGE

R. M. Balzer and D. J. Farber

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of ARPA.

DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA • 90406

# PREFACE

This Memorandum describes a parsing capability embedded within the PL/I programming language. This extension allows users to specify the syntax of their parse requests in a BNF-like language and the semantics associated with a successful parse request in the PL/I language.

The APAREL system has been designed for a wide range of parsing applications including macro expansion, symbol manipulation, on-line command parsing, analysis of program, and translation of programming languages.

## SUMMARY

This Memorandum describes APAREL, an extension to an algorithmic language (PL/I) that provides the pattern-matching capabilities normally found only in special purpose languages such as SNOBOL4 and TMG. This capability is provided through parse requests stated in a BNF-like format. These parse requests form their own programming language with special sequencing rules. Upon successfully completing a parse request, an associated piece of PL/I code is executed. This code has available for use, as normal PL/I strings, the various pieces (at all levels) of the parse. It also has available as normal PL/I variables, the information concerning which of the various alternatives were successful. Convenient facilities for multiple input-output streams, the initiation of sequences of parse requests as a subroutine, and parse time semantic checks are also included.

APAREL has proven convenient in building a powerful SYNTAX and FUNCTION macro system, an algebraic language preprocessor debugging system, an on-line command parser, a translator for Dataless Programming, and as a general string manipulator.

## CONTENTS

# I. INTRODUCTION

Higher-level descriptions of the problem of compiling have attracted much interest in the past few years. Along with the desire to develop higher-level specialized languages tailored to particular users, the need has arisen to develop similar specialized languages for the writing of these compilers. In general, these so-called compiler-compiler languages are characterized by their facility to define in a BNF-like manner the syntax of the target language. In addition, they possess a programming language designed to operate on and to direct the results of the parsing.

With most compiler-compilers a problem arises both in controlling the parse sequencing and in operating on the results of the parsing. In particular, flexibility is usually lacking in 1) the specification of sequences of parse attempts, 2) the determination of the success or failure of a parse attempt on other than purely syntactic grounds, and 3) the specification of when semantic routines should be invoked. Furthermore, the semantic language is usually a small special-purpose language with facilities for the production of machine code. These systems ignore such other, non-compilation applications for parsers as on-line command parsers (which produce actions instead of machine code), interpretive parsers (which produce pseudo-code), "natural-language" parsers (which produce semantic trees), macro parsers (which produce source code), reformatting programs (which produce formatted listings), and so on. In short, the non-machine-code generation applications of parsers have not been well handled by the translator writing systems.

APAREL attempts to provide a single system for all these applications by providing the user with a powerful general-purpose programming language (PL/I) for performing the wide range of semantics required, and a flexible high-level syntax language for specifying parse attempts, together with facilities for controlling the sequences of these parse attempts, determining success and/or failure on both syntactic and semantic grounds, invoking semantics when desired, and for manipulating the parts of a successful parse. Also, the familiarity of programmers with PL/I and the simplicity of the APAREL extensions and additions make it feasible for potential users to design, implement and modify special-purpose languages without extensive learning.

## II. APAREL--A PARSE-REQUEST LANGUAGE

Our view of translation is composed of three parts:

1) A request to find sequences of syntactic constructs in the source string to be parsed;
2) Context-sensitive validity checks to be made after successful syntactic parses (e.g., has the label been defined before? Is the type of a variable arithmetic? etc.);
3) Semantic routines to be executed only if both the syntactic parse and the context-sensitive validity checks are successful.

This view of translation, while very general, is easy for non-professional translator writers (but experienced programmers) to use in constructing easily modifiable translators.

Requests for parses are specified in a language very similar to BNF rather than a production-type language, because non-professional translator writers tend to conceptualize the syntax of their language top-down (for which purposed BNF-type languages are well suited). Professional translator writers, on the other hand, have learned that the bottom-up approach (for which production-type languages are appropriate) is usually more efficient. Furthermore, they tend to think of both the syntax and semantics at the statement level.

To keep the syntax language simple, while still allowing generality in describing conditions falling in the hazy area between syntax and semantics (which one would like to verify before accepting a parse made on syntactic grounds alone), we allow the specification of "parse-time" routines that return truth values. If they return a value of TRUE, the parse will continue. However, if a value of FALSE is returned, the parse will be unsuccessful, just as if the syntactic parse failed. (The total parse may still be successful if alternatives are available to the unsuccessful subparse.) In addition to returning truth values, these "parse-time" routines may do any semantic processing desired. They are written in the semantic language described below.

The semantic routines are activated upon successful completion of the total parse and successful returns from all the relevant parse-time validity checks, if any, specified within the parse. The code for the semantic routine immediately follows the request for the parse in the syntax language. The semantic language, rather than being a restricted special-purpose language, is full PL/I. The wide range of desirable "semantic" actions resulting from various syntactic parses necessitates a general-purpose programming language; and a major shortcoming of most compiler-compilers has been their restrictions on the semantic language.

To facilitate the semantics, the various pieces of the successful parse are put into normal PL/I strings as specified in the syntax language; and the options chosen, where alternatives were specified in the syntax language, are made available in normal PL/I variables.

## DESCRIPTION OF PARSE-REQUESTS

The syntax of the parse-request language, specified in BNF, appears in Appendix I. However, the following examples are used to describe the language informally.

All parse-requests begin and end with a parse-deliminator (a double colon). After the beginning deliminator, the name of the request (the parse-name) is set off by a colon. The remainder of the parse-request is a list of the alternative parses (parse_alternative_list) desired, separated by OR (|) symbols. The parse-request is successful if any one of the alternatives is successfully parsed. These alternatives may be either parse-elements or lists of parse-elements. Letting $PE_i$ represent a set of parse-elements, we can describe the following parse requests:

| | |
|---|---|
| :: A: $PE_1PE_2$ :: | (the parse-request named "A" will succeed if and only if the parse-string contains $PE_1$ following $PE_2$) |
| :: B: $PE_1 \vert PE_2$ :: | (the parse-request named "B" will succeed if and only if the parse-string contains either $PE_1$ or $PE_2$) |
| :: C: $PE_1 \vert PE_2PE_3PE_4$ :: | (the parse-request named "C" will succeed if and only if the parse-string contains either $PE_1$ or the sequence $PE_2PE_3PE_4$) |

The parse-elements can either be a parse-group or a parse-atom. A parse-group is simply a named or un-named parse-alternative list enclosed in brackets ("⟨" and "⟩"), allowing naming of parts of a parse and alternatives within a sequence of parse-elements. The parse-atoms--the basic, indivisible components of a parse-request--consist of literal strings, parse-request names, and primitive functions; e.g., ARBNO (arbitrary number of), and BAL (balanced strings). These atoms are the components that determine whether a parse is successful or not. The literal strings require that an exact match be found between the literal and the corresponding piece of the parse-string; the parse-request names require that the named parse-request be successful on the corresponding piece of the parse-string; and primitive functions require that the corresponding piece of the parse-string satisfy the conditions of that particular function. There is no syntactic distinction made between these atoms. The category determination is made in the following way. First, the list of primitive functions is checked. If the atom is not a primitive function, then the list of parse-names is checked. Finally, if it is not one of these, it is considered to be a literal. This mechanism alleviates the need to quote most literals within the parse-request language.

Consider the following set of parse-requests to parse PL/I DO statements:

```
:: do_statement:  do iterative_specification
      while_clause ';'::
:: iterative_specification:  ⟨variable = expression
      ⟨to_clause by_clause|by_clause to_clause⟩⟩|::
:: to_clause:  to expression|::
:: by_clause:  by expression|::
:: while_clause:  while '('expression')'|::
```

The do_statement request requires the sequence of atoms do iterative_specification while_clause ;

in the parse-string to be successful. Of these, the middle two are parse-names and invoke parse-requests as they are encountered in a left to right scan. The first and last atoms are literals (because they are not defined as parse-names or primitive functions), and require exact matches with a piece of the parse-string. The final atom is quoted because semicolons are part of the parse-request language (explained below), and the semicolon here is used as a literal.

The iterative_specification request requires the sequence:

1) Variable_expression

2) either 2a. to_clause
   2b. by_clause

   or 2a. by_clause
   2b. to_clause

Variable and expression are primitives, and are defined as specified in the PL/I language specification [1]. Similarly, a to_clause is the literal "to" followed by an expression, or is null, and a while_clause is the literal "while" followed by an expression enclosed in parentheses (quoted because they are part of the syntax language and are used here as literals), or is null.

Thus, the do_statement parse-request invokes parse-requests for iterative_specification and while_clause, and iterative_specification invokes parse_requests for to_clause and by_clause and functions calls for variable and expression.

Unless otherwise specified, the parses allow an arbitrary number of blanks (including none) between atoms, and require the parse start at the beginning of the parse-string although it may be satisfied before the end of the parse-string. Thus, with the above set of parse-requests, successful parses will occur on the following parse-strings:

```
do I = 1;
do I = 1 by 5 to (n-3/2);
do;
do while (A<B);
```

and will fail on the following parse-strings:

| | |
|---|---|
| I = 1 to 10: | (no initial do) |
| Now do I = 1; | (no initial do) |
| do I = 1 to 5 | (no semicolon) |
| do I = 1 to 5 to 6; | (to_clause followed by to_clause) |

The portion of the parse-request language described so far allows fairly sophisticated parse-requests to be specified easily and naturally in a language similar to the normally used syntax description languages (BNF or IBM's syntax notation). However, this is not yet a useful facility, because neither the sequencing rules for initiating parse-requests and for making sequencing decisions based upon the success or failure of a parse-request, nor the method of accessing the various parts of a successful parse have been defined.

## PARSE-REQUEST SEQUENCING RULES

A parse-request-sequence is composed of all parse-requests occurring in a common do-group or block. This does not include any parse-requests contained in blocks or do-groups within the common do-group or block forming parse-request-sequences of their own. The order of parse-requests within a parse-request-sequence is the same as their lexicographical ordering in the block or do-group. The semantic portion of a parse-request is the code between the end of the syntax portion of the parse-request and the beginning of the next parse-request in the parse-request-sequence, or the end of the do-group or block if there are no more parse-requests in the sequence.

A parse-request sequence begins with the first parse-request. If the initial parse-request fails, its semantic code portion is skipped, and the next parse-request in that sequence is tried, and so on, until either a successful parse-request is found or all parse-requests fail. If a successful parse-request is found, the associated semantic code portion is executed; then, normally, the parse-request-sequence is terminated with a successful indication (see Sec. V, Additional Features). Otherwise, the parse-request-sequence is terminated with an unsuccessful indication.

There are three ways in which a parse-request-sequence can be initiated. The first is as a parse-atom in a parse-request. Upon termination, its success-failure indicator is used in determining which alternatives, if any, are successfully parsed. The second is through use of an explicit command, INITIATE PARSE, which specifies which parse-request-sequence to initiate and can be issued in any code portion. Upon termination of the parse-request-sequence, its success or failure is available (see Sec. III, Parse Results), and control continues with the statement following the INITIATE PARSE command. The third method is by program control flowing into the first parse-request in a parse-request-sequence. Upon completion of the parse-request-sequence, its success or failure is available, and control passes to the end statement at the end of the do-group or block in which the parse-request-sequence occurs. Thus, if it is contained in an iterative do-group, control will continue around in the loop until iteration is complete. Otherwise, in blocks or non-iterative do-groups, control will flow out the bottom of the block or do-group upon termination of the parse-request-sequence.

In the first two cases, where a parse-request-sequence is explicitly named, it is specified by referring to the label of the do-group or block in which the parse-request-sequence occurs. If the name of a parse-request is specified instead, only that parse-request will be initiated, and no others in its parse-request-sequence.

These sequencing rules allow the creation of sequences of parse-requests to be attempted, and the control of the execution order of these requests based on the results of the parses and/or explicit program control.

## III. PARSE RESULTS

APAREL also contains capabilities to make the results of a successful (or unsuccessful) parse available to the code portions of the language. This information is of two kinds: pieces of the string parsed and information about which alternatives were successful in the parse.

Various parse-elements, such as parse-request-sequences, parse-requests, parse-alternatives, and parse-groups, can have names specified in APAREL. These names are the means by which the semantic code portions can utilize information about a parse. If "NAME" is the name of one of these parse-elements, then after a parse, a PL/I varying length string variable with the same name will contain that portion of the parse-string corresponding to the named parse-element, and a PL/I variable, whose name is "NAME_OPTION" (i.e., "_OPTION" is appended to the end of the name of the parse-element), will contain the index of the alternative selected within the parse-element. Thus the semantic portions can manipulate desired portions of the parse-string through PL/I's normal string-handling capabilities, and can interrogate any portion of the parse-tree to determine which alternatives were selected.

In applications with large syntax specifications, changing the syntax--either by addition or deletion of an alternative from the syntax--can affect the semantics, because alternative determination is made on an indexed basis; and altering the syntax alternative alters the indexing. To alleviate the problem, APAREL allows the user to label any or all of the alternatives. If a labeled alternative is selected, then the OPTION variable for that group will contain the name of the alternative selected rather than its index. This naming correspondence is invariant under additions or deletions to the set of alternatives.

## IV. PARSE-TIME ROUTINES

Sometimes success or failure of a parse cannot be made on purely syntactic grounds alone; or, it is desired to perform some semantic operations during a parse. For these reasons, the parse-time facility has been included in APAREL. Parse-time routines are indicated in a parse-element by placing the parse-time routine name followed by its arguments, if any, enclosed in parentheses after a semicolon at the end of the parse-element. The parse-time routine will be initiated if and only if the parse-element in which it occurs was successfully parsed. The initiation results in a function call of the parse-time routine passing its arguments, if any. The parse-time routine, like the semantic portions of APAREL, is coded in full PL/I and can make use of all the facilities of APAREL, such as initiating parse-requests, manipulating parse-strings, and interrogating the parse-trees. In addition, the parse-time routine can perform any semantics desired and return a true or false value indicating whether the parse-element it is attached to should be considered successfully parsed or not.

Note that since parse-request-sequences initiated in the syntactic portion of a parse can be a block or a do-group that may begin with a code section or may not contain any parse-requests at all, these parse-request-sequences can be considered parse-time routines that return a success or failure indication (and are formally the same as the parse-time routines discussed above). Both ways of specifying these parse-time routines have been allowed in APAREL, enabling users to choose the one corresponding to their way of conceptualizing its function in their particular application.

## V. ADDITIONAL FEATURES

In the semantic portions of APAREL, very often one would like to output a modified or "translated" version of the parse-string. To make this operation simpler, a special variable, TRANSLATION, has been defined; and whenever an assignment is made to this variable, the value assigned is output to the SYSPRINT data set. For more flexibility, the user may define additional variables as being output variable of specified size and associated with a specified file. When an assignment is made to one of these variables, if the value can be added to the end present string value without exceeding the maximum size of the variable, then the new value is concatenated onto the existing value. If not, then the existing value is output on the file specified and the new value becomes the value of the variable. If the size is not specified, then outputting occurs with every assignment. If neither a file nor a size is specified, then a user-defined procedure of the same name as the output variable is called with the new value as the argument. This allows the user to define arbitrarily complex procedures for outputting, and corresponds to the updating routine (left-hand size function) definitional capability of Dataless Programming [2] and CPL [3].

Similarly, for input, a variable, PARSE_STRING, will be automatically defined to hold the input to be parsed. When the amount of input in this variable falls below a system-defined limit, new input will be concatenated to the variable to fill it out to its maximum size. The user may define additional input variables together with their minimum sizes, maximum sizes, and file from which input is to come. If the minimum and maximum sizes are not specified, references to the input variable will invoke a user-defined accessing function of arbitrary complexity, a la Dataless Programming.

The user also can control which of several input sources is used via the CONSIDER command. He may later re-establish an input source via the RECONSIDER command.

In parsing there are normally three requirements for blank separation between elements in the parse-string. The first is that no blank may occur between the elements. This is indicated in a parse-request by placing a minus sign between the elements. The other two normal blank-separation requirements are that either any number of blanks (perhaps none), or at least one blank (perhaps more), separate the elements. Since the need for each of these requirements is highly application dependent, APAREL allows the user to define the normal mode (indicated in the parse-request by separating the elements by at least one blank) and to request the other requirement by placing a period between the elements. The normal mode is set by either NORMAL SEPARATION IS 0 or NORMAL SEPARATION IS 1 commands. The default setting is NORMAL SEPARATION IS 1.

Similarly, the two normal ways to view the semantic code portion are either as open or closed subroutines. In an open subroutine, flowing out of the bottom of a semantic code portion into a parse-request initiates that parse-request. Whereas in a closed subroutine, flowing out the bottom of a semantic code portion into a parse-request effects a return to the caller of the parse-request whose semantics have just completed. APAREL allows a user to define which of these two modes he is using via the SEMANTICS OPEN and SEMANTICS CLOSED. The default setting is SEMANTICS CLOSED.

Both the SEPARATION and SEMANTICS commands are compile-time commands and affect the interpretation of all lexicographically following parse-requests in the current or contained blocks or do-groups, until either the end of the block or do-group, or another mode command, overrides the present normal mode.

Within a semantic code portion, the user may desire to initiate a remote parse-request, or to terminate the semantics for the present parse. These capabilities are available, respectively, through the INITIATE PARSE and TERMINATE PARSE commands.

The TERMINATE PARSE command is also used to specify the success or failure of a parse-request. TERMINATE PARSE SUCCESSFULLY indicates a successful termination, while TERMINATE PARSE UNSUCCESSFULLY indicates an unsuccessful parse. TERMINATE PARSE with neither operand specified defaults to TERMINATE PARSE SUCCESSFULLY. Thus, a parse-request can be declared unsuccessful in three ways: 1) in the syntactic specification of the parse-request when a syntactic parse is unsuccessful; 2) in a parse-time routine; or 3) in the semantics of a parse-request. The parse is successful only if none of these indicate an unsuccessful parse.

When initiating a remote parse-request-sequence within a semantic code portion, a user often wishes to be able to inspect and manipulate the results of the parse-requests before accepting any translation produced. Since these parse-requests should not (and need not) know that they have been initiated from above, they must be able to create translations just like any other parse-request. Therefore, the user needs a way of telling APAREL to redirect the translation (or output variables) of any parse-request. This redirection causes the translation produced for the specified output variables to be collected into the specified strings for review and/or manipulation by the initiating routine. This redirection is specified as additional operands to the initiate parse-command as follows:

```
INITIATE PARSE k COLLECTING translation IN s AND
  output IN def;
```

The parse-request-sequence named k will be initiated. All translation it, or any parse-request it initiates, produces in the output variable named "translation" will be collected in the string named "s", and all translation produced in the output variable named "output" will be collected in the string named "def".

Finally, by placing a dollar sign ($) in front of parse-names, parse-time routine names, or parse-atoms, the user can indicate indirection; i.e., the parse-name, parse-routine name, or parse-atom specified is the contents of the named string. This facility provides considerable flexibility for users desiring to alter the parse-requests dynamically. It also facilitates context-sensitive parses requiring repetition of a parse-element within the input string.

## VI. EXAMPLES

One use of APAREL is as a macro processor, handling macros of the type commonly referred to as SYNTAX and/or FUNCTION macros [4]. In such an application, a user passes the macros over the source text, translating those portions that satisfy the macro syntax while leaving the rest of the text undisturbed. APAREL is easily restricted to this mode by defining a parse-request that picks off source-language statements, one at a time, from the input stream. The result of this parse, a single source-language statement, is then passed through the various macros that produce the desired translation when a parse request for a macro is satisfied. If the source statement passes all the way through the macros without matching, it is output unmodified. Assuming the parse-request, PL1_statement, has been pre-defined and will pick off one PL/I statement at a time, the following is an APAREL program that acts as a SYNTAX and FUNCTION macro processor for any parse-requests defined in its body.

```
next_PL1_statement:
     INITIATE PL1_statement; /* get next PL/I statement*/
     IF PL1_statement_option = 0 /* was the parse successful*/
          THEN DO; /* no, end of input must have been reached*/
               IF CONSIDERED_STRING (CONSIDER_LEVEL)='rescan'
                         THEN DO; /*reconsider the original
                         input string*/
                    RECONSIDER;
                    GO TO next_PL1_statement;
                    END;
```

```
          ELSE /* we have exhausted the original input
                    string*/
               TERMINATE PARSE; /* terminate the parse
                    in this manner in case we were
                    initiated by someone, and are not
                    the top level routine*/
          END;
ELSE DO; /* parse was successful, we now have a single
                    PL/I statement*/
CONSIDER PL1_statement; /* use result of PL/I statement
                    as parse-string*/
INITIATE user_macros COLLECTING translation IN partial_
                    translation; /* initiate users
                    syntax and function macro parse-
                    request-sequence contained in the
                    block or do_group labeled "user-
                    macros". The translation output
                    of these macros is collected in
                    the PL/I string "partial_trans-
                    lation"*/
If user_macros_option¬ = 0 THEN DO; /* one of the parse-
                    requests in the user_macros parse-
                    request-sequence was successful*/
     RECONSIDER; /* stop considering PL1_statement and
                    reconsider the parse-string in
                    effect before it*/
     rescan = partial_translation||rescan; /* add
                    partial translations to front of
                    rescan string so that it will be
                    retranslated first. Notice that
                    this defines a depth first
                    translation*/
```

```
IF CONSIDERED_STRING (CONSIDER_LEVEL)¬ = 'rescan'
                    /* is rescan the currently considered
                    parse-string*/
        THEN /* no it is not the currently considered
                    string*/
                CONSIDER rescan; /* make it the current
                    parse-string*/
    GO TO next_PL1_statement;
    END;
ELSE DO; /* none of the parse-requests in the user_macros
                    parse-request-sequence were successful*/
    TRANSLATION = PL1_statement; /* output the
                    PL1_statement that did not match*/
    GO TO next_PL1_statement;
    END;
```

Continuing the above example, two parse-requests are shown below, both of which provide translations into Pl/I. They are placed in the do_group labeled "user_macros" to conform to the preceding example/s initiation command. The first is a syntax macro that translates increment or decrement commands, and the second is a functional macro that translates various notations for asking if a value is equal to one of a number of items. Notice that the only difference between syntax and function macros is that syntax macros require successful parses to be anchored to the beginning of the parse-string, while functional macros allow successful parses anywhere within the parse-string.

The annotated parse-requests are given below, followed by a set of example input parse-strings with their translations:

```
user_macros:  DO; /* begin labeled do group that defines a
                     parse sequence*/
    NORMAL SEPARATION IS1; /* unless otherwise specified,
                     parse-elements must be separated
                     by one or more blanks*/
    SEMANTICS CLOSED; /* upon reaching the end of the
                     semantics of a parse-request,
                     automatically generate a terminate-
                     parse command*/
:: Increment_command:  command_type <updated_variable:
                     subscripted_variable> by <increment_
                     amount: ARB>.';' :: /* an increment
                     command is a command type followed
                     by a possibly subscripted variable,
                     called "updated_variable", followed
                     by the literal "BY" (literal since
                     it is not defined), followed by an
                     arbitrary string called "increment_
                     amount", followed by a semicolon
                     (the semicolon has to be quoted
                     since it is part of the parse-
                     request language).  The period
                     indicates that a space is not re-
                     quired in front of the semicolon.*/
    IF command_type_option = "increment_command" /* was the
                     option in command_type labeled
                     "increment_command" chosen*/
    THEN /* yes this is an increment command*/
        translation = updated_variable||'='||updated_
                     variable||'+'||increment-amount
                     ||';'; /*output PL1 assignment for
                     incrementing variable*/
```

```
ELSE /* no, must be decrement command*/
     translation = updated_variable||'='||updated_
                        variable||'-('||increment_amount
                        ||');'; /*output PL/I assignment
                        for decrementing variable enclosing
                        increment_amount in parenthesis*/
/* the next statement is a parse-request in the same
                        block or do group as the present
                        parse-request; therefore, it
                        indicates the end of this semantic
                        code; and since semantics have to
                        be set closed, it automatically
                        generates a terminate-parse
                        command.*/
/* this parse-request will be activated if the preceding
                        parse-request failed*/
:: one_of:⟨front:ARB⟩⟨x:  subscripted_variable⟩⟨is|is among|.=.⟩
                        alternative_list⟨back:ARB⟩::
                        /* a one_of function macro is an
                        arbitrary string named "front"
                        followed by a subscripted variable
                        named "x" followed by either "is",
                        "is" followed by "among", or by "=".
                        This is followed by an alternative_
                        list followed by an arbitrary string
                        named "back".  The separation between
                        these elements is one or more blanks--
                        except for the equal sign, which may
                        have zero or more blanks on either
                        side of it as indicated by the normal
                        separation override notation (the
                        periods).*/
```

```
        translation = front||PL1_alternatives||back; /*the
                       string "PL1_alternatives" replaces
                       the function macro in the parse-
                       string, and the result is output as
                       the translation of the parse-string.
                       The PL1_alternatives string was
                       built up in the semantic portion of
                       the alternative_list parse-request
                       shown below*/
    END user_macros; /* this is the end of the do-group.
                       It indicates the end of the semantic
                       portion of the one_of parse-request;
                       and, since semantics are closed,
                       automatically generates a terminate
                       parse-command for that parse-request.
                       If this parse-request had failed,
                       then, since it was the last parse-
                       request in the parse-request-sequence,
                       the sequence would have failed.*/
    /* the following are parse-requests referred to above.
                       Since they are defined in another
                       do-group or block than the preceding
                       parse-requests, they do not form
                       part of its parse-request-sequence.*/
:: subscripted_variable:  variable <.'('.BAL.')'.|>::  /*a
                       subscripted variable is a variable
                       followed by a left parenthesis
                       followed by an arbitrary string
                       balanced with parentheses followed
                       by a right parenthesis or a variable
                       followed by a null.  The parentheses
```

and the balanced string do not have to be separated by blanks. There are no semantics specified for this parse-request.*/

:: command_type: ⟨increment_command: increment|i|inc⟩| ⟨decrement_command: decrement|d|dec⟩:: /* a command type is either an increment_command or a decrement_ command. These two types can each be indicated in one of three ways: "increment", "i"; or "inc" and "decrement", "d"; or "dec". There are no semantics specified for this parse-request */

:: alternative_list: Initial_semantics BOUND(1,alternative, ⟨','| or ⟩) :: /* an alternative_ list is an initial_semantics followed by an arbitrary number (with a minimum of one) of alternatives separated by either commas or the literal "or". The parse-request, initial_semantics, does not perform any parsing, but is used co initial- ize the string, PL1_alternative, used in the semantics of "alterna- tive". There are no semantics specified for this parse-request.*/

:: alternative: expression: /* an alternative is an ex- pression. Its semantics follow. The same effect could have been achieved by replacing alternative

```
                     in the parse-request alternative_list
                     by
                         expression; alternative_semantics
                     where alternative_semantics would be
                     the name of the following semantic
                     routine.  The choice is left to the
                     user depending on his particular
                     basis.*/
     if ¬ first_alternative then PL1_alternatives=PL1_
                     alternatives||'|'||x||'='|| expression;
                     /* the alternative is added to the end
                     of the alternatives already found.
                     It is separated from the preceding
                     alternatives by "|", and consists of
                     the subscripted variable (the value
                     of x from the parse-request, "one_of")
                     followed by an equal sign followed
                     by an expression just parsed above.*/
     ELSE DO; /* this is the first alternative*/  first_
                     alternative = '0'B; /* indicate no
                     longer first alternative*/
                     PL1_alternatives = x||'='|| expression;
                     /* PL1_alternatives is set to the
                     first alternative found*/
     END;
     TERMINATE PARSE; /* indicate end of semantics*/
initial_semantics:  DO; /* initial_semantics is a parse-request-
                     sequence containing no parse-request*/
     first_alternative = '1'B; /* indicate parse-request was
                     successful*/
     END;
```

## VII. TRANSLATION RESULTS

Using the APAREL program defined in Sec. VI, we indicate below the translations that would result for various input examples. If the input passes through unchanged, the translation entry is left blank to facilitate recognition.

| input | translation | comments |
| --- | --- | --- |
| increment x by 5; | x = x+5 | |
| d abc by x-4; | abc = abc - (x-4); | the decrement translation supplies parentheses around the decrement amount. |
| i def by7; | | no separating blank after 'by' |
| decrement by 3; | | 'by' is picked up as the subscripted variable, but the parse then fails because 'by' cannot be found. |
| if abc is x-3 or 0 then do; | if abc = x-3 | abc = 0 then do; | |
| R = (def is among 1,2,Z-4 or 9); | R = (def = 1 | def = 2 | def = Z-4 | def = 9); | |

| input | translation | comments |
|---|---|---|
| when h = 5, or 7 then do; | when h = 5 \| h = or 7 then do; | comma after 5 causes parser to pick up "or" as an expression rather than as the separator between expressions. The syntax of the functional macro should be corrected to prevent this error. Notice how the error is reflected in the translation; |
| if x is 3,>5, or 0 | | ">5" is not an expression. |
| if x = 1 or 4 then i x by x-1; | if x = 1 \| x = 4 then x = x+x-1; | |

## VIII. IMPLEMENTATION

Implementation of APAREL was of course affected by the desirability of using PL/I as the basic language for expressing the semantic operations. PL/I was chosen because of its power and familiarity. At the same time, it was decided that the actual parsing be implemented as an interpreter. This necessitated that the parse-request language of APAREL be translated into an interpretable structure. The benefits gained from the simplicity and extendability of this approach are considered worth the price of translation.

APAREL is a two-pass processor that first translates the APAREL program into an interpretive program for driving the parser and then crosscouples the results of the parse with the semantic section of the user's program. Two passes are necessary because of certain limitations of PL/I, in which APAREL is programmed. The main limitation is the inability to dynamically define equivalences, at least in the PL/I DEFINED context.

The heart of APAREL is an interpretive representation of the parse-request. The structure of the interpretive list and other internal details are described below.

The problem of relating APAREL names with PL/I variables has been solved by twice examining the parse-request. The first is done in a pass over the APAREL program. The main purpose of this pass is to form a symbol table identical with that used during execution. This symbol table will be used at run time to hold (among other things) the current values of named items. The result of the first pass is to output, for compilation by PL/I, a modified APAREL program where the parse-requests are replaced by a call on a COMPILE subroutine, which is given both the name of the parse-request and the request body and uses these at run time to construct the interpretive table. There is then generated a call on the parser to initiate the parse-request-sequence. PL/I code is generated, which when executed at run time, after a successful parse, will transfer the values of names from the APAREL symbol table into the appropriate PL/I strings. Code is also produced that handles the conditional flow, within a parse-request and between parse-requests, which depends on the successful-unsuccessful indicator returned by the parse-request. In addition, the semantic body is scanned for use of the TRANSLATION string, output strings, parse-request blocks, etc.; and the appropriate modifications and additions to the APAREL program are made. At the conclusion of this first pass, the output is compiled as a normal PL/I program and executed. During the "second pass" the COMPILE statements are threaded through, resulting in the construction of a run-time SYMBOL_TABLE. Then control is passed to the modified APAREL program.

# IX. BASIC DATA STRUCTURE

APAREL is basically a parser driven from an interpretive form of the syntax description given below. Thus, the system data base is molded for such an interpretive environment. The three basic data structures in APAREL are defined and discussed in turn. The PL/I declare statement will be used to define the structures.

## A. SYMBOL TABLE

```
DECLARE 1 symbol_table (  ),
          2 literal CHARACTER (  ) VARIABLE,
          2 value CHARACTER (  ) VARIABLE,
          2 rule BINARY FIXED,
          2 option CHARACTER (  ) VARIABLE;
```

In the interpretive structure all references to a symbol are through its index in the SYMBOL_TABLE. The literal character string contains the name of the symbol represented by this symbol-table entry. If the symbol is acting as an APAREL name, then its value can be referenced by the value subfield, which contains a character string as the value of the symbol. When in the course of a parse, a successful match is done to a named alternative structure, the name of the alternative, or its index number if no name has been given to the alternative, is assigned to the OPTION subfield of the entry in the SYMBOL_TABLE corresponding to the name of the alternative structure. The rule that is named by this literal is pointed to in the sense that SYMBOL_TABLE.RULE points to the location in the SYNTAX table at which this rule begins.

## B. SYNTAX

```
DECLARE 1 SYNTAX ( ),
          2 TYPE BINARY FIXED,
          2 SYMBOL_INDEX BINARY FIXED;
```

This array contains the interpretive code that drives the parser. Basically, it contains the parse-atom type as an integer in the TYPE field, and (in most cases) a pointer, SYMBOL_INDEX (an index), to the symbol-table entry for this parse atom.

As an example, consider the APAREL rule:

:: SAM : JOE ⟨ ABE : MAY ; IKE ⟩ ⟨ ABE | DEF ⟩ ::

During the second pass, after this parse-request has been "compiled," the SYMBOL_TABLE will have seven entries. For example, the first entry will be SYMBOL_TABLE(1).LITERAL containing the character string "SAM", SYMBOL_TABLE(1).RULE containing the location (1) in SYNTAX of this rule. Since, at this time, the value of SAM is null, the value field is set to the null string and the option field to zero. Similarly JOE, ABE, MAY, IKE, ABC, and DEF are assigned slots 2, 3, 4, 5, 6, and 7 in the SYMBOL_TABLE. At this point, only the literal field is set. The other fields may be filled in further as additional information is gathered from the "compilation" of later parse-requests.

Figure 1 illustrates the SYNTAX entry for this rule.

| | TYPE | SYMBOL_INDEX | |
|---|---|---|---|
| SYNTAX (1) | LITERAL | 2 (JOE) | Note: namely the string is in SYMBOL_TABLE(2).literal. |
| (2) | Beginning of Group | 3 (ABE) | Note: symbol_index has the name of the group. |
| | NAME | 4 (MAY) | Note: if MAY is a rule name at the time of execution it will activate the parse-request or else it will use the literal string MAY. |
| | End of Group | 5 (IKE) | Note: in the end of group is the name of the procedure (a pointer to the symbol_table) which will be used if the group succeeds. |
| | Beginning of Group | | Note: no group name. |
| | NAME | 3 (ABE) | |
| | or | | Note: marks an alternative. |
| | NAME | 7 (DEF) | |
| | End of Group | | |
| | End of Rule | 1 (SAM) | Note: the end of rule contains the rule named for assignment of the substring matched if the parse-request is successful. |
| | | | |

Fig. 1--Syntax of a Rule

## Appendix

## BNF DEFINITION OF APAREL'S SYNTAX LANGUAGE

```
<PARSE_REQUEST> := <PARSE_DELIMINATOR><PARSE_NAME>:
          <PARSE_ALTERNATIVE_LIST><PARSE_DELIMINATOR>
<PARSE_ALTERNATIVE_LIST> := <PARSE_ELEMENT_LIST> |
          <PARSE_ELEMENT_LIST> '|' <PARSE_ALTERNATIVE_LIST>
<PARSE_ELEMENT_LIST> := <PARSE_ELEMENT> |
          <PARSE_ELEMENT>;<PARSE_TIME_ROUTINE_NAME> |
          <PARSE_ELEMENT><PARSE_ELEMENT_LIST> |
          <PARSE_ELEMENT>.<PARSE_ELEMENT_LIST>
<PARSE_ELEMENT> := <PARSE_ATOM> | <PARSE_GROUP>
<PARSE_GROUP> := '<' <PARSE_ALTERNATIVE_LIST> '>' |
          '<' <PARSE_NAME>:<PARSE_ALTERNATIVE_LIST> '>'
<PARSE_ATOM> := <PARSE_NAME> | <TEXT_LITERAL>
<PARSE_NAME> := <PL/1 IDENTIFIER>
<PARSE_DELIMINATOR> := ::
<PARSE_TIME_ROUTINE_NAME> := <NAME OF A PL/1 BIT VALUED FUNCTION >
```

## REFERENCES

1. PL/I Language Specification, IBM Corporation, form C28-6571-4.
2. Balzer, R. M., Dataless Programming, The RAND Corporation, RM-5290-ARPA, July 1967.
3. Strachey, C. (ed.), CPL Working Papers, London Institute of Computer Science and the University Mathematical Laboratory, Cambridge, 1966.
4. Leavenworth, B. M., "Syntax Macros and Extended Translation," Communications of ACM, Vol. 9, No. 11, November 1966, pp. 790-793.
5. Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Proceedings of the International Conference on Information Processing, UNESCO (1959), pp. 125-132.
6. Cheatham, T. E., "The Introduction of Definitional Facilities into Higher Level Programming Languages," Proceedings of the AFIPS FJCC (1966), pp. 623-637.
7. Farber, D. J., R. E. Griswold, and I. P. Polonsky, "SNOBOL3," Bell System Technical Journal, August 1966.
8. Feldman, J. A., and D. Gries, "Translator Writing Systems," Technical Report #CS69, Stanford, June 9, 1967.
9. Gauer, B., and A. J. Perlis, "A Proposal for Definitions in ALGOL," Communications of the ACM, Vol. 10, April 1967, pp. 204-219.
10. Irons, E. T., "A Syntax Directed Compiler for ALGOL 60," Communications of the ACM, Vol. 4, January 1961, pp. 51-55.
11. McClure, R. M., "TM6--A Syntax-Directed Compiler," Proceedings of the 20th National ACM Conference, 1965, pp. 262-274.
12. Mondschein, L., VITAL Compiler-Compiler Reference Manual, TN 1967-1, Lincoln Laboratory, January 1967.

DOCUMENT CONTROL DATA

| 1 ORIGINATING ACTIVITY | 2a. REPORT SECURITY CLASSIFICATION |
|---|---|
| THE RAND CORPORATION | UNCLASSIFIED |
| | 2b. GROUP |

3. REPORT TITLE

APAREL--A PARSE-REQUEST LANGUAGE

4. AUTHOR(S) (Last name, first name, initial)

Balzer, R. M. and D. J. Farber

| 5. REPORT DATE | 6a. TOTAL No. OF PAGES | 6b. No. OF REFS. |
|---|---|---|
| October 1968 | 42 | 12 |

| 7. CONTRACT OR GRANT No. | 8. ORIGINATOR'S REPORT No. |
|---|---|
| DAHC15-67-C-0141 | RM-5611-ARPA |

| 9a. AVAILABILITY/LIMITATION NOTICES | 9b. SPONSORING AGENCY |
|---|---|
| DDC-1 | Advanced Research Projects Agency |

10. ABSTRACT

A description, without listing, of APAREL, a flexible high-level parsing-language extension to PL/I that provides the pattern-matching capability normally found only in special-purpose languages such as SNOBOL4 and TMG. This capability is provided through parse requests stated in a BNF-like format, which form their own programming language with special sequencing rules. Upon successful completion of a parse request, an associated piece of PL/I code is executed, which has as normal PL/I strings the various elements of the parse (at all levels). It has as normal PL/I variables the information concerning which of the various alternative parsings were successful. Convenient facilities for multiple input-output streams, the initiation of sequences of parse requests as a subroutine, and parse-time semantic checks are included. APAREL has proved convenient in building a powerful SYNTAX and FUNCTION macro system, an algebraic language preprocessor debugging system, an on-line command parser which produces actions rather than machine code, and a translator for Dataless Programming (RM-5290-ARPA), and as a general string manipulator.

11. KEY WORDS

Language and linguistics
Computer programming language