ARPA ORDER NO. 189-1

MEMORANDUM RM-5611-1-ARPA SEPTEMBER 1969

**AD**694442

## APAREL-A PARSE-REQUEST LANGUAGE

R. M. Balzer and D. J. Farber



PREPARED FOR: ADVANCED RESEARCH PROJECTS AGENCY

The Corporation SANTA MONICA + CALIFORNIA

THIS DOCUMENT HAS BEEN APPROVED FOR PUBLIC RELEASE AND SALL; ITS DISTRIBUTION IS UNLIMITED.

ARPA ORDER NO. 189-1

MEMORANDUM RM-5611-1-ARPA SEPTEMBER 11 69

## APAREL-A PARSE-RE \_\_\_\_\_ST LANGUAGE

M. Balzer and D. J. Farber

This research is supported by the Advanced Research Projects Agency under Contract No. DA'tC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of ARP2.

DISTRIBUTION\_STATEMENT This document has been approved for public release and safe; its distribution is unlimited.



This study is presented as a competent treatment of the subject, worthy of publication. The Rand Corporation vonches for the quality of the research, without necessarily endorsing the opinions and conclusions of the authors.

,

Published by The RAND Corporation

#### PREFACE

Ti is 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 h wide range of parsing applications including macro expansion, symbol manipulation, on-line command parsing, analysis of programs, and translation of programming languages.

This revised Memorandum, representing the actual implementation of the system, supersedes the authors' <u>APAREL--A</u> <u>Parse-Request Language</u>, The Rand Corporation, RM-5611-ARPA, October 1968.

APAREL has been developed as a basic tool for use in man-machine communication studies at Rand under sponsorship of the Advanced Research Projects Agency.

#### SUMMARY

This Memorandum describes APAREL, an extension to an algorithmic language (PL/I) that provides the patternmatching capabilities normally found only in such specialpurpose languages as SNOBOL4 and TMG. This capability is provided through parse-requests stated in a BNF-like for-These parse-requests form their own programming mat. 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 parserequests as a subroutine, and parse-time semantic checks are also included.

APAREL has proven convenient not only as a general string manipulator but also in building a powerful SYNTAX and FUNCTION macro system, an algebraic language preprocessor debugging system, an cn-line command parser, and a translator for Dataless Programming.

-v-

## -vii-

## CONTENTS

PREFACE	iii		
SUMMARY	v		
Section			
I. INTRODUCTION	1		
II. APARELA PARSE-REQUEST LANGUAGE Description of Parse-Requests Parse-Request Sequencing Rules	3 4 8		
III. PARSE RESULTS	11		
IV. PARSE-TIME ROUTINES	13		
V. ADDITIONAL FEATURES	14		
VI. EXAMPLES	18		
VII. TRANSLATION RESULTS	26		
VIII. IMPLEMENTATION	28		
Appendix			
BNF DEFINITION OF APAREL'S SYNTAX LANGUAGE	30		
REFERENCES	31		

# **BLANK PAGE**

#### 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 compilercompiler 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, noncompilation applications for parsers as on-line command parsers (which produce actions instead of machine code), interpretive parsers (which produce pseudocode), "natural-language" parsers (which produce semantic trees), macro parsers (which produce source code), reformatting programs (which produce formatted listings), and In short, the nonmachine-code generation applicaso on. tions of parsers have generally 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

#### -1-

the wide range of semantics required, and a flexible highlevel syntax language for specifying parse attempts, together with facilities for controlling the sequencing of these parse attempts, determining success and/or failure on both syntactic and semantic grounds, invoking seman ics 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:

- 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; (For example, has the label been defined before? Is the type of a variable arithmetic? etc.)
- 3) Semantic rout nes 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 nonprofessional 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 Floyd-Evans production language), because nonprofessional translator writers tend to conceptualize the syntax of their language top-down (for which purposes BNF-type languages are well suited). Professional translator writers, on the other hand, have learned that the bottom-up approach (for which productiontype languages are appropriate) is usually more efficient. Furthermore, the former 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 azy 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 contine. However, if a value of FALSE is returned, the parse will be unsuccessful, just as if the

- 3-

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 routine associated with a parse is activated upon successful completion of that 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 syntatic 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 PARSL-REQUESTS

The syntax of the parse-request language, specified in BNF, appears in the Appendix. 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-request-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 parsed successfully. These alternatives may be either parse-elements or lists of parse-elements. Letting PE<sub>i</sub> represent a set of parseelements, we can describe the following parse requests:

:: A:	PE1 <sup>PE</sup> 2 ::	(the parse-request named "A" will succeed if and	
		only if the parse-string	
		contains $PE_1$ followed by $PE_2$ )	
:: B:	$PE_1   PE_2 ::$	(the parse-request named	
	÷ -	"B" will succeed if and	
		only if the parse-string	
		contains either $PE_1$ or $PE_2$ )	
:: C:	PE1 PE2PE3PE4 ::	(the parse-request named	
	1 2 3 7	"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 unnamed 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, parse-request-sequence names (described below, pp. 8-10), and primitive parse-request functions; e.g., ARBNO (arbitrary but nonzero number of the first argument separated by the second argument, if there is more than one occurrence of the first argument), 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 and parse-request-sequence names require that the named parse-request or parse-request-sequence be

-5-

successful on the corresponding piece of the parse-string; and primitive parse-request 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 parse-request functions is checked. If the atom is not a primitive parse-request function, then the list of parsenames (both parse-request and parse-request-sequence 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 ';'::
- :: 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 parsenames 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 either the sequence:

- 1) Variable=expression
- 2) either 2a. to\_clause 2b. by\_clause or 2a. by\_clause 2b. toclause

or NULL.

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 parserequests 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 pieces of the parsestring and require that 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 parserequests, successful parses will occur on the following parsestrings;

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

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)

--7-

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 parserequests 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 parserequests 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 parserequest-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-requesting-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 parserequest. 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-requestsequence is terminated with a successful indication (see Sec. V, Additional Features). Otherwise, the parse-requestsequence is terminated with an unsuccessful indication.

-8-

There are three ways in which a parse-request-sequence can be initiated. The first is as a parse-atom in a parserequest. 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-requestsequence 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 parserequest-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 noniterative do-groups, control will flow out the bottom of the block or do-group upon termination of the parse-requestsequence.

In the first two cases, in which a parse-request-sequence is explicitly named, it is specified by referring to the label (which must be in the same block as the invoking statement) 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.

As stated previously, the semantic routine associated with a parse-request is activated upon successful completion of that parse-request and upon successful return from all the

-9-

relevant parse-time validity checks, if any, specified within the parse-request. This is true whichever way the parse-request is initiated. Thus, if a parse-request, Pl, is initiated as a parse-atom or a parse-request, P2, and if it is successful, then its semantic routine will be initiated at that point, in the midst of the parse of P2. Semantics thus can be initiated at any point during a parse, giving the user considerable flexibility. However, care must be exercised when specifying "intermediate" semantics because the parse may fail later in the parse element list, which contained the parse-request that invoked the semantics, and either move on to the next alternative or fail completely.

#### 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: 1) pieces of the string parsed, and 2) 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 parseelements, 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. (In the case of a parse-request-sequence, the name is both the name of the result string and the label of the DO-block. APAREL contextually resolves all uses of this name to remove any ambiguity.) Also, 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 iterrogate any portion of the parsetree 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

#### -11-

rather than its index (APAREL contextually resolves all uses of this variable so that it can, in effect, take on either string or numeric values). 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 "oded in full PL/I and can make use of all the facilities of APAREL, such as initiating parserequests, manipulating parse-strings, and interrogating the In addition, the parse-time routine can perparse-trees. form any semantics desired and return a true or false value indicating whether the parse-element to which it is attached should be considered successfully parsed.

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 parserequests 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.

-13-

#### V. ADDITIONAL FELTURES

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 of the 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, <u>a la</u> Dataless Programming. These minimum and maximum sizes limit the amount of backtracking that can occur.

-14-

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. These commands stack and unstack respectively which input source is being parsed. CONSIDER\_LEVFL contains the rumber of input sources so stacked, and CONSIDER\_STRING is an array containing, in ascending order, the names of those stacked input sources.

In parsing there are normally three requirements for blank separation between the individual segments of the parse-string matched by parse-atoms. The first is that no blank may occur between the segments. This is indicated in a parse-request by placing a minus sign between the parseelements. 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 segments. Since the need for each of these requirements is highly application dependent, APAREL allows the user to define the normal mode (used between parse-elements unless otherwise specified) and to request the other requirement by placing a period between the parse-elements. The normal mode is set by either NORMAL SEPARATION IS 0 or NORMAL SEPARATION IS 1 command 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 compiletime 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

-15-

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 TERMI-NATE PARSE UNSUCCESSFULLY indicates an unsuccessful parse. TERMINATE PARSE with neither operand specified defaults to TERMINATE FARSE 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 indicates an unsuccessful parse.

When initiating a parse-request-sequence, 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 parserequest. 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 x IN y , and separated by 'AND' operands (of the form to the initiate parse-command. For example:

)

INITIATE PARSE & COLLECTING translation IN s AND output IN def;

-16-

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 instead in the string named "s". and all translation produced in the output variable named "output" will be collected instead 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, parseroutine name, or parse-atom specified is the contents of the named string. This facility, accomplished via a runtime symbol table of all parse-related names (which must all be unique), provides considerable flexibility for users desixing to alter the parse-requests dynamically. It also facilitates context-sensitive parses requiring rep<sup>-+:-</sup>ion 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, PLI statement, has been predefined 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.

/\* Method: PL/I statements are picked off the input stream one at a time and used as the parse-string input for the user-defined syntax and function macros contained in the parse-request-sequence USER MACROS. If no parse-request in this parse-request-sequence is successful, then the PL/I statement is output. Otherwise, the translation produced is added to the front of the string RESCAN. If this string is not already being CONSIDERed as the input string from which PL/I statements are picked off, it is so CONSIDERed. Thus all PL/I statements in the translation produced by the USER MACROS are processed before any more is taken from the original input source. After RESCAN has been exhausted, the original input source is RECONSIDERed \*/

-18-

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 2L/I statement as parse-string for user macros\*/

- INITIATE user\_macros COLLECTING translation IN partial\_ translation; /\* initiate users syntax and function macro parserequest-sequence contained in the block or dc\_group labeled "usermacros". The translation output of these macros is collected in the PL/I string "partial\_translation"\*/ RECONSIDER; /\* stop considering PLL\_statement and
  - reconsider the parse-string in effect before it\*/

-19-

```
If user macros option-= 0 THEN DO; /* one of the parse-
                    requests in the user macros parse-
                    request-sequence was successful*/
    rescan = partial translational | rescan; /* add
                    partial translation 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) \neg = '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 IS 1; /\* 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 terminateparse 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 parserequest language.) The period indicates that a space is not required 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

-21-

```
||';'; /*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 parentheses*/
    /* 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 (the ARB primitive
                         parse-request function matches the
                         smallest string that allows the
                         rest of the parse-request to be
                         successful; this may require
                         backup and repeated attempts, each
                         time increasing the length of the
                         string matched by _ne ARB parse-
                         request function) 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
```

-22-

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||PLl\_alternatives||back; /\*the
 string "PLl\_alternatives" replaces
 the function macro in the parse string, and the result is output as
 the translation of the parse-string.
 The PLl\_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, it automatically generates a terminate parse-command for that parse-request. If this parse-request had failed, then, since it was the last parserequest 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 preced 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

-23-

-24-	
<pre>-24- by a right parenthesis or a variabl followed by a null. The parenthese 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", " "; or "dec". There are no semantics specified for this parse-request */ :: alternative_list: Initial_semantics ARBNO(alternative, (','  or \) :: /* an alternative_ list is an initial_semantics follow by an arbitrary number (with a</pre>	ed
<pre>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 to initial ize the string, PLL_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_li by expression; alternative_semantic</pre>	- st

```
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 = 'l'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 paren- theses around the decrement amount.
i def by7;		no separating blank after 'by'
decrement by 3;		'by' is picked up as the sub- scripted variable, but the parse then fails because 'by' cannot be found.
if abc is x-3 or 0 then do;	<pre>if abc = x-3   abc = 0 then do;</pre>	
R = (def is among 1,2,2-4 or 9);	R = (def = 1   def = 2   def = Z-4   def = 9);	
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

input	translation	comments
		expressions. The
		syntax of the
		functional macro
		should be cor-
		rected to prevent
		this error. Notice
		how this error is
		reflected in the
		translation;
if x is $3,>5$ , or 0		">5" is not an
		expression.
if $x = 1$ or 4	if $x = 1   x = 1$	
then i x by x-l;	then $x = x+x-1$ ;	

.

#### VIII. IMPLEMENTATION

The initial implementation of APAREL, which has been completed on an IBM-360 computer, consists of two parts: 1) a preprocessor that converts APAREL programs into equivalent legal PL/I pr grams with external calls for parserequests, and 2) the run-time parser, which provides APAREL's parsing capabilities. The preprocessor is an APAREL program that was bootstrapped into operation, and the run-time parser is an assembly-language program. The current implementation of each of these parts imposes the following restrictions on the full APAREL language.

- The BAL primitive parse-request function is not implemented.
- 2) The scan of parse-requests is strictly left to right. Thus, in the parse-request (A|B)C if A is matched, B will be skipped; if C then

fails, the sequence B followed by C will not be tried. This can be remedied by

(AC BC)

3) The parser matches the maximum string possible. This applies only to the nonliteral matches; e.g., ARBNO and the blank scan, which match as much as possible. Note that this will prevent the parserequest

#### ARBNO (A, ")A

from being parsed successfully because the arbitrary number of A's separated by NULLs will include all such A's in the input, forcing the final A after the APBNO to fail.

-28-

2) the position in the parse-request. Before attempting a match for any alternative, the parser checks to see if the present state has occurred before (during the current initiation of the original parse-request). If it has, a left recursive loop has occurred, and the pars r simply moves on to the next alternative to break this left recursive loop. This, therefore, would cause the rule

number: number digit digit to fail on more than two digit numbers. This can be remedied by using the ARBNO function, which allows iterative specification rather than nested recursive definition. Thus,

number: ARBNO(digit,") A number is an arbitrary nonzero number of digits separated by NULLs. Or even more elegantly:

expression: ARBNO (expression, operator) | (expression)

variable number

unary operator expression

An expression is an arbitrary nonzero number of expressions separated by operators, a parenthesized expression, a variable, a number, or a unary\_operation followed by an expression.

#### Appendix

#### BNF DEFINITION OF APAREL'S SYNTAX LANGUAGE

#### REFERENCES

- <u>PL/I Language Specification</u>, IBM Corporation, form C28-6571-4.
- Balzer, R. M., <u>Dataless Programming</u>, The RAND Corporation, RM-5290-ARPA, July 1967. (Also <u>Proceedings of</u> the AFIPS FJCC (1967), pp. 535-544.)
- Strachey, C. (ed.), <u>CPL Working Papers</u>, London Institute of Computer Science and the University Mathematical Laboratory, Cambridge, 1966.
- Leavenworth, B. M., "Syntax Macros and Extended Translation," <u>Communications of ACM</u>, Vol. 9, No. 11, November 1966, pp. 790-793.
- Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," <u>Proceedings of the International</u> <u>Conference on Information Processing</u>, 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.
- Farber, D. J., R. E. Griswold, and I. P. Polonsky, "SNOBOL3," <u>Bell System Technical Journal</u>, August 1966.
- Feldman, J. A., and D. Gries, "Translator Writing Systems," <u>Technical Report #CS69</u>, Stanford, June 9 1967.
- Gauer, B., and A. J. Perlis, "A Proposal for Definitions in ALGOL," <u>Communications of the ACM</u>, Vol. 10, April 1967, pp. 204-219.
- Irons, E. T., "A Syntax Directed Compiler for ALGOL 60," <u>Communications of the ACM</u>, Vol. 4, January 1961, pp. 51-55.
- 11. McClure, R. M., "TM6--A Syntax-Director Compiler," <u>Pro-</u> <u>ceedings of the 20th National ACM Conference</u>, 1965, pp. 262-274.
- Mondschein, L., <u>VITAL Compiler-Compiler Reference Manual</u>, TN 1967-1, Lincoln Laboratory, January 1967.