UBC CLPARSER

A COMMAND LANGUAGE PARSING FACILITY

by

Alan Ballard

Computing Centre

UNIVERSITY OF BRITISH COLUMBIA
2075 Wesbrook Place
Vancouver, B.C., Canada V6T 1W5

August 1979

Revised November 1982

Acknowledgement

The original version of this program was based in part on the Spires parser and grammar analyzer. See "Design of SPIRES II", published by Spires Project, Stanford University, July 1973. They have been completely rewritten (in PLUS) and extensively modified for use in MTS.

Note

This writeup corresponds to Version 2 of the parser and table generator as implemented in July 1981.

Table of Contents

I. <u>INTRODUCTION</u>

This writeup describes a facility that assists in the development of programs which perform some kind of syntactic analysis of input. Such analysis is typically found in programs which read input commands and must determine the command and its operands.

The facility consists of two parts: a self-contained program called the table generator, and a library of subroutines called the parser.

The table generator processes a description of the command language to be implemented. This description uses a notation which is based on BNF, with some extensions. It is similar to the notation frequently used in the documentation of command prototypes. The complete description is called a <u>grammar</u> and consists of a number of <u>productions</u> describing the elements of the command language. The table generator produces as output an object deck containing syntax tables which are basically a digested form of the grammar. It also produces some PLUS declarations or Assembler equates which may be useful in writing the command-processing program.

The productions define the structure of the language in terms of other productions and basic symbols called <u>terminals</u>. Each production consists of one or more <u>alternates</u>, and each alternate consists of one or more <u>terms</u>. A term may be the name of another production, a <u>literal</u> (keyword), a <u>character class</u> defining an allowable sequence of characters, or a special predefined <u>system terminal</u>. Productions also indicate points at which a user-supplied <u>semantic routine</u> is to be called.

The parser is a subroutine which attempts to parse a given character string by interpreting the tables produced by the table generator. During the process, the parser will call the user's semantic routines as specified in the grammar. A semantic routine performs <u>semantic actions</u> to save and act upon the command elements that are encountered.

The parser, starting from a given <u>goal symbol</u>, attempts to match the input against the grammar definition. Each term may <u>succeed</u> or <u>fail</u>. If a term succeeds, the process continues with the next term of the current alternate. If a term fails, the parser backs over input matched by previous terms of the alternate, and tries again with the next alternate of the production. If all terms of an alternate succeed, the production as a whole succeeds. If no alternate succeeds, then the production fails. During the process, the parser also maintains a <u>semantic stack</u> which contains values corresponding to various terms in the productions processed. Values may be placed on the stack by the parser or by a semantic routine.

To use the parser one must write a grammar (using the table generator’s input language) to describe the syntax of the language to be parsed. One must then write (at least) a main program and semantic routine to perform the required semantic actions. These programs may be written in any language which supports either the PLUS (coding conventions) linkage or the OS/Fortran S-type linkage. The table generator produces extra output (a number of declarations) which are useful when writing the semantic routines in either PLUS or Assembler.

The user’s main program will usually initialize the parser, read the input lines to be processed, and call the parser to analyze each input line. The parser then calls the user’s semantic routine as specified in the grammar. The user’s semantic routine may in turn call other routines (provided as part of the parser) to obtain further information about the elements that have been parsed.

Most programs using the parser take one of two forms. In one form, the semantic routine only sets switches or builds a data structure representing the structure and contents of the input being parsed. Then, when the parser returns, the main program interprets the resulting information to perform the actual semantics of the command. In the other form, the semantic routine is responsible for actually performing the commands as they are parsed. In this case, the main program is usually little more than a loop, reading input and passing it to the parser.

It is also possible to specify, within the grammar, points at which a new line is to be processed. The user’s program specifies a routine which the parser is to call for input, then calls the parser which will request input as necessary. Using the parser in this mode is relatively complicated and is generally not necessary for most applications.

II. <u>THE TABLE GENERATOR</u>

A grammar consists of a series of <u>productions</u> and <u>declarations</u>. The productions and declarations may appear in any order. A production may be a <u>goal production</u>, a <u>nonterminal production</u>, or a <u>terminal production</u>. A declaration may be a <u>semantic procedure declaration</u> or a <u>semantic label declaration</u>.

Each production defines a symbol called the <u>production name</u>. A production name is an arbitrary sequence of alphanumeric characters, surrounded by "<" and ">". Each production is terminated by a semicolon.

A terminal production defines a "character-class" terminal by specifying a set of allowed characters. A nonterminal production defines a construct of the language in terms of other nonterminals and terminals. It may also specify that the production is defined externally, i.e., in another grammar. A goal production is exactly the same as a nonterminal production, except that it defines a top-level construct of the language--a "goal" of the parsing process. When the parser is called, a parameter specifies which goal it is to parse.

The following sections describe the syntax of the table generator's input language. Upper-case letters are used to denote keywords. Appendix A contains a grammar which uses the table generator's input language to describe itself.

A. <u>TERMINAL PRODUCTIONS</u>

A terminal production specifies the production name, the minimum and maximum allowed lengths, and a set of characters to be allowed or excluded in terminals of that type.

It has one of the forms

```
   TERMINAL <symbol> IS length-specification OF strings;
or
   TERMINAL <symbol> IS
      length-specification EXCLUDING strings;
or
   TERMINAL <symbol> IS length-specification CHARACTERS;
```

A length-specification defines an allowable length range. It may be given in any of the forms

```
   n
   n TO m
   n OR MORE
```

where n and m are unsigned integers in the range 0 to 255.

The set of characters allowed or excluded is given by one or more quoted strings and/or hexadecimal strings. A quoted string consists of arbitrary characters surrounded by quotation marks ("). If a quotation mark is to appear in the string, it must be doubled. A hexadecimal string consists of an arbitrary sequence of hexadecimal characters (0-9, A-F) and blanks surrounded by apostrophes.

The third form above is used to match a number of arbitrary characters.

The parser always attempts to match the longest possible sequence of allowed characters, subject to the maximum length restriction.

<u>Examples</u>

1) terminal <Blanks> is 0 or more of " ";

   This matches the longest sequence of blanks.

2) terminal <Sign> is 1 of "+-*/";

   This matches any one character from the set +, -, *, /.

3) terminal <Special> is 1 of '00 FF';

   This matches either of the hexadecimal values 0 or FF.

4) terminal <Nonblank> is 0 or more excluding " ";

   This matches an arbitrary sequence of nonblank characters.

5) terminal <Rem> is 0 or more characters;

   This matches all possible characters; i.e., the remainder of the input line.

B. <u>NONTERMINAL PRODUCTIONS</u>

A nonterminal production specifies the production name, and one or more alternates defining possible forms of the construct. The alternates are separated by the word OR.

Each alternate consists of one or more terms specifying the constituent elements of the alternate.

Thus the overall form of a nonterminal is:

```
<symbol> IS term ...
         OR term ...
         ... ;
```

The terms out of which an alternate is built are the following.

<symbol>	The name of any terminal, nonterminal or goal production, including the surrounding < >. The parser will attempt to match the specified production. This term succeeds if the production specified succeeds.
{expression} or (/expression/)	An arbitrary subexpression consisting of one or more alternates surrounded by braces. This is a nested production definition. The parser will attempt to match the subexpression. The term succeeds if any alternate within the braces succeeds. (/ /) may be used as a substitute for braces.
[expression] or (expression)	An arbitrary subexpression consisting of one or more alternates surrounded by square brackets. This denotes an optional occurrence of the nested definition. The parser will attempt to match the expression, but the term will succeed (and hence continue with the next term of the alternate containing it) regardless of whether the expression succeeds or fails. Parentheses may be used as a substitute for brackets. The subexpression may, of course, be as simple as a single term. For example, [<Sign>] denotes an optional occurrence of <Sign>.
term...	The ellipsis (...) following a term means that it may be repeated. If the term succeeds, then the parser will retry it on the remaining input until it fails. The parser then continues with the next term of the alternate containing this term. If the ellipsis follows an optional term (i.e., [expression]...), then the effect is to match zero or more occurrences of expression; in this case the term... will always succeed. If it follows a nonoptional term (e.g., <symbol>... or {expression}...), then the effect is to match one or more occurrences of the element; the alternate will fail if the

term doesn’t match the first time.

literal-symbol

A literal symbol may be specified as a quoted string (quotation marks within the literal are doubled) or a hexadecimal string (surrounded by apostrophes). This term succeeds if the next characters in the input match the literal.

The parser normally works with an upper-case version of the text being parsed. Hence literal symbols should usually contain upper-case letters only. Case-dependent parsing can be performed by setting the parser variable Case_Convert.

|literal-symbol|

Vertical bars surrounding a literal symbol indicate that an arbitrary initial substring (including zero characters) of the literal may be matched. This term always succeeds, matching the longest substring it can.

This is most often used in conjunction with a literal symbol to indicate allowable abbreviations of keywords in the language being defined. For example "SYS"|"TEMSTATUS"| matches the input SYS followed by any substring of TEMSTATUS.

#symbol

Indicates a point at which a user semantic routine should be called. The grammar must contain one or more procedure declarations (see section E in this chapter) which specify the procedures to be called. The table generator associates a value with each symbol used in a semantic call, and passes this value as a parameter to the semantic routine.

The semantic routine returns a Boolean result indicating whether the term should succeed (True) or fail (False).

symbol(parameters)*
or *symbol*

The symbol specified must be defined as a system terminal name. The parser will attempt to match the specified system terminal. See Chapter IV for a description of currently implemented system terminals.

Certain system terminals expect parameters; these are specified in the

	grammar following the symbol name. Parameters may be integers, quoted character strings, or hexadecimal strings.
	The table generator distinguishes between the use of a symbol as a system terminal name and possible use of the same symbol in a production name or semantic action name. Thus the user need not be aware of exactly what symbols have been defined as system terminals, and new ones can be added without affecting existing grammars.
GO TO <symbol>	This item causes the parser to continue processing from the first term of the first alternate of the specified production (which must be a nonterminal or goal production).
	It will not return to subsequent terms or alternates of the production containing it; therefore it makes sense only as the last term of an alternate.
label:term	Any simple term (not an expression in braces or brackets) may be preceded by a label, which is separated from the term by a colon. A label is an alphanumeric symbol.
	The table generator will associate a value with each label. The label is used as a "tag" to identify the value (if any) left on the semantic stack by the term. A semantic routine may retrieve a stacked value by calling Parse_Get_Stack with the code for the label as one of the parameters.
	The grammar may optionally include one or more label declarations (Section F below) specifying the symbols which are used as labels in productions.
	See Section E in Chapter III for details of the semantic stack.
FAIL	This term causes the production containing it to fail immediately, without attempting other alternates. Subsequent terms of the alternate containing FAIL will not be processed, hence it makes sense only as the last term of an alternate.

FAIL is normally used to negate the result of a preceding item. That is, in the fragment

```
      <term1> FAIL
   OR <term2>
```

if <term1> succeeds, then the production containing this fragment will fail. However, if <term1> fails, the parser will switch to the next alternate (and so the term FAIL is never processed).

Note that, if FAIL is used within a nested production (inside {...} or [...]), then only that sub-production fails; the outer production will continue as it would for any other failing term.

SUCCEED	This term causes the production containing it to succeed immediately. Its only use is as the last alternate of a production, when it is desired that the production should succeed without matching any input. As with FAIL, if SUCCEED is used inside a nested production, it is only the sub-production that succeeds; processing continues in the outer production.
READ	This term is used in parsing multiple-line input. It always fails unless the program has set up an input routine by calling Parse_Set. If an input routine is defined, READ passes to the next line of the input, calling the input routine if the next line has not yet been read. It fails if an end-of-file indication is returned from the input routine. See Section F in Chapter III for further information about input routines and the READ term.
FENCE	This term is used to prevent the parser backing up from the current point in the parse. If a subsequent term in the alternate containing FENCE fails, then the entire parse will fail (returning from the parser) rather than trying other alternates. FENCE also indicates that any values left

on the semantic stack by previous terms, and any preceding input lines, are of no further interest and can be discarded by the parser. FENCE also indicates the point from which parsing should be retried if either CORRECT or RETRY is encountered subsequently.

Whenever a fenced production is completed successfully, the production from which it was called is automatically fenced at that point in the input. This is necessary because the parser has no way of retrying from "inside" a completed production.

RETRY — This term causes the parser to back up and retry the parse from the point of the last FENCE, or from the beginning of the input if there has been no FENCE.

This normally only makes sense if the input has been changed by some semantic action (by calling Parse_Set to change one of the items Current_Production_Text or Terminal_Text). It is intended for use in programming error correction.

CORRECT — This term causes the parser to attempt error correction. It is generally used as the last of a list of alternates. The parser will attempt to replace some part of the current input, then will retry the parse from the beginning, or from the position of the last FENCE.

The error recovery actions are described in Chapter VI. Note that parser variables may be used to disable error correction.

QUIT — This term causes the parser to abandon the current parse and return to the program that called it. QUIT is generally used as the last of a list of alternates, perhaps following a semantic action to issue an error message.

The parse will return failure (i.e., False) if this term is executed.

Examples

1. `<Expression> is <Term> [{"+" or "-"} <Term>]...`

   Defines an "expression" to be a sequence consisting of a "term" followed by zero or more repetitions of (operator, term) pairs, where an operator may be either "+" or "-".

2.
```
<Term> is *Blanks* Expr:*Integer* *Blanks* #Copy_Top
          or *Blanks* "(" Expr:<Expression> ")" *Blanks*
                                                #Copy_Top;
```

   The system terminal *Blanks* matches zero or more blanks, and *Integer* matches an integer. Thus this production matches either an integer or a parenthesized expression, optionally surrounded by blanks.

   In each case, the term corresponding to the value of the expression is labelled with "Expr:", and the semantic action Copy_Top is applied at the end. This will presumably fetch the value of the labelled term, and leave it on top of the semantic stack.

3. `<Command> is "ST"|"OP"| *Blanks* *End_Of_Line*;`

   End_Of_Line succeeds only at the end of the input. Hence this production matches "STOP" (or a substring), followed by zero or more blanks only.

4.
```
<Commmand> is
        "LIST" {" " *Blanks* or *End_Of_Line*} <Rem>;
```

   If <Rem> is defined to match the rest of the input line as in the previous section, then this production matches LIST followed by an optional operand. If the operand is given, then there must be at least one blank separating it from the operand. Note that, if LIST is followed either by blanks only, or by nothing, the string matched by <Rem> will be null.

C. EXTERNAL NONTERMINAL PRODUCTIONS

An alternative form of nonterminal production is used to specify that the production is defined in another grammar.

The external production has the form:

```
   <Symbol> IS EXTERNAL;
or
   <Symbol> IS EXTERNAL extname;
```

If "extname" is given, it specifies the external (loader) name defining the production. It may be specified as either an alphanumeric symbol, or an arbitrary quoted string. If it is not given, then the production name "symbol" is used. If the symbol is longer than eight characters, the external name is generated by taking the first four characters and the last four characters.

The external name must be defined as the entry point name for a goal in some other grammar, as described in the next section.

Example

```
<Command> is external "CMD";
```

The production <Command> is defined in another grammar by a goal production with the entry name "CMD".

D. GOAL PRODUCTIONS

A goal production is just like a nonterminal production, except it is preceded by the word GOAL. Each goal production defines an entry point in the object deck produced by the table generator. A goal name is passed as a parameter when the parser is called, to indicate the starting point for the parse. A goal name may also be used from another grammar to invoke the production as a subgrammar.

The external symbol for the entry point may be given explicitly in the form

```
GOAL <symbol> ENTRY extname IS alternates;
```

where, as before, extname may be an alphanumeric symbol or an arbitrary quoted string. If extname is not given, the production name "symbol" is used. Goal productions may also be used like nonterminal productions within the same grammar, as terms in the definitions of other productions.

Example

```
goal <Command> entry "CMD"
         is "ST"|"OP"| #Stop
         or "HE"|"LP"| *Blanks* *Help_Command*;
```

defines <Command> to be the goal, with external symbol CMD. A command may be either STOP or HELP followed by a help command.

E. SEMANTIC PROCEDURE DECLARATIONS

A procedure declaration is used to specify what semantic procedure is to be called for the various semantic actions used in the grammar.

There are two forms of procedure declaration:

```
   PROCEDURE extname FOR ALL;
or
   PROCEDURE extname FOR symbol-list;
```

In each case, extname is the external (loader) symbol for the semantic routine to be called. It may be either an alphanumeric symbol, or a quoted string. If it is longer than eight characters, the table generator will form an external name by taking the first four and last four characters.

The first form specifies that the procedure is to be called for all semantic actions used in the grammar. If this appears in the grammar, then it must be the only procedure declaration; the second form may not also be used.

The second form specifies a list (separated by commas) of symbols that are used as semantic action names in the grammar. The specified procedure will be called for each action in the list. There may be any number of such declarations in the grammar. However, if this form is used, _every_ semantic action must be named in some procedure declaration. A semantic action may only appear in one list.

One of the parameters of a call to a semantic procedure is a code for the specific semantic action to be performed. The table generator will associate a code with each semantic action name. If the first form of declaration is used, the codes will be assigned in alphabetical order of the symbols, starting from zero. If the second form is used, the codes are assigned in the order the symbols appear in the symbol list, starting from zero for the first symbol in each list.

The table generator will also produce PLUS declarations or Assembler equates specifying the codes assigned. These can be incorporated into the semantic routine. For languages other than PLUS or Assembler, it is advisable to specify all semantic action names in a declaration, so that the association between symbols and values is fixed by the person writing the grammar and doesn't change if the grammar is modified.

By default, the parser will perform an R-type call to the semantic routine, with the parameters contained in registers. A Fortran-compatible S-type call may be requested in a procedure declaration:

```
   PROCEDURE extname S-TYPE FOR ...
```

R-TYPE may also be specified explicitly if desired.

Example

```
   procedure Cmdsem for Mts_Command, Stop_Command,
                                            Print_Result;
   procedure Exprsem for Operation, Copy_Top;
```

The procedure Cmdsem will be called for semantic actions Mts_Command, Stop_Command and Print_Result in the grammar. The codes passed for the semantic actions will be 0, 1, and 2 respectively.

The procedure Exprsem will be called for actions Operation and Copy_Top, with codes 0 and 1.

F. SEMANTIC LABEL DECLARATIONS

A grammar may also include an optional label declaration:

```
   LABEL symbol-list;
```

where "symbol-list" is a list of symbols, separated by commas, that are used as semantic labels in the grammar.

As with semantic action symbols, the table generator assigns a code to each label. If the label declaration appears, these codes will be assigned, in the order that the symbols appear in the list. Labels which are not defined in a LABEL declaration are assigned codes in alphabetical order.

The table generator will produce PLUS declarations or Assembler equates defining the codes assigned. The use of a label declaration is recommended if the parser is used from other languages, since it fixes the association between symbols and the codes used.

Example

```
   label Expr, Opnd, Op;
```

defines the three symbols to be labels which should be assigned codes of 0, 1, and 2 respectively.

III. THE PARSER

A. PARSER CONTROL BLOCK

Information about the current state of the parse is maintained in a data structure called a Parser Control Block. The routine Parse_Initialize or Parse_Initialize_Nonmts allocates and initializes a Parser Control Block, and returns its address to the caller. This address is then passed to the parser and other parser routines to provide the required link between them. By calling the initialization routine multiple times, it is possible to have a number of completely independent parsing processes going on simultaneously.

A user’s semantic routine should not examine the Parser Control Block in any way[1] However, much of the information contained in it may be accessed (changed or retrieved) by calling the routines Parse_Set and Parse_Get. These routines take as a parameter a code for the item to be accessed. The descriptions of parser variables in Chapter V give the symbolic names that would be used from a PLUS program, and the equivalent numeric values that would be used from Assembler or other languages.

B. PARSER SUBROUTINE DESCRIPTIONS

This section describes the subroutines provided by the parser. Appendix B contains a number of examples illustrating how the routines are used from PLUS or Fortran programs.

Most of the subroutines have two entry points. One provides a standard OS/Fortran subroutine linkage, with an S-type parameter list. The second entry point provides for the subroutine linkage used by PLUS and the Assembler coding conventions used internally in MTS. These entries use R-type parameter passing. Details of the linkage and parameter passing for all routines are given by Appendix D.

The subroutine descriptions refer to the routines by descriptive names rather than the actual external symbols used in linking to the subroutines. The external symbol names for the two entry points are given in each subroutine description. The descriptive names would be used by a PLUS program which calls the subroutines. (Declarations for these names are contained in a source library for using the parser from PLUS.)

----
[1] Although for debugging purposes, it may be useful to know that the first eight bytes of a Parser Control Block always contain "PARSERCB", and the 17th byte contains the trace flag.

1. Parse Initialize

Purpose:	Allocates and initializes a Parser Control Block.
Parameters:	One only. An arbitrary "user psect pointer" which is passed to the semantic procedures called from the parser.
Result:	The address of the allocated and initialized Parser Control Block.
External Names:	R-type entry: CPINITCC S-type entry: CPINIT
Description:	This routine allocates a Parser Control Block and initializes it.

2. Parse Initialize Nonmts

Purpose:	Allocates and initializes a Parser Control Block without requiring the MTS environment.
Parameters:	1) An arbitrary "user psect pointer" which is passed to the semantic procedures called from the parser. 2) A routine that the parser may call to dynamically allocate storage. 3) A routine that the parser may call to release storage.
Result:	The address of the allocated and initialized Parser Control Block.
External Names:	R-type entry: CPINMCC S-type entry: CPINM
Description:	This routine allocates a Parser Control Block and initializes it. Parameters specify getspace/freespace routines that may be used by this call, or by subsequent calls to any of the parser routines. When Parse_Initialize_Nonmts is used, error correction and spelling correction are not available, unless other system services are also supplied. The system services used by the parser are described in Section D of this chapter.

3. Parse

Purpose:	To parse a given sequence of characters.
Parameters:	1) The address of the Parser Control Block (as returned from Parse_Initialize). 2) The external name of the parsing goal. 3) The address of the first byte of the input to be parsed. 4) The number of bytes, starting at the address given, that are to be parsed.
Result:	A Boolean. True if the parse succeeded; False if it failed.
External Names:	R-type entry: CPARSECC S-type entry: CPARSE
Description:	This routine parses the given string, starting from the specified goal. The goal determines which syntax table gets used, which in turn determines which semantic routines are called during the parsing process. The routine returns True if the specified goal was successfully parsed, False otherwise. Whenever False is returned, an error code and error message will have been set as the values of Last_Error_Code and Last_Error_Message. These may be retrieved by calling Parse_Get. The possible codes and corresponding messages are listed in Appendix C. If a null pointer is passed as the third parameter, the parser will initially begin parsing with a null string (length 0) input. (The fourth parameter can be omitted in this case.) This is generally only useful in grammars that use the READ term to obtain more input. For such grammars, an input subroutine must be specified with a call to Parse_Set. The parser copies the input specified by parameters 3 and 4. Hence the user program does not need to preserve these locations during the entire parsing process.

Parse normally works with an upper case version of the text passed, so that the parsing is done on a case-independent basis. The semantic routines can obtain either the upper-case or the original case versions of the parsed items. Parse_Set can be used to set the variable Case_Convert if it is desired to parse the original, unconverted text.

The parser may be reentered recursively from a semantic routine with a new input to parse, using the same parser control block. The state of the previous parse will be stacked, and will resume when the semantic routine returns. The second call may leave a value on the semantic stack which becomes the result left by the semantic action.

4. <u>Parse Terminate</u>

Purpose:	Releases a Parser Control Block and associated storage.
Parameters:	One only. The address of the Parser Control Block.
Result:	A Boolean. False if any problems were encountered.
External Names:	R-type entry: CPTERMCC S-type entry: CPTERM
Description:	This routine releases the parser control block and other associated storage, and performs any other required cleaning up.

5. <u>Parse Set</u>

Purpose:	To set various parser variables.
Parameters:	1) The address of the Parser Control Block. 2) A code specifying the item to be set. From a PLUS program this will be a value of type Parse_Item_Type. From other languages, it will be an appropriate numeric code. 3) A variable containing the new value. In a PLUS program, this is specified as a

"reference value parameter", allowing the use of either a variable or a constant.

4) For certain codes, a fourth parameter is required. This specifies the number of bytes represented by parameter 3. Currently, this is only used when setting the variable Semantic_Result.

Result:	A Boolean. False if the specified item is one that cannot be set.
External Names:	R-type entry: CPSETCC S-type entry: CPSET
Description:	This routine is used to set parser variables. It may be called before beginning parsing, or from a semantic routine during the parsing process. The list of items that may be specified is described in Chapter V. Note that the type and length of the third parameter variable depend on which item is being set. Some care must be taken since this procedure is unable to do much checking. It is important to be sure that the correct number of bytes are passed, and that the correct reference level is used. The requirements are described in Chapter V.

6. Parse Get

Purpose:	To get the current value of various parser variables.
Parameters:	1) The address of the Parser Control Block. 2) A code specifying the item to be returned. From a PLUS program this will be a value of type Parse_Item_Type. From other languages, it will be an appropriate numeric code. 3) A variable to be assigned the value. 4) The number of bytes of memory at the location specified by parameter 3. The parser will modify at most the specified number of bytes.

5) Certain parser variables, representing portions of the input text, may be returned in either the original mixed-case form, or an upper-case-only form. Parameter 5 is required only for these items. It should be a Boolean (Fortran LOGICAL*4) True to return the upper-case form, False for the original form.

The descriptions of the parser variables in Chapter V indicate whether this parameter is required.

Result: An integer, specifying the actual number of bytes required for the item requested. A value of -1 means the requested item is unavailable.

If the value returned is bigger than the integer given as parameter 4, then the returned value will have been truncated.

External Names: R-type entry: CPGETCC
S-type entry: CPGET

Description: This routine may be called before beginning parsing, or from a semantic routine during a parse. It is used to access items from the parser.

The list of items that may be specified is described in Chapter V. They include various options that may be set by the user, the values of system terminals that have been processed, and substrings of the input text corresponding to the current state of the parse. Some items can only be accessed when Parse_Get is called from a semantic routine (e.g., the text corresponding to the current production). Items corresponding to system terminals can only be accessed if there is an appropriate element on the semantic stack.

Note that the type of variable expected for the third parameter depends on which item is being requested. Some care is required, since the procedure is unable to do much checking. The fourth parameter to Parse_Get specifies the number of bytes of memory occupied by the third parameter, as protection against unintentionally overwriting other storage.

7. Parse Get Stack

Purpose: To retrieve a value that has been left on the semantic stack by a previous term of the current production.

Parameters: 1) The address of the Parser Control Block.

2) A fullword numeric code for the label of the stack element to be returned. For PLUS or Assembler programs, symbolic declarations for the labels used in the grammar will be produced by the table generator.

3) A variable to be assigned the value.

4) The number of bytes of memory at the location specified by parameter 3. The parser will modify at most the specified number of bytes.

5) This parameter is required only when the stack element to be returned is a portion of the input text. This occurs when the stack entry corresponds to a terminal or a literal in the grammar. These values may be returned in either the original mixed-case form, or an upper-case-only form. Parameter 5 should then be a Boolean (Fortran LOGICAL*4) True to return the upper-case form, False for the original form.

Result: An integer, specifying the actual number of bytes required for the item requested. A value of -1 means there is no element with the specified label among the stack elements for the current production.

If the value returned is bigger than the integer given as parameter 4, then the returned value will have been truncated.

External Names: R-type entry: CPGSTKCC
S-type entry: CPGSTK

Description: This routine may be called only from a semantic routine during a parse. It is used to access values left on the semantic stack during parsing.

The routine searches the values left on the stack by preceding terms of the current production, starting from the leftmost term. It returns the first stack entry found with the label specifed by parameter 2. The entry found is <u>removed</u> from the stack, so that it will not be found by subsequent calls to Parse_Get_Stack with the same label code.

See Section E in this chapter for further information about the operation of the semantic stack, and the interaction of the grammar and semantic routines.

Note that the type of variable expected for the third parameter depends on what is on the stack at the position requested. Some care is required, since the procedure is unable to do much checking. The fourth parameter to Parse_Get_Stack specifies the number of bytes of memory occupied by the third parameter, as protection against unintentionally overwriting other storage. The routine Parse_Get_Stack_Size can be used to determine the number of bytes required, before calling Parse_Get_Stack.

8. <u>Parse Get Stack Size</u>

Purpose:	To determine the size (in bytes) of a value that has been left on the semantic stack by a previous term of the current production.
Parameters:	1) The address of the Parser Control Block.
	2) A fullword numeric code for the label of the stack element to be returned. For PLUS or Assembler programs, symbolic declarations for the labels used in the grammar will be produced by the table generator.
Result:	An integer, specifying the number of bytes required for the item requested. A value of -1 means there is no element with the specified label among the stack elements for the current production.

External Names:	R-type entry: CPGSSCC S-type entry: CPGSS
Description:	This routine may be called only from a semantic routine during a parse. It is used to determine the size of a value left on the semantic stack during parsing. The routine searches the values left on the stack by preceding terms of the current production, starting from the leftmost term. It returns the size of the first stack entry found with the label specifed by parameter 2. The entry found is <u>not</u> removed from the stack. Thus Parse_Get_Stack_Size indicates the number of bytes required for the value that would be returned by a call to Parse_Get_Stack.

9. <u>Parse Reset</u>

Purpose:	Resets a Parser Control Block and associated memory to indicate no parse is in progress.
Parameters:	One only. The address of the Parser Control Block.
Result:	A Boolean. False if any problems were encountered.
External Names:	R-type entry: CPRSETCC S-type entry: CPRSET
Description:	This routine must be called if, for any reason, the parsing process is interrupted and not allowed to return normally. For example, if an attention-handling routine wishes to abort the parser and branch somewhere else, it should call Parse_Reset to reset the Parser Control Block to the initial state with no parse in progress Parse_Reset also does some cleaning up, so it may be good practice to call it occasionally in other situations. Various data structures in the parser are allowed to grow dynamically as necessary during parsing. Parse_Reset shrinks these structures back to the original size. It also releases the Fdub acquired for the help file if any HELP requests have been processed.

Note that, if the parser is being used recursively, Parse_Reset will clear the state of all parses currently in progress (using the same Parse_Control_Block).

10. Current Position

Purpose:	To access the current scan position.
Parameters:	One only. The address of the Parser Control Block.
Result:	A fullword integer specifying the scan position, 0-relative.
External Names:	R-type entry: CPCURPCC S-type entry: CPCURP
Description:	This routine may be called from a semantic routine to determine the position of the next character to be matched by the parser. It returns the offset from the beginning of the input string. The routine is equivalent to calling Parse_Get for the variable Scan_Position.

11. Command Text

Purpose:	To retrieve the text being parsed.
Parameters:	1) The address of the Parser Control Block. 2) A Boolean. True means the upper-case version should be returned; False means the original, unmodified version.
Result:	A character string containing the text being parsed. See Appendix D for calling sequence details.
External Names:	R-type entry: CPCTXTCC S-type entry: none
Description:	This routine may be called from a semantic routine to retrieve the parser text. Use of this routine is similar to calling Parse_Get for the item Input_Text. However, Command_Text will return at most 255 bytes.

Note there is no S-type (Fortran-callable) version of this routine.

12. Production Text

Purpose:	To retrieve the text corresponding to the current production.
Parameters:	1) The address of the Parser Control Block. 2) A Boolean. True means the upper-case version should be returned; False means the original, unmodified version.
Result:	A character string containing the text. See Appendix D for calling sequence details.
External Names:	R-type entry: CPPTXTCC S-type entry: none
Description:	This routine may be called from a semantic routine to retrieve the portion of the input text corresponding to the production currently being performed. That is, it returns the text from the scan position when the production was started up to the current scan position. Use of this routine is similar to calling Parse_Get for Current_Production_Text, except that at most 255 bytes will be returned. Note there is no S-type (Fortran-callable) version of this routine.

13. Last Terminal Text

Purpose:	To retrieve the text for the last terminal that was successfully scanned.
Parameters:	1) The address of the Parser Control Block. 2) A Boolean. True means the upper-case version should be returned; False means the original, unmodified version.

Result: A character string containing the text. See Appendix D for calling sequence details.

External Names: R-type entry: CPLTTCC
S-type entry: none

Description: This routine may be called from a semantic routine to retrieve the value of the last character-class terminal that was scanned.

Use of this routine is similar to calling Parse_Get for the item Terminal_Text, except that at most 255 bytes will be returned.

Note there is no S-type (Fortran-callable) version of this routine.

14. Parse Help

Purpose: To print an entry from the help file.

Parameters: 1) The address of the Parser Control Block.

2) A character string (in the form of a halfword length followed by the specified number of characters), to be looked up in the help file.

Result: A Boolean. True if the specified string was found in the help file directory and the corresponding member printed; False if it was not found.

External Names: R-type entry: CPHELPCC
S-type entry: CPHELP

Description: This routine may be called to print out an entry from the help file. The help file must have been previously specified by calling Parse_Set to set the parser variable Help_File_Name.

See Chapter VII for details of help file format and processing.

C. USER-SUPPLIED SUBROUTINES

This section describes the routines called by the parser, but written by the user. Every program will have one or more semantic routines. The attention test routine and parser input routine are only required for some applications.

1. Semantic Routines

Purpose:	To perform requested semantic actions encountered during the parse.
Parameters:	1) The address of the Parser Control Block. 2) The user psect pointer specified in the call to Parse_Initialize. (Or as set by a subsequent call to Parse_Set.) 3) A fullword containing a code for the semantic action to be performed.
Result:	A Boolean. False if the semantic action has failed.
Description:	This routine performs any desired processing to save or act upon the grammar elements that have been parsed. It is passed, as parameters: a code for the semantic action to be performed; the Parser Control Block address; and the user psect pointer. (The values used for semantic actions are defined by the declarations produced by the table generator.) The value returned determines whether the parse continues with the current alternate (True), or fails and tries the next alternate (False). A semantic routine can call Parse_Get or Parse_Get_Stack to obtain information about the elements that have been scanned. The user psect pointer can be used to pass the address of a storage area to the semantic routines. This may be needed to record the various components of a command as they are identified, to communicate between semantic actions, etc. In many situations, a program will just use global storage for this, and will not need to use the psect pointer. Semantic routines may also communicate by saving values on the semantic stack for retrieving during later semantic actions. See Section E in this chapter.

Note that semantic procedures may be called with either an S-type or R-type parameter list, as specified in the grammar. Linkage details are given in Appendix D.

2. The Attention-Testing Routine

Purpose:	To test whether an attention interrupt has occurred.
Parameters:	One only. The user psect pointer specified in the call to Parse_Initialize. (Or as set by a subsequent call to Parse_Set.)
Result:	A Boolean. True if an attention has occurred; False if not.
Description:	The user program may provide an attention-testing routine by calling Parse_Set to set the parser variable Attention_Test_Routine. If such a routine has been provided, the parser will call it at appropriate points during error recovery, help file processing, and READ processing, to determine whether the attention key has been pressed.

3. The Parser Input Routine

Purpose:	To obtain another input line for a READ term in the grammar.
Parameters:	1) The address of the Parser Control Block. 2) The user psect pointer specified in the call to Parse_Initialize. (Or as set by a subsequent call to Parse_Set.) 3) A buffer into which the input should be read. 4) The length of the buffer provided by parameter 3.
Result:	An integer specifying the actual length of the next input record. -1 should be returned if end-of-file has been received.

Description: The user program may provide an input routine by calling Parse_Set to set the parser variable Input_Routine.

If such a routine has been provided, the parser will call it when input lines are required as a result of READ terms in the grammar.

The parser provides a buffer into which the input line should be read. The length of this buffer can be controlled by setting the parser variable Input_Buffer_Length via a call to Parse_Set. The buffer passed to the input routine will always be at least that long. The default value is 255.

If the buffer provided for a call to read is not big enough (perhaps as a result of passing to another input file or device with longer records), the input routine can indicate this by returning the length required. In this case, the parser will <u>repeat</u> the call to the input routine with a new, bigger buffer. If the input routine wants subsequent calls to also provide the larger buffers, it should reset Input_Buffer_Length.

See Section F in this chapter for further explanation of the use of parser input routines.

D. <u>SYSTEM SERVICES USED</u>

The parser uses a small number of system services during its processing. Normally, regular MTS system subroutines are used. However, it is possible to replace any of the routines with functionally equivalent ones. Options to the Parse_Get subroutine can be used to specify the routines to be called.

When the parser is initialized with the Parse_Initialize_Nonmts routine, substitute getspace and freespace routines are provided as parameters. The other routines are set to Null, indicating the functions they are used for are not available. For the most part, other subroutines are required only for error correction and help processing.

The following descriptions specify the system subroutines called by the parser, and the purposes for which they are required. For details of the calling sequences of system subroutines, see the MTS system subroutine descriptions.

1. Getspace

   This is used to allocate storage as required by the parser. It is the only system subroutine that is absolutely necessary when using the parser.

   If a routine other than MTS's Getspace is to be used, it is normally specified by using the non-MTS initialization routine, and passing the Getspace substitute as a parameter. The routine to call can also be changed with Parse_Set by setting the parser variable Getspace_Routine.

   The parser will pass a value of zero for the first parameter on all calls to the getspace routine.

   The MTS Getspace routine does not require that R13 point to a save area. However, all calls from the parser will provide a save area (in fact a stack) in R13, except if the S-type initialization entry CPINM is used.

2. Freespace

   This is used to release space formerly allocated by calling Getspace.

   If a routine other than MTS's Freespac is to be used, it is normally specified by using the non-MTS initialization routine, and passing the Freespac substitute as a parameter. The routine to call can also be changed with Parse_Set by setting the parser variable Freespace_Routine.

   The parser will pass a value of zero for the first parameter on all calls to the freespace routine.

   If Null is specified as a freespace routine, the parser will function correctly, but will be unable to release any space it acquires.

   The MTS Freespac routine does not require that R13 point to a save area. However, all calls from the parser will provide a save area (in fact a stack) in R13, except if the S-type initialization entry CPINM is used.

3. Guser

   Guser is used only to get confirmations or replacement strings during error correction. Hence it is required only for grammars that include CORRECT terms.

4. Sercom

   Sercom is used for issuing prompts during error correction, and for printing help information.

   Certain serious error conditions also result in a message to Sercom. However, if Sercom is not defined, the parser will continue without issuing these messages (but successful continuation may be doubtful).

5. Guinfo

   Guinfo is used to obtain the prefix string during prompts issued for error recovery.

   If Guinfo isn’t available, operation will continue normally without changing the prefix.

6. Cuinfo

   Cuinfo is used to change the prefix string during prompts issued for error recovery.

   If Cuinfo isn’t available, operation will continue normally without changing the prefix.

7. Getfd

   Getfd is used to obtain an Fdub for the help file during help processing. It is also used to get an Fdub for parsed Fdname system terminals, if Parse_Get is called with the option Parsed_Fdub.

   If the routine is not available, then help processing and Parsed_Fdub will not be available.

8. Freefd

   Freefd is used to free the Fdub obtained for the help file. If it is not available, the parser will continue normally, but Fdubs will never be released.

9. Read

   Read is used to read lines of the help file. If it is not available, then Help cannot be used.

E. THE SEMANTIC STACK

While parsing the input text, the parser maintains a stack of values associated with terms from the input. This stack is known as the "semantic stack". The parser subroutine Parse_Get_Stack can be used to retrieve values from the stack. Certain cases of Parse_Get also access the stack.

Various terms in a grammar may put a value on the stack, as follows:

1. Every terminal and literal stacks the text that it matches.

2. System terminals may place various values on the stack. The descriptions of the system terminals in Chapter IV state what, if anything, is put on the stack.

3. A semantic action may put an arbitrary value on the stack by calling Parse_Set specifying one of the items:

   Semantic_Result, Semantic_Result_Word, or
   Semantic_Result_String.

When a nonterminal production finishes, all the values stacked by earlier terms in the production are popped off the stack, then the last value is returned to the stack. Thus, generally, the values stacked during a production must be retrieved and processed by semantic actions later in the same production, except that the last value placed on the stack is returned as a "result" to the higher-level production. If no values were left by terms in the production, then no result is returned.

The stack is not popped in this way when an internal nested production (a grammar expression surrounded by braces or brackets), is completed. Values left by terms inside the expression remain until the end of the enclosing production.

Any term in a grammar may be labelled. The table generator associates a code with each symbol used as a label. A label code can be used as a parameter to Parse_Get_Stack in order to retrieve the value that was stacked by the term having that label.

The same label may be used on more than one term in the grammar. Moreover, when a term is repeated (followed by "...") each successful repetition may leave a value with the same label. When Parse_Get_Stack is called, it searchs through all values left by terms from the current production. This search is from left-to-right in the production. It will return the first found with the specified label. The element will then be removed from the semantic stack, so that successive calls with the same label will access different elements.

Following the return from the parser, the value (if any) of the goal production remains on the stack. It may be retrieved by calling Parse_Get with the option Parse_Result.

Note that, if a FENCE term in the grammar is processed, all stacked semantic values (from the current and higher level productions) are popped off the stack.

Example

```
<Expr> is Opnd:<Term> [Op:<Addop> Opnd:<Term>
                              Opnd:#Do_Operation]...;
<Addop> is "+" or "-";
```

These productions might be used in implementing arithmetic expressions.

<Addop> leaves either a "+" or a "-" on the semantic stack, which is passed back as the value of the <Addop> production. The label Op can be used from the semantic routine for Do_Operation to retrieve the value.

Note that the production <Expression> could not contain

```
--- Op:{"+" or "-"} ---
```

since subexpressions cannot be labelled. It could contain

```
--- {Op:"+" or Op:"-"} ---
```

with equivalent effect if desired.

The semantic routine for Do_Operation will call Parse_Get_Stack for two entries with label Opnd, and one with label Op. It will perform the requested operation, then return the result to the semantic stack. Since the semantic call is itself labelled Opnd, it will serve as one of the <Term>'s to be retrieved during the next iteration if any.

See the examples in Appendix B for further details.

F. MULTIPLE-LINE INPUT PROCESSING

Normal use of the parser is to process a single line of input, which is specified by parameters to Parse.

It is also possible to process multiline inputs. To do so, the grammar must specify the points at which line transitions may occur, and the program must set up an input subroutine that the parser can call to obtain successive lines of the input.

The input subroutine is specified by calling Parse_Set for the option Input_Routine. The input routine functioning is described in Section C-3 of this chapter. Note that the length of the buffer provided is controlled by the Parse_Set option Input_Buffer_Length.

A transition to a new input line is indicated by READ in the grammar. Note that this can be used in conjunction with other terms to produce quite flexible free-format parsing. The system terminals *More* and *To_Nonblank* are useful for constructing grammars that pass across multiple input lines.

For example,

```
   "A" *To_Nonblank* "=" *To_Nonblank* "B"
```

would match inputs of the form "A = B", with arbitrary interleaved blanks and crossing over an arbitrary number of input lines.

Since an alternate in a production may fail, causing the parse to back up, it is necessary for the parser to buffer input lines which it has read so it can return to them if necessary. The term FENCE in the grammar instructs the parser not to back up past the point where it occurs. The parser will then release any buffered lines that have been completely processed. Grammars which may process long sequences of input lines in this way, should generally include FENCE at appropriate points to avoid using unnecessarily large amounts of memory. Note that FENCE also causes all previous values from the semantic stack to be discarded.

IV. SYSTEM TERMINALS

System terminals are predefined syntactic items for various useful command language elements. Most system terminals consist of a syntactic definition which matches some sequence of input characters, and a semantic action which produces a value from the matched characters. Values produced by system terminals are always left on the semantic stack. For each such system terminal, there is an option of Parse_Get that returns the value of the last one on the semantic stack. As with all other terms that leave values on the semantic stack, the system terminal may be labelled in the grammar so that Parse_Get_Stack can be used to access the value.

Some system terminals require one or more parameters, which are specified in the grammar in parentheses following the terminal name. Parameters may be integers or quoted character strings, depending on the particular terminal. A hexadecimal string, surrounded by apostrophes, may be used as an alternative way of specifying a character string.

Some system terminals may fail because of semantic restrictions. For example, *Integer* must not only be syntactically valid (an optional sign followed by a sequence of digits), but must also evaluate to an integer within the allowed range of the machine. If it does not, the terminal will fail. In such cases, an error message and error code will be set by the system terminal and may be retrieved by calling Parse_Get. Semantic restrictions and the resulting error codes and error messages are indicated in the following descriptions.

The currently defined system terminals are:

Blanks	matches an arbitrary (possibly null) string of blanks. It always succeeds, and leaves nothing on the semantic stack.
Ccid	matches an MTS CCID. For the purposes of this terminal, an MTS CCID is considered to be 1 to 4 arbitrary characters, excluding any of the following: blank , ; ( ) ’ " : = The result is padded to four characters, and left on the semantic stack. It may be accessed by Parse_Get by requesting the value of Parsed_Ccid.

End_Of_File is intended for use with grammars that READ input. It succeeds if the current input position is at the end of the last line of the input. (That is, if the input routine has returned end-of-file, and the parse has reached the end of the last line.) It leaves nothing on the semantic stack.

End_Of_Line succeeds if the current scan position is the end of the input line, and fails otherwise. Nothing is left on the semantic stack.

Fdname matches an arbitrary MTS file or device name.

The result is left on the semantic stack, with one trailing blank appended. It may be retrieved by Parse_Get as the item Parsed_Fdname. The Parse_Get item Parsed_Fdub may be used to obtain an Fdub for the last parsed Fdname.

Help(item-name) may be used in implementing a simple help facility. In order to use this, a help file must first be specified by setting the value of "Help_File_Name" via a call to Parse_Set.

This system terminal takes a single parameter, which must be a character string. The value of the string is a name which should be defined in the help file directory. The corresponding lines of the file are then listed. See Chapter VII for details.

This system terminal always succeeds. No input characters are matched, and no value is left on the semantic stack.

Help_Command may be used in implementing a simple help facility. In order to use this, a help file must first be specified by setting the value of "Help_File_Name" via a call to Parse_Set.

This system terminal skips over blanks, then matches an arbitrary sequence of nonblank characters. The characters matched represent the name of the "help item" to be looked up in the help file. The corresponding lines of the file are then listed. See Chapter VII for details.

It always succeeds, and leaves nothing on the semantic stack.

Hex_Integer matches a sequence of one to eight hexadecimal characters (0-9, A-F). It converts the value to a binary integer which is left on the semantic stack. It may be retrieved with the Parse_Get option Parsed_Integer.

Hex_String matches a sequence of hexadecimal characters (0-9, A-F) with possible interspersed blanks and commas. It converts the value to a character string which is left on the semantic stack. It may be retrieved with the Parse_Get option Parsed_String.

Note: This terminal does not assume any particular delimiter for the hex characters. Grammars using this terminal will often want to specify delimiters. For example:

```
   <Hex> is "X'" *Hex_String* "'"
```

matches hexadecimal strings of the form

```
   X'cccc,cccc'
```

etc.

Integer matches a signed decimal integer. The value of the integer is left on the semantic stack, and may be accessed with the Parse_Get item Parsed_Integer.

If the value of the integer will not fit in a fullword, it will fail, after setting Last_Error_Code and Last_Error_Message as follows:

```
   10  "Integer out of range"
```

Line_Number matches an MTS line number. The internal form of the line number is left on the semantic stack, and is available from Parse_Get as the value of Parsed_Line_Number.

If the value of the line number is invalid, the terminal will fail, after setting Last_Error_Code and Last_Error_Message as follows:

11 "Line number has too many fractional digits"

At most three are allowed.

12 "Line number out of range"

The converted value will not fit in a fullword. (Note that this terminal does not restrict line numbers to the range ±99999.999.)

Message(string,...)	may be used to issue a simple message from within a grammar. It takes zero or more parameters, each a quoted character string. The strings are written to SERCOM. It matches no input, and leaves nothing on the semantic stack. It will fail if SERCOM is not available, but otherwise always succeeds.
More	is intended for use with multiline input. It moves to the next input line if the parse is currently positioned at the end of the current line (calling the input subroutine if the line has not already been read). No input characters are matched. It fails if positioned at end-of-file. Nothing is left on the semantic stack.
Primed_String	matches a character string delimited by apostrophes ('). An apostrophe within the string is represented by ''. The value of the string (with delimiters removed) is left on the semantic stack, and is available with the Parse_Get option Parsed_String. The string may be up to 32767 characters long. If it is longer, the semantic will fail, after setting Last_Error_Code and Last_Error_Message as follows: 13 "String longer than 32767 characters"
Project	matches an MTS project code (department code). This is identical to *Ccid*, except that project codes are padded with blanks. The result is left on the semantic stack, and may be accessed with the Parse_Get item Parsed_Project.

Push(constant) just leaves a value on the semantic stack. It takes a single parameter, which may be a quoted string, a hexadecimal string, or an integer.

The system terminal always succeeds, and no input is matched. The specified constant is left on the semantic stack. It may be retrieved with the Parse_Get option Semantic_Result_Word (if the parameter is an integer) or Semantic_Result_String (if the parameter is a string).

Qualified_Fdname matches a file or device name with optional modifiers or line number range, but no concatenations.

The result is left on the semantic stack, with one trailing blank appended. It may be retrieved by Parse_Get as the item Parsed_Fdname. The Parse_Get item Parsed_Fdub may be used to obtain an Fdub for the last parsed Fdname.

Quoted_String matches a character string delimited by quotation marks ("). A quotation mark within the string is represented by "". The value of the string (with delimiters removed) is left on the semantic stack, and is available with the Parse_Get option Parsed_String.

The string may be up to 32767 characters long. If it is longer, the semantic will fail, after setting Last_Error_Code and Last_Error_Message to:

```
13  "String longer than 32767 characters"
```

Real matches a "real number" in the form

```
±digits.digitsE±digits
```

where "digits" represents decimal digits, and most of the pieces are optional. The number must contain at least one digit and either a decimal point or an exponent to be valid. There may be up to 35 decimal digits. The number is converted to an extended-precision floating-point number.

The converted value is saved on the semantic stack, and can be accessed by the Parse_Get item Parsed_Real. The value returned by either Parse_Get or Parse_Get_Stack will be an extended-precision number if a 16-byte return area is specified. It will be rounded to double precision if an 8-byte area is provided, and rounded to single precision if a 4-byte area is given.

The system terminal will fail if the number is syntactically valid, but is outside the possible range of the machine. It will set Last_Error_Code and Last_Error_Message as follows:

    14 "REAL number out of range"

Simple_Fdname matches a simple unqualified file or device name (no modifiers, no line number range, no concatenation).

The result is left on the semantic stack, with one trailing blank appended. It may be retrieved by Parse_Get as the item Parsed_Fdname. The Parse_Get item Parsed_Fdub may be used to obtain an Fdub for the last parsed Fdname.

String matches a character string delimited by either apostrophes or quotation marks. Any occurrence of the delimiter within the string must be doubled. The value of the string (with delimiters removed) is left on the semantic stack, and is available with the Parse_Get option Parsed_String.

The string may be up to 32767 characters long. If it is longer, the semantic will fail, after setting Last_Error_Code and Last_Error_Message as follows:

    13 "String longer than 32767 characters"

To_Nonblank is intended for use with multiline input. It skips to the next nonblank character, passing to new input lines as necessary.

It always succeeds. Nothing is left on the semantic stack.

Trace(integer) can be used to enable or disable the parser tracing from within a grammar. It requires a single parameter which must be an integer 0, 1 or 2, and sets the Parse_Set option Parse_Trace accordingly. This can be used to easily implement a trace command for use while debugging the program.

It matches no input characters, and leaves nothing on the semantic stack.

V. PARSER VARIABLES

This section describes the various parser variables which may be accessed by means of routines Parse_Set and Parse_Get. Most can both be set and retrieved. However, some variables can be retrieved but may not be changed, and some can be accessed only from a semantic routine. Variables corresponding to system terminals may be accessed only if there is a value of the appropriate type on the semantic stack. Parse_Get returns a result of -1 if the item requested is not one that can be retrieved.

The third parameter to both Parse_Get and Parse_Set is of various types, depending on the particular parser variable being requested. Note that Parse_Get requires a length parameter which specifies how many bytes have actually been provided for the return area. Each of these descriptions refers to the type of variable required as the third parameter to Parse_Set or Parse_Get.

For several of the variables below, the third parameter is described as "a halfword length followed by the specified number of characters". When used from a PLUS program, the corresponding parameter should normally be a varying-length, character-string type, with maximum length defined as at least 256. This will conform to the halfword-length format expected. Note also, that the parser variable Short_Strings can be used to modify the behaviour of these subroutines, such that these variables are set or returned by using <u>one-byte</u> lengths, followed by the characters. In this case, a PLUS string type with maximum length <=255 (such as the type Varying_String) must be used.

The names used below are the names of PLUS constants which can be used as the second parameter when calling Parse_Set or Parse_Get from a PLUS program. The numbers following in parentheses can be used by the Assembler (or other language) programmer.

Scan_Position (0)	This variable may be accessed only from a semantic routine. The current scan position, as an offset from the beginning of the current input line, is set or returned. The scan position may be set either less than its current value (thus "backing up" in the input) or greater than its current value (thus ignoring the text in between). The third parameter must be a fullword integer variable.

Production_Start_Position (1)

This variable may be accessed only with Parse_Get and only from a semantic routine. If parsing multiple-line input, it will fail if the current production began in a different input line.

The scan position at which the current production started, as an offset from the beginning of the input line, is returned.

The third parameter must be a fullword integer variable.

Input_Text (2)

This variable may be accessed only with Parse_Get, and only from a semantic routine.

A copy of the current input line is returned, in the form of a halfword length followed by the specified number of characters.

The fifth parameter to Parse_Get is required for this variable. If the value passed is True, the upper-case version is returned. If it is False, the original mixed-case version is returned.

Current_Production_Text (3)

This variable may be accessed by Parse_Set or Parse_Get, but only from a semantic routine. If parsing multiple-line input, it will fail if the current production began in a different input line.

When Parse_Get is called, the portion of the input text corresponding to the production currently being performed is returned. That is, it returns that portion of the input string from the scan position when the production was started, to the scan position when the semantic routine was called. The value is returned in the form of a halfword length followed by the specified number of characters.

The fifth parameter to Parse_Get must be given to specify whether the upper-case-only version of the text or the original mixed-case version should be returned.

When Parse_Set is called, the specified string <u>replaces</u> the portion of the input text corresponding to the current production. The third parameter consists of a halfword length followed by the specified number of characters.

Changing the production text normally makes sense only if grammar specifies RETRY following the semantic action doing the replacement. With some care in the grammar, this is one way that a semantic routine can interact with the parser in performing error recovery.

Terminal_Text (4) This variable may be accessed by Parse_Set or Parse_Get, but only from a semantic routine, and only if a terminal is on the semantic stack.

It is used to set or retrieve the input text corresponding to the last terminal scanned.

When Parse_Get is called, the value is returned in the form of a halfword length followed by the specified number of characters.

The fifth parameter to Parse_Get must be given to specify whether the upper-case-only version of the text or the original mixed-case version should be returned.

When Parse_Set is called, the specified string <u>replaces</u> the portion of the input text corresponding to the terminal. The third parameter consists of a halfword length followed by the specified number of characters.

Changing the terminal text normally makes sense only if grammar specifies RETRY following the semantic action doing the replacement. With some care in the grammar, this is one way that a semantic routine can interact with the parser in performing error recovery.

Literal_Text (5) This variable may be accessed by Parse_Set or Parse_Get, but only from a semantic routine, and only if a literal is on the semantic stack.

It is used to set or retrieve the input text corresponding to the last literal scanned.

When Parse_Get is called, the value is returned in the form of a halfword length followed by the specified number of characters.

The fifth parameter to Parse_Get must be given to specify whether the upper-case-only version of the text or the original mixed-case version should be returned.

When Parse_Set is called, the specified string <u>replaces</u> the portion of the input text corresponding to the literal. The third parameter consists of a halfword length followed by the specified number of characters.

Changing the literal text normally makes sense only if grammar specifies RETRY following the semantic action doing the replacement. With some care in the grammar, this is one way that a semantic routine can interact with the parser in performing error recovery.

Semantic_Name (6) This variable may be accessed only by Parse_Get, and only from a semantic routine.

The name for the current semantic action (as used in the grammar), is returned in the form of a halfword length followed by the specified number of characters.

If the syntax tables were generated with the /NOSYMBOLS option, the names will not be available. In this case, the semantic action number, converted to a character string, will be returned.

Semantic_Result (7) This variable may be accessed with Parse_Get or Parse_Set, but only from a semantic routine.

It is used with Parse_Set to save a value on the semantic stack for retrieving later with Parse_Get or Parse_Get_Stack. The third parameter is an arbitrary variable. A fourth parameter must also be given, for this item, to specify the length in bytes of the third parameter.

When used with Parse_Get, it returns the top element from the semantic stack which was previously set with the code Semantic_Result. The returned value of Parse_Get indicates how many bytes were saved. (Remember, if this is greater than the fourth parameter of the call, the value has been truncated.)

Semantic_Result_Word (8)
This variable may be accessed with Parse_Get or Parse_Set, but only from a semantic routine.

It is used with Parse_Set to save a value on the semantic stack for retrieving later with Parse_Get or Parse_Get_Stack.

When used with Parse_Get, it returns the top element from the semantic stack which was previously set with the code Semantic_Result_Word.

The third parameter is an arbitrary fullword variable.

Semantic_Result_String (9)
This variable may be accessed with Parse_Get or Parse_Set, but only from a semantic routine.

It is used with Parse_Set to save a value on the semantic stack for retrieving later with Parse_Get or Parse_Get_Stack. The third parameter specifies the value to be saved. It is a variable consisting of a halfword length, followed by the specified number of characters.

When used with Parse_Get, it returns the top element from the semantic stack which was previously set with the code Semantic_Result_String. The value is returned in the form of a halfword length followed by the characters.

Parse_Result (10) This variable may be accessed only from Parse_Get, and only after completion of a parse.

It returns the value, if any, left on the semantic stack at completion of the goal production.

The type of variable required for the third parameter depends on what kind of value is left by the parse.

Last_Error_Code (11)

This variable may be accessed only by Parse_Get.

Last_Error_Code is set by Parse before returning, and during the processing of some system terminals. It contains a code describing an error condition; a corresponding error message is always set as the value of Last_Error_Message. A code of 0 always indicates no error. See Appendix C for a list of possible error codes and error messages.

The third parameter must be a fullword integer variable.

Last_Error_Message (12)

This variable may be accessed only by Parse_Get.

The message corresponding to Last_Error_Code is returned, in the form of a halfword length followed by the specified number of characters. A null string always indicates no error.

Error_Correction (13)

This variable determines whether error correction is attempted or suppressed in response to CORRECT terms in the grammar.

A value of True means attempt error correction; False suppresses it. The value is set initially to True in conversational mode and False in batch. When the parser is initialized by Parse_Initialize_Nonmts, error correction is set to False. See Chapter VI for details of the error-correction actions.

The third parameter must be a one-byte Boolean (Fortran LOGICAL*1) variable.

Spelling_Correction (14)

This variable determines whether spelling correction is attempted or suppressed in response to CORRECT terms in the grammar.

A value of True means attempt spelling

correction; False suppresses it. The value is set initially to False in batch mode. In conversational mode it is set to True if the MTS option SPELLCOR is ON or PROMPT. When the parser is initialized by Parse_Initialize_Nonmts, spelling correction is set to False.

The third parameter must be a one-byte Boolean (Fortran LOGICAL*1) variable.

Help_File_Name (15) This variable is used to set or retrieve the name of the file to be used in processing the system terminal "Help_Command". It is initially a null string.

The third parameter consists of a halfword followed by the specified number of characters. If the string is longer than 17 on a call to Parse_Set, only the first 17 characters will be used.

Case_Convert (16) This variable determines whether an upper-case form of the input text is parsed, or the original (possibly mixed-case) text.

Its initial value is True, meaning parse an upper-case form. The value at the time the input text is set determines whether an upper-case form is parsed or not.

The third parameter must be a one-byte Boolean (Fortran LOGICAL*1) variable.

Note that, if this is set False, then the parser will not create an upper-case form of the input text. In this case the options of Parse_Get, Parse_Get_Stack, Production_Text, etc., which request an upper-case copy of the input, will return the original-case input, <u>not</u> an upper-case-only form.

Parse_Trace (17) This variable controls tracing of the parser's actions in processing its input.

The third parameter is a fullword integer. A value of 0 means no tracing, 1 means trace all semantic actions, and 2 means trace each term as it is processed. The value is initially set from an external variable CPTRAC, which is normally 0.

The format of the parse trace is described in Section F of Chapter VIII.

Short_Strings (18)

This variable is used to modify the way that string values are passed between the parser and semantic routines by Parse_Get, Parse_Set and Parse_Get_Stack.

The third parameter is a Boolean (Fortran LOGICAL*1) variable.

A value of False means that strings are to be passed in the form of a halfword length followed by a variable number of characters, as stated in other descriptions in this section. A value of True for this variable means that strings should be passed as a one-byte length, followed by a variable number of characters instead. In this case, the maximum length of string that can be set or returned is, of course, limited to 255. The value is initially False.

This option is intended to simplify use in some PLUS applications.

Print_Errors (19)

By default, certain severe errors will cause a message to be printed on Sercom (providing Sercom is defined). This variable controls that printing. If it is set False, the messages will not be printed, but will just set the value of Last_Error_Code and Last_Error_Message. See Appendix C for details.

The third parameter must be a Boolean (Fortran LOGICAL*1) variable.

Input_Buffer_Length (20)

This variable controls the default length of buffers allocated for input lines when a READ term is processed. The initial value is set to 255.

The third parameter should be a fullword integer variable.

User_Psect (21)

This variable is an arbitrary fullword, usually containing a pointer. It is not used at all by the parser, but allows a user to pass the address of a storage area ("Psect") to the semantic routine. The value to be passed is set up as a parameter to Parse_Initialize; it can be changed subsequently, if necessary, by calling Parse_Set.

This mechanism can be used in situations where global storage cannot be used to communicate between the main program and the various semantic actions.

The third parameter is an arbitrary fullword, usually a variable of some pointer type.

Table_Generation_Time (22)

This variable may be accessed only from Parse_Get and only from a semantic routine.

It contains the Julian date and time (in minutes since March 1 1900) at which the syntax tables defining the current production were generated. It is intended for use in "version control", since it provides, in effect, a version identification for the version of the user’s grammar that is in use.

The third parameter must be a fullword (normally an integer).

Parser_Generation_Time (23)

This variable may be accessed only from Parse_Get.

It contains the Julian date and time at which the version of the parser in use was generated.

The third parameter must be a fullword (normally an integer).

Analyzer_Generation_Time (24)

This variable may be accessed only from Parse_Get, and only from a semantic routine.

It contains the Julian date and time at which the version of the table generator (grammar analyzer), which produced the current parse tables, was itself generated.

The third parameter must be a fullword (normally an integer).

Getspace_Routine (25)	This variable allows setting or accessing the getspace subroutine to be used by the parser for subsequent getspace calls. See Section D of Chapter III for further details. The third parameter is a fullword containing the address of the routine to be called.
Freespace_Routine (26)	This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Freespac. See Section D of Chapter III for further details. The third parameter is a fullword containing the address of the routine to be called.
Guser_Routine (27)	This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Guser. See Section D of Chapter III for further details. The third parameter is a fullword containing the address of the routine to be called.
Sercom_Routine (28)	This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Sercom. See Section D of Chapter III for further details. The third parameter is a fullword containing the address of the routine to be called.
Getfd_Routine (29)	This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Getfd. See Section D of Chapter III for further details. The third parameter is a fullword containing the address of the routine to be called.
Freefd_Routine (30)	This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Freefd. See Section D of Chapter III for further details. The third parameter is a fullword containing the address of the routine to be called.

Read_Routine (31) This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for the MTS Read routine. See Section D of Chapter III for further details.

The third parameter is a fullword containing the address of the routine to be called.

Guinfo_Routine (32) This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Guinfo. See Section D of Chapter III for further details.

The third parameter is a fullword containing the address of the routine to be called.

Cuinfo_Routine (33) This variable allows setting or retrieving a subroutine to be used by the parser as a substitute for Cuinfo.

The third parameter is a fullword containing the address of the routine to be called. See Section D of Chapter III for further details.

Input_Routine (34) This variable allows setting or retrieving a subroutine to be used by the parser to perform READ operations requested in the grammar. See Section C of Chapter III for further details.

The third parameter is a fullword containing the address of the routine to be called.

Attention_Test_Routine (35)
This variable allows setting or retrieving a subroutine to be used by the parser during error and help processing, to determine whether an attention interrupt has occurred. See Section C of Chapter III for further details.

The third parameter is a fullword containing the address of the routine to be called.

Parsed_Fdub (36) This variable may be accessed only from Parse_Get, only from a semantic routine, and only if a system terminal has left an Fdname on the semantic stack. Parse_Get will obtain an Fdub for the specified Fdname and return it.

The third parameter must be a fullword variable.

Parsed_Integer (37) This variable may be accessed only from Parse_Get, only from a semantic routine, and only if a system terminal has left an integer on the semantic stack.

The numeric value of the integer is returned.

The third parameter must be a fullword integer variable.

Parsed_String (38) This variable may be accessed only from Parse_Get, only from a semantic routine, and only if a system terminal has left a string on the semantic stack.

The value of the string (with delimiters removed) is returned, in the form of a halfword length, followed by the specified number of characters.

Parsed_Fdname (39) This variable may be accessed only from Parse_Get, only from a semantic routine, and only if a system terminal has left an Fdname on the semantic stack. The Fdname string, with one trailing blank, will be returned in the form of a halfword length, followed by the specified number of characters.

Parsed_Ccid (40) This variable may be accessed only from Parse_Get, only from a semantic routine, and only if the system terminal *Ccid* has left a value on the semantic stack.

The value of the CCID scanned (padded to 4 characters with the appropriate substring of ".$.") is returned.

The third parameter must be a 4-byte character variable (character(4) in PLUS).

Parsed_Project (41) This variable may be accessed only from Parse_Get, only from a semantic routine, and only if the system terminal *Project* has left a value on the semantic stack.

The value of the project code scanned (padded to 4 characters with blanks) is returned.

The third parameter must be a 4-byte character variable (character(4) in PLUS).

Parsed_Line_Number (42)
This variable may be accessed only from Parse_Get, only from a semantic routine, and only if the system terminal *Line_Number* has left a value on the semantic stack.

The internal form of the MTS line number is returned.

The third parameter must be a fullword integer.

Parsed_Real (43)
This variable may be accessed only from Parse_Get, only from a semantic routine, and only if a system terminal has left a parsed real on the semantic stack.

The third parameter may be a 4-, 8- or 16-byte floating-point variable. If a 16-byte return area is provided, the value will be returned as an extended-precision real. If an 8-byte area is provided, the value will be rounded and returned as a double-precision (REAL*8) result. If a 4-byte area is provided, the value will be rounded to single precision and returned.

Default_Help_Name (44)
This variable is used to set or retrieve the name of an entry in the help file that is to be printed for help requests that specify a nonexistent or no symbol.

It is initially the string "DEFAULT".

The third parameter consists of a halfword followed by the specified number of characters. If the string is longer than 10 on a call to Parse_Set, only the first 10 characters will be used.

VI. ERROR CORRECTION

In attempting to parse a given string, the parser simply works its way through the grammar trying all alternatives until it either completes the goal production or fails to find a path that succeeds. Thus it has no *a priori* knowledge of "errors" in the input--other than that the entire string is invalid.

A mechanism has been provided to allow the parser to perform some error correction in response to extra information in the grammar. The item CORRECT in the grammar indicates that, if this point in the production is reached, the parser should attempt to replace part of the input and retry the parse.

Error correction may be enabled or disabled by the parser variable Error_Correction. When it is enabled (True), the parser will first attempt spelling correction, and if this is unsuccessful, ask the user for a replacement string. If error correction is disabled, neither part of the error correction is attempted, and the parser will return to the calling program with Last_Error_Code set as described in Appendix C.

A. THE ERROR STRING

When error recovery is to be attempted, the parser must select a substring of the input as a possible candidate for replacement. The string chosen is called "the error string" in the following descriptions.

The error string will always begin at the scan position when the current production was started. It will extend as least as far as the furthest point reached in any unsuccessful production alternate. The parse looks beyond that point for one of a small set of delimiters to determine the end of the error string.

B. SPELLING CORRECTION

Spelling correction may be enabled or disabled with the parser variable Spelling_Correction. If it is disabled, the parser will proceed directly to error replacement. If it is enabled, the parser will scan the syntax tables to determine whether the error string is a possible misspelling of any literal, optional literal, or combination of the two, that is syntactically allowed by the current production.

Where an optional literal or a literal followed by an optional literal is allowed, the parser will determine whether the error string is a possible misspelling of any valid substring.

If a possible spelling correction is detected, the user will be asked to confirm the substitution. If the user OK's it, the substitution is made and the parser retries part of the parse. Alternatively, the user can reject the substitution and look for an alternative correction, suppress further attempts at error correction and proceed to error replacement, or cancel the parse in progress and return from the parser.

C. ERROR REPLACEMENT

If spelling correction is bypassed or unsuccessful, then the parser computes a new (possibly longer) error string to be replaced, and prompts the user for a replacement for this error string. The user may either enter a replacement, or cancel the parse in progress and return from the parser.

D. RETRYING AFTER ERROR CORRECTION

After spelling correction is applied or a portion of the input replaced, the parser will back up the parse to the nearest production which includes a FENCE term, or if there is none, back to the beginning. It will then retry parsing the (presumably modified) input that resulted from the correction processing, starting from the position corresponding to the FENCE (or from the beginning of the input if there was none).

Note that when a production including FENCE is completed successfully, then the production calling it is automatically fenced at that point. Hence if a fenced production finishes successfully, any subsequent errors will retry from the *end* of that production.

E. ERROR HELP

A simple HELP mechanism is provided as part of the parser's error handling. When a prompt is issued, in either spelling correction or error replacement modes, the user may enter a help command for further information. The following are allowed:

HELP	with no parameter, HELP produces an explanation of the allowable responses.
HELP KEYWORDS	produces a list of all the literals (but not terminals or system terminals) that are valid at the point of the error.
HELP CONTEXT	echoes the current input line, with a row of dashes to indicate the error string.

HELP symbol	provides application-dependent help. If the program has set up a help file (see next chapter), then the help information for the given symbol is printed. If there is no help file, this form is treated the same as HELP with no parameter.

VII. HELP FILES

To encourage programmers to provide a HELP command for command languages, a simple mechanism has been provided to look up symbols in a file and print corresponding information.

A. HELP FILE FORMAT

The help file is an MTS line file with a special "library format". It consists of a directory, followed by an arbitrary number of members.

The directory consists of any number of lines, each consisting of a symbol and the line number in the file at which its definition is found. The library is terminated by a record containing "/END", starting in column 1.

The symbols in the directory are "help items" that may be requested. A directory record may indicate that substrings of the symbol should also be accepted as valid help items. This is done by surrounding the optional part with "|...|". For example

```
EXPR|ESSION| 1000
```

indicates that the library member at line 1000 should be printed for help requests "EXPR", "EXPRE", ... "EXPRESSION", etc.

Each member of the library begins with a record "/BEGIN x", where x is the full form of the symbol in the directory. This record must be at the line specified in the directory. The member ends with the record "/END".

A library member may copy other elements of the same help file by containing a line of the form

```
/INCLUDE helpitem
```

where "helpitem" is another symbol in the help file directory.

A simple example of a help file is given in Appendix B.

B. USING THE HELP MECHANISM

The programmer must initialize the help mechanism by specifying the help file with a call to Parse_Set.

The help file may be used from a program or a grammar in a variety of ways.

1. The subroutine Parse_Help may be called directly to request the information for an item specified as a parameter.

2. The grammar may contain calls to the system terminal *Help(helpitem)* which specify what is to be printed from the help file.

3. The grammar may use the system terminal *Help_Command*.

   This skips over blanks, then matches an arbitrary nonblank symbol, or a null string. If a symbol is given, it is looked up in the directory of the library and the corresponding lines of the file are written to SERCOM. If no symbol is given, the symbol "DEFAULT" is looked up instead. If a symbol is given, but is not defined in the directory, a message is printed, then the symbol DEFAULT is used instead. (An alternative symbol to use instead of DEFAULT may be specified by calling Parse_Set to set the variable Default_Help_Name.)

Thus the simplest form of help command can be provided with just a production of the form:

```
   <Help_Command> is "HE"|"LP"| *Help_Command*;
```

More elaborate forms can be implemented by using *Help* and calling the help routine directly.

VIII. USING THE TABLE GENERATOR AND PARSER

A. RUNNING THE TABLE GENERATOR

The table generator is invoked with the command

```
$RUN *CLPARSEGEN [I/O units] [PAR=parameters]
```

The following logical I/O units may be used.

SCARDS	specifies the file or device containing the grammar to be processed.
SPRINT	specifies the destination of the listing and cross-reference produced by the table generator.
SPUNCH	specifies the destination of the object deck produced by the table generator.
SERCOM	is used, if different from SPRINT, to echo error messages produced by the table generator.
1	specifies the destination of Assembler or PLUS source statements produced by the table generator for use by the user’s semantic routine.

The PAR field may be used to specify any of the options described below. If two or more options are given, they must be separated by commas.

Input to the table generator, specified by SCARDS, is free-format. It consists of productions and declarations defining the command language, table generator options, and comments. Blank lines may appear anywhere.

An input line beginning with "/" is used to specify options. Any number of options may appear in a single command line, separated by commas.

The characters "--" are used to begin a comment. The remainder of the input line is ignored by the table generator.

B. TABLE GENERATOR OPTIONS

The following options may appear either in the PAR field or following "/" in an input command line. Underlining in these descriptions indicates minimum allowed abbreviations.

LIST or NOLIST	controls whether a listing of the input lines is produced on SPRINT. It defaults to LIST.
XREF or NOXREF	specifies whether a cross-reference of symbols in the grammar is to be produced. The default is XREF.
TITLE "quoted string"	specifies a title to appear in the source listing.
OBJECT or NOOBJECT	determines whether an object module is to be produced. The default is OBJECT.
DECK or NODECK	is a synonym for OBJECT/NOOBJECT.
DECLARATIONS [decl-option] or NODECLARATIONS	determines whether auxiliary output is produced on unit 1 in PLUS, Assembler, or neither. "decl-option" may be PLUS, PLUS LIBRARY or ASSEMBLER. The form PLUS LIBRARY will cause the generated PLUS declarations to be in the form of a PLUS source library with a single member. The default is DECLARATIONS PLUS.
EJECT [eject-option]	causes the table generator listing to skip to a new page. The optional "eject-option" is used to skip to a "double page". It may be EVEN or ODD to skip to an even- or odd-numbered page, or DOUBLE, which is equivalent to ODD.
OPTIMIZE or NOOPTIMIZE	determines whether the table generator attempts to perform table optimizations. Currently the only optimization performed is to attempt to "fold" character strings used in the grammar by searching for strings embedded in other strings. This results in a rather small reduction in the table size, with a large increase in the time required by the table generator. The default is NOOPTIMIZE.

SYMBOLS or NOSYMBOLS	determines whether the names of productions, semantics, and labels are put into the object module generated. These names are used by the parser's tracing procedure, and for the Parse_Get item Semantic_Name. Omitting the symbols will make the object deck a lot smaller. If the symbols are not included in the object module, numeric codes will be used instead for tracing and Semantic_Name. The default is SYMBOLS.
NAME "quoted symbol"	specifies a name which may be used to qualify the PLUS declaration for labels, and to form the library member name if the PLUS LIBRARY declarations option is specified. It defaults to the name of the first goal symbol.
TRACE or FULLTRACE or NOTRACE	is used in debugging the table generator to turn on tracing of the input parsing. The default is NOTRACE. This option is only of use to programmers maintaining the table generator.

C. TABLE GENERATOR OUTPUT

1. Listings

The table generator normally produces a listing of the input grammar and a cross-reference of all symbols used in the grammar as output to SPRINT. These may be suppressed by specifying either or both of the options NOLIST and NOXREF.

Any input line in which an error is detected is listed regardless of the setting of the listing option. Input lines containing errors and error messages are echoed to SERCOM if it refers to a different file or device from SPRINT.

2. Object Program

An object program is produced on SPUNCH unless the option NOOBJECT is specified. The object module defines a single csect, whose name is that of the first goal. It includes entry points for each goal symbol in the grammar.

3. Declarations

Auxiliary declarations may be produced in PLUS, Assembler or both (or may be suppressed) according to the DECLARATIONS option. These declarations are produced on unit 1.

The declarations are intended to assist in the writing of semantic procedures. The declarations consist of three parts:

a. definitions specifying the correspondence between semantic action symbols used in the grammar and the numeric values which will be used in calls to the semantic routine.

   For PLUS declarations, this is given by a PLUS identifier-list type declaration for each semantic procedure specified in the grammar. The declarations have the form:

```
type x_Semantic_Action_Type is
                          (symbol0,...,symboln);
```

   where "x" is the name of the semantic procedure and symbol0,...,symboln are the semantic actions for which calls will be made to that procedure.

   For Assembler declarations, a series of equates of the form:

```
symbol0  EQU   0
symbol1  EQU   1
          .
          .
          .
symboln  EQU   n
```

   will be generated for the semantic actions of each semantic procedure.

b. definitions specifying the correspondence between label symbols used in the grammar, and the numeric values that will be used as parameters to Parse_Get_Stack.

   For PLUS declarations, another PLUS identifier-list type declaration is produced, of the form:

```
type y_Label_Type is (symbol0,...,symbolm);
```

   where "y" is the grammar name. This defaults to the name of the first goal symbol in the grammar. It may be changed with the /NAME option.

For Assembler declarations, another sequence of equates is produced to specify the codes.

c. definitions for the goals of the grammar.

For PLUS programs, this takes the form of a series of PLUS external variable declarations of the form:

```
variable goali is Parse_Goal_Type
                                  external "extnamei";
```

where goali is the production name of a goal symbol, and extnamei is the corresponding external symbol, as specified with ENTRY, or determined by default from the production name.

Corresponding Assembler declarations are not currently produced. A simple EXTRN is all that is required.

If the declarations were requested in the form of a PLUS library (/DECLARATIONS PLUS LIBRARY), then the output file will be a library with a single member. The member will have the name

```
y_Definitions
```

where y is the grammar name determined from the first goal or the /NAME option.

D. <u>PLUS SOURCE LIBRARY</u>

To assist in writing PLUS programs which use the parser, a PLUS source library is provided. This source library contains definitions of parser routines and types which are required when using the parser. Thus a PLUS program will normally use %Include to include the definitions of those procedures and types it requires. See Appendix B for a complete example program using the definitions from the library.

The library is contained in the file *CLPARSELIB. When using this library, the standard PLUS source library is required also. Thus to compile a PLUS program using the parser, one specifies a run command of the form:

```
$RUN *PLUS ... 0=*CLPARSELIB+*PLUS.SOURCELIB
```

The library contains the following:

1. Parser Procedures

   The library contains a member defining each of the procedures of the parser. Each of these includes any associated definitions it requires. In particular, the definitions of Parse_Set and Parse_Get will include the definitions of codes for all the parser variables described in Chapter V.

   Those routines which return a result indicating success or failure are declared in the library to have an "optional result". Thus programs calling these parser routines may choose to ignore the returned value.

2. Parse Goal Type

   Member Parse_Goal_Type contains a type definition that is used in defining the goal symbols to be passed to the parser.

3. Semantic Procedure Type

   Member Semantic_Procedure_Type defines the type required for the semantic procedures called by the parser.

   The PLUS program will usually contain something like:

```
%Include(Semantic_Procedure_Type);
procedure Command_Semantics is
          Semantic_Procedure_Type;
```

   If this declaration is used, the semantic routine may refer to its parameters by the names Parser_Control_Block, Psect_Pointer, and Semantic_Action, and to the result by the name Success.

4. Other Procedure Types

   The library also contains members:

```
Parse_Attention_Test_Type,   Parse_Input_Type,
Parse_Getspace_Type  and  Parse_Freespace_Type
```

   each of which defines procedure types for procedures that may be provided to the parser.

   Parse_Attention_Test_Type may be used to define the procedure to be called to test for attentions during error and help processing. (See Section C of Chapter III.)

   Parse_Input_Type may be used to define the procedure to be called by the parser for input.

Parse_Getspace_Type and Parse_Freespace_Type define procedures to be passed to Parse_Initialize_Nonmts.

5. Parse Trace Type

   This simply defines constants that may be used for Parse_Set when setting the Parse_Trace variable.

6. Parse String Type

   This defines a PLUS varying-length string of maximum length 256. It may be used when the halfword-length string form is required, but only fairly short strings are actually involved.

E. RUNNING THE PARSER

The parser routines, and PLUS run-time support routines are contained in the resident system. They will normally be found as part of the usual MTS library-searching process.

The PLUS/coding-conventions entry points are defined by a low-core symbol table CCSYMBOL. A program using these entry points must contain the necessary loader control records to cause this table to be searched. These records are normally generated by the PLUS compiler.

F. PARSER TRACING

The parser includes a built-in mechanism for tracing the actions performed as it attempts to parse its input. This is useful when debugging grammars or programs using the parser.

Two levels of tracing are provided. The first level prints out a line of input immediately before each semantic action is performed. The second level prints out a line as each term of a production is begun.

The lines printed contain two parts. To the left is a description of the term, more-or-less as it appears in the grammar. When the second level of tracing is used, these terms are indented to indicate the production nesting. To the right of this is a portion of the input text being parsed, with a vertical bar "|" indicating the point currently being examined by the parser.

The parse trace may be enabled from a program by calling Parse_Set to set the parser variable Parse_Trace.

The system terminal *Trace* may be used within a grammar to set tracing. For example, the following provides a possible implementation of a trace command:

```
      <Trace_Cmd> is "TRACE" *blanks*
                       { "OFF" *Trace(0)*
                       or "ON" *Trace(1)*
                       or "FULL" *Trace(2)*};
```

Tracing can also be enabled under SDS by patching the 17th byte of the parser control block to the option required (0 for no tracing, 1 for semantic tracing, 2 for full tracing).

The example in Appendix B includes a sample of the trace output.

APPENDIX A - TABLE GENERATOR INPUT LANGUAGE

This Appendix uses the table generator's input language to define itself. The grammar is similar to that actually used by the table generator in parsing its input, but has been simplified by removing all semantic calls, and all considerations of comments, options, and error handling.

```
goal <Grammar> is [<Sep> <Production>] ... ;

<Production> is *End_Of_File* fail
             or <Next_Production> <Sep> ";" ;

<Next_Production> is "TERMINAL" <Sep> <Terminal>
                  or "PROCEDURE" <Sep> <Semantic_Declaration>
                  or "GOAL" <Sep> <Goal_Definition>
                  or "LABEL" <Sep> <Label_Declaration>
                  or <Nonterminal_Definition>;

--  Semantic declarations...

<Semantic_Declaration> is <Entry_Name> <Sep> [<Linkage> <Sep>]
                             "FOR" <Sep> <Semantic_Names>;

<Semantic_Names> is "ALL"
                 or <Semantic_Name> [<More_Semantic_Names>]...;

<Linkage> is "R-TYPE"
          or "S-TYPE";

<More_Semantic_Names> is <Sep> "," <Sep> <Semantic_Name>;

--  Label declarations...

<Label_Declaration> is <User_Name> [<More_Label_Name_Decl>]...;

<More_Label_Name_Decl> is <Sep> "," <Sep> <User_Name>;

--  Terminals...

<Terminal> is <Prod_Name> <Sep> "IS" <Sep> <Integer> <Sep>
                   [<Upper>  <Sep>] <Terminal_Characters>;

<Upper> is "TO" <Sep> <Integer>
        or "OR" <Sep> "MORE";

<Terminal_Characters> is "CHARACTER"|"S"|
                      or {"OF" or "EXCLUDING"}
                                  <Terminal_String>...;
<Terminal_String> is <Sep> <Terminal_Segment>;
<Terminal_Segment> is <Qstring>
```

```
                   or <Hex_String>;

--  Goal definitions...

<Goal_Definition> is <Prod_Name> <Sep> [<Entry>] "IS"
                                      <Sep> <Alternate_List>;

<Entry> is "ENTRY" <Sep> <Entry_Name> <Sep>;

--  Define nonterminal rules...

<Nonterminal_Definition> is <Prod_Name> <Sep> "IS" <Sep>
                                        <Right_Part>;

<Right_Part> is "EXTERNAL" <Sep> [<Entry_Name>]
             or <Alternate_List>;

-- Alternates...

<Alternate_List> is <Alternate> ["OR" <Sep> <Alternate>];

<Alternate> is <Term>...;

--   Terms.  This fails only when at end-of-alternate...

<Term> is [<Term_Label>] <Simple_Term> <Sep> [<Repeated>];

<Term_Label> is <Id> <Sep> ":" <Sep>;

<Simple_Term> is <Prod_Name>
              or <Lbracket> <Sep> <Alternate_List> <Rbracket>
              or <Lbrace> <Sep> <Alternate_List> <Rbrace>
              or "GO" <Sep> "TO" <Prod_Name>
              or "#" <Sep> <Semantic_Name>
              or "*" <Sep> <Sys_Term> <Sep> [<Params>] "*"
              or <Literal>
              or "|" <Literal> "|"
              or "SUCCEED"
              or "FAIL"
              or "READ"
              or "FENCE"
              or "RETRY"
              or "CORRECT"
              or "QUIT";

<Repeated> is "..." <Sep>;

<Params> is "(" <Sep> <Parlist> <Sep> ")" <Sep>;

<Parlist> is <Parameter> [<Sep> "," <Sep> <Parameter>]...;

<Parameter> is <Integer>
            or <Literal>;
```

```
--   Various kinds of symbols...

<Prod_Name> is "<" <User_Name> ">";

<Entry_Name> is <Id>
             or <Qstring>;
<Semantic_Name> is <User_Name>;
<User_Name> is <Id>;
<Sys_Term> is <Id>;

<Literal> is <Qstring>
          or <Hex_String>;
<Hex_String> is "'" [Hex] "'";
<Qstring> is """" [String_Seg]... """";
<String_Seg> is <Nonq>
             or """"""; -- double quote

<Lbrace> is "{" or "(/";
<Rbrace> is "}" or "/)";

<Lbracket> is "[" or "(";
<Rbracket> is "]" or ")";

-- Separator used between any two tokens...

<Sep> is <Sp> [*End_Of_Line* read <Sp>]...;

--  Scan-classes used above...

terminal <Sp> is 0 or more of " ";
terminal <Nonq> is 1 or more excluding """";
terminal <Hex> is 1 or more of " 1234567890ABCDEF";
terminal <Integer> is 1 to 15 of "1234567890";
terminal <Id> is 1 to 100 of
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ_1234567890";
```

APPENDIX B - EXAMPLES

This appendix contains some simple but complete examples. It consists of three variations of grammars and programs to process simple integer arithmetic expressions. The input consists of either an expression, or one of the commands HELP, TRACE, MTS or STOP.

A. THE GRAMMAR

The following grammar defines the allowed input. Note that the definition of expression is recursive, and proceeds through two levels, processing "sums" and "products". This format, which is typical of grammars defining expressions, forces the semantic actions to be performed in the right order to implement the standard "precedence rules" of arithmetic.

The label Expr is used on two alternatives of <Factor> to label the term that has a "value" to keep. The semantic action #Copy_Top will fetch the value of the term labelled Expr, and save it as the value of the production, thus passing it back to the <Term> production. This isn't necessary for the first alternative, since *Integer* is the last term, and hence the value is already on top of the stack.

Also note that the <Term> and <Expression> productions are very similar. The operation is left on the semantic stack by one of the productions <Addop> or <Multop>. Each time the semantic action Operation is performed, it will fetch two operands and an operation, perform the operation, then set a semantic value. This value is itself labelled Opnd, and thus serves as one of the operands to be fetched on the next iteration, if any.

```
/title "Example grammar for simple expressions"
procedure Semantics for all;

goal <Command> is "ST"|"OP"| #Stop_command
               or "MT"|"S"| #Mts_command
               or "HE"|"LP"| *Help_Command*
               or "TRACE" *Blanks* { "ON" *Trace(1)*
                                   or "OFF" *Trace(0)*
                                   or "FULL" *Trace(2)*}
               or Res:<Expression> <Nothing> #Print_Result
               or correct;

<Expression>  is Opnd:<Term>
                  [Op:<Addop> Opnd:<Term> Opnd:#Operation]...;
<Term> is *Blanks* Opnd:<Factor> *Blanks*
                  [Op:<Multop> Opnd:<Factor> Opnd:#Operation]..
```

```
<Factor> is *Integer*
         or "’" Expr:*Hex_Integer* "’" #Copy_Top
         or "(" Expr:<Expression> ")" #Copy_Top;

<Nothing> is *End_Of_Line*
          or correct;
<Addop> is "+" or "-";
<Multop> is "*" or "/";
```

B. RUNNING THE TABLE GENERATOR

If the grammar is in the file SYNTAX, the table generator can be run with a command of the form:

```
$RUN *CLPARSEGEN SCARDS=SYNTAX SPUNCH=-TABLES 1=-DECL
```

The following declarations are produced in file -DECL by the table generator:

```
%Include(Parse_Goal_Type);
type Semantics_Action_Type is (Copy_Top, Mts_Command,
      Operation, Print_Result, Stop_Command);
type Command_Label_Type is (Expr, Op, Opnd, Res);
variable Command is Parse_Goal_Type external "COMMAND ";
```

C. A PLUS PROGRAM

The program to evaluate expressions uses the semantic stack to maintain intermediate results. The basic elements of expressions (*Integer* and *Hex_Integer*) leave the parsed numbers on the stack. The semantics for a binary operation will pop two values off the stack, apply the operation, and push the result back on the stack. The final result is popped off the stack and printed.

The entire program appears on the following pages. In the interests of brevity, it does not check the return values from the various parser routines called.

The program requires declarations from the parser source library and the standard PLUS source library. If the program is in the file EXPR.S, it can be compiled with a command of the form:

```
$RUN *PLUS SCARDS=EXPR.S SPUNCH=-EXPR                    -
                           0=*CLPARSELIB+*PLUS.SOURCELIB
```

D. THE HELP FILE

The program expects a help file in the file CCID:HELPFILE. A listing of possible contents for this file follows:

```
   1  default 1000
   2  stop 2000
   3  mts 3000
   4  expr|ession| 4000
   5  /end
1000  /begin default
1001  To stop, enter STOP or an end-of-file.
1002
1003  To calculate an arithmetic expression, enter the
                                                    expression
1004  Enter HELP EXPRESSION for description of expressions
                                                       allowed
1005  /end
2000  /begin stop
2001  The STOP command terminates execution of this program.
2002  /end
3000  /begin mts
3001  The MTS command returns to MTS.  $RESTART can be used to
3002  resume execution of this program.
3003  /end
4000  /begin expression
4001  An expression is an arbitrary sequence of constants
4002  connected with the operators +, - , *, /.
4003
4004  The expression is evaluated according to the standard
4005  arithmetic precedence rules.  Parentheses may be used in
4006  the normal way to control order of evaluation.
4007
4008  A constant may be an optional signed integer, or
4009  a hexadecimal number surrounded by apostrophes.
4010  /end
```

E. RUNNING THE PROGRAM

The complete program consists of the object generated by the compiler, the object generated by the table generator and the resident system parser and library routines. Thus a possible run command would be:

```
$RUN -EXPR+-TABLES
```

Following is a sample session, illustrating error correction and help processing. (Input from user is in upper case, output from program/parser in mixed upper and lower case.)

```
#$RUN -EXPR+-TABLES
 2+2
 = 4
 2 * (3+4)
 = 14
 'FF'
 = 255
 X+2
 "X+2" is invalid.
 Enter replacement, "CANCEL", or a help command.
?HELP
 The current input line is invalid.
 You may enter:
   (... here it explains the allowed responses)
 Enter replacement, "CANCEL", or a help command.
?HELP KEYWORDS
 The following keywords are valid:
       STOP
       MTS
       HELP
       TRACE
 Enter replacement, "CANCEL", or a help command.
?HELP EXPR
 An expression is an arbitrary sequence of constants
   (... etc.; explanation comes from help file)
 Enter replacement, "CANCEL", or a help command.
? 2+2
 = 4
 NT
 "ST" ("STOP") for "NT"?
?NO
 "MT" ("MTS") for "NT"?
?YES
```

F. TRACE OUTPUT

The example grammar includes a "TRACE" command to enable parser tracing. The following is an example of the output that might be produced if this is activated. (The numbers at the beginning of each line are not part of the trace output; they are included for reference in the following explanations.)

```
 1     "ST"                            |  '00f0'
 2     "MT"                            |  '00f0'
 3     "HE"                            |  '00f0'
 4     "TRACE"                         |  '00f0'
 5     Res:<Expression>                |  '00f0'
 6       Opnd:<Term>                   |  '00f0'
 7         *Blanks*                    |  '00f0'
 8           ({44})                    |  '00f0'
 9             <Sp>                    |  '00f0'
10             #Reset_Semantic_Stack        |'00f0'
```

```
11          Opnd:<Factor>                    |'00f0'
12            *Integer*                      |'00f0'
13              ("+")                        |'00f0'
14              <Number>                     |'00f0'
15              "-"                          |'00f0'
16            "'"                            |'00f0'
17            Expr:*Hex_Integer*             '|00f0'
18              <Hexint>                     '|00f0'
19              #Set_Hex_Integer             '00f0|'
20            "'"                            '00f0|'
21            #Copy_Top                      '00f0'|
22          *Blanks*                         '00f0'|
23            ({44})                         '00f0'|
24              <Sp>                         '00f0'|
25          ({12})...                        '00f0'|
26            Op:<Multop>                    '00f0'|
27              "*"                          '00f0'|
28              "/"                          '00f0'|
29        ({11})...                          '00f0'|
30          Op:<Addop>                       '00f0'|
31            "+"                            '00f0'|
32            "-"                            '00f0'|
33      <Nothing>                            '00f0'|
34        *End_Of_Line*                      '00f0'|
35          #Check_Eol                       '00f0'|
36      #Print_Result                        '00f0'|
```

This is a complete trace, as the parser attempts each term of the grammar. To the right is the input being parsed (the string " '00f0'"). The vertical bar shows the current scan position.

The first four lines show the attempt to match the first alternatives of <Command>. At line 5, the parser is attempting the <Expression> term (which is labelled Res:). This requires that it attempt to match <Term> (indented, line 6), which in turn attempts to match *Blanks*. Lines 8-10 are the internal expansion of *Blanks*. The parse then continues with the next element of <Term>, by attempting to match a <Factor>. Note the input position has moved over the blanks. The parse looks first for an *Integer*; then, since this fails, for a *Hex_Integer*. The parse continues in this way, printing each term from the grammar as it is processed.

The numbered productions at line 8, 23, 25, etc. correspond to nested, unnamed productions used in the grammar. The numbers are internal names for these productions. For example, line 25 corresponds to the element

```
   [Op:<Multop> Opnd:<Term> Opnd:#Operation]...
```

in the grammar, and the lines following show the attempt to match this subexpression.

PLUS Example Program

```
%Title := "Example program for parser";
/* Include some standard SOURCELIB definitions */
%Include(Numeric_Types, String_Types);
%Include(Integer_To_Varying);

%Include(Sercom_String, Guser);
%Include(Mts);

/* Include the parser declarations that are needed. */
%Include(Semantic_Procedure_Type,
   Parse_String_Type, Parse, Parse_Initialize, Parse_Terminate,
   Parse_Set, Parse_Get, Parse_Get_Stack);

%Include(Parse_Goal_Type);
type Semantics_Action_Type is (Copy_Top, Mts_Command, Operation,
      Print_Result, Stop_Command);
type Command_Label_Type is (Expr, Op, Opnd, Res);
variable Command is Parse_Goal_Type external "COMMAND ";

/* Global variables used by the semantic routines. */
global Semantics_Definitions;
   variable Stop_Flag is boolean;
end Semantics_Definitions;

procedure Main;
procedures Semantics is Semantic_Procedure_Type;

definition Main;
   variable Input_Buffer is character(256),
      Input_Len is Short_Integer,
      Line_Number is Integer,
      Rc is Integer;
   variable Pcb is pointer to Parser_Control_Block_Type;

   Pcb := Parse_Initialize(Null);

   /* Set up the help file. */
   Parse_Set(Pcb, Help_File_Name,
      Parse_String_Type("CCID:HELPFILE"));

   Stop_Flag := False;
   cycle
      Guser(Input_Buffer, Input_Len, 0, Line_Number, return code
         Rc);
      exit when Rc  = 0;
      Parse(Pcb, Command, Address(Input_Buffer), Input_Len);
      exit when Stop_Flag;
   end cycle;

   Rc := Parse_Terminate(Pcb);
end;
```

```
definition Semantics
   Success := True;

   select Semantic_Action from
   case Operation:
      variable Opstr is Parse_String_Type,
         Operand1, Operand2 are Integer;
      Parse_Get_Stack(Parser_Control_Block, Opnd, Operand1,
         Byte_Size(Operand1));
      Parse_Get_Stack(Parser_Control_Block, Op, Opstr,
         Byte_Size(Opstr), True);
      Parse_Get_Stack(Parser_Control_Block, Opnd, Operand2,
         Byte_Size(Operand2));
      select Substring(Opstr, 0, 1) from
      case "+":
         Operand1 +:= Operand2
      case "-":
         Operand1 -:= Operand2
      case "*":
         Operand1 *:= Operand2
      case "/":
         if Operand2 ¬= 0
         then
            Operand1 /:= Operand2
         else
            Operand1 := 0;
            Success := False;
         end;
      end select;
      Parse_Set(Parser_Control_Block, Semantic_Result_Word,
         Operand1);

   case Copy_Top:
      variable Value# is Integer;
      Parse_Get_Stack(Parser_Control_Block, Expr, Value#,
         Byte_Size(Value#));
      Parse_Set(Parser_Control_Block, Semantic_Result_Word,
         Value#);

   case Print_Result:                  /* print */
      variable Value# is Integer;
      Parse_Get_Stack(Parser_Control_Block, Res, Value#,
         Byte_Size(Value#));
      Sercom_String(" = " || Integer_To_Varying(Value#, 0));

   case Stop_Command:                  /* Stop */
      Stop_Flag := True;

   case Mts_Command:                   /* MTS */
      Mts();

   end select;
end;
```

Examples

G. FORTRAN EXAMPLE

This example repeats the previous one, using a Fortran semantic routine instead.

Minor changes to the grammar are necessary. The semantic procedure must be declared to be S-TYPE, and its name must be no more than six characters. Similarly, the goal symbol must have an external name of six characters or less. It is also advisable to specify the semantic actions and labels, so that the codes are determined by the programmer rather than by the table generator.

Thus the declarations should be:

```
procedure Semant S-type for Print_Result, Stop_Command,
   Mts_Command, Operation, Copy_Top;

label Expr, Opnd, Op, Res;

goal <Command> entry "CMD" is
    --- etc. ---
```

The semantic action Print_Result will have code 0, Stop_Command will be 1, and so on.

A possible Fortran program comparable to the preceding PLUS program follows. Note that the goal symbol is declared EXTERNAL in the program that calls the parser.

This program also fails to check the return values from the parser routines called. A safer version would declare the routines as LOGICAL (CPSET, CPARSE) or INTEGER (CPGSTK, etc.) and invoke them as functions in order to check for successful operation.

Fortran Example Program

```
C Parser example program in Fortran
C
C
C Define symbols for codes used in calls to Parse_Get/Parse_Set
      INTEGER   HELPFN/15/
C Variables for I/O
      INTEGER*2 LEN
      INTEGER   LNUM, ILEN
      LOGICAL*1 BUF(100)
C Parser declarations
      INTEGER CPINIT, PCB
      EXTERNAL CMD
C Semantic variables
      LOGICAL*1 STOP
      COMMON /PCOM/ STOP
C Help file
      INTEGER*2 HELPF(8) /13, ’CC’, ID’, ’:H’, ’EL’,
     1  ’PF’, ’IL’, ’E ’/
C Initialize
      PCB = CPINIT(0)
      CALL CPSET(PCB, HELPFN, HELPF)
      STOP = .FALSE.
C Command processing loop
10      CALL GUSER(BUF, LEN, 0, LNUM, &99)
        ILEN = LEN
        CALL CPARSE(PCB, CMD, BUF, ILEN)
        IF (.NOT. STOP) GO TO 10
99    CALL CPTERM(PCB)
      STOP
      END
C
      LOGICAL FUNCTION SEMANT(PCB, PSECT, ACTION)
C
C procedure Semant S-type for Print_Result, Stop_Command,
C    Mts_Command, Operation, Copy_Top;
C label Expr, Opnd, Op, Res;
C
C Define symbols for labels...
      INTEGER EXPRL, OPNDL, OPL, RESL/0,1,2,3/
C Parse_Get/Set codes...
      INTEGER SEMRES/8/
C
C Parameters...
      INTEGER PCB, PSECT, ACTION
C Character constants
      INTEGER*2 PLUS/’ +’/, MINUS/’ -’/, MULT/’ *’/, DIV/’ /’/
C Kludge for character compares
      INTEGER*2 ITEST / ’  ’/
      LOGICAL*1 LTEST(2)
      EQUIVALENCE (ITEST, LTEST)
```

```
C Semantic variables
      LOGICAL*1 STOP
      COMMON /PCOM/ STOP
C Local Temporaries
      INTEGER IWORD1, IWORD2
C Return area for operation (2-byte length + 1 character)
      LOGICAL*1 IOP(3)
C
C Set default result
      SEMANT = .TRUE.
C Offset to one-origin
      ACTION = ACTION + 1
      GO TO (10,20,30,40,50), ACTION
C
C Error
        SEMANT = .FALSE.
        RETURN
C Print_Result
10      CALL CPGSTK(PCB, RESL, IWORD1, 4)
        WRITE (6,999) IWORD1
        RETURN
C Stop Command
20      STOP = .TRUE.
        RETURN
C MTS command
30      CALL MTS
        RETURN
C Operation
40      CALL CPGSTK(PCB, OPNDL, IWORD1, 4)
        CALL CPGSTK(PCB, OPL, IOP, 3, .TRUE.)
        CALL CPGSTK(PCB, OPNDL, IWORD2, 4)
        LTEST(2) = IOP(3)
        IF (ITEST.EQ.PLUS) GO TO 41
        IF (ITEST.EQ.MINUS) GO TO 42
        IF (ITEST.EQ.MULT) GO TO 43
C   Divide
          IF (IWORD2.EQ.0) GO TO 44
            IWORD1 = IWORD1 / IWORD2
            GO TO 45
44        IWORD1 = 0
            SEMANT = .FALSE.
            GO TO 45
C   Plus
41        IWORD1 = IWORD1 + IWORD2
            GO TO 45
C   Minus
42        IWORD1 = IWORD1 - IWORD2
            GO TO 45
C   Mult
43        IWORD1 = IWORD1 * IWORD2
C   Set result.
45        CALL CPSET(PCB, SEMRES, IWORD1)
          RETURN
```

```
C
C  Copy Top
50      CALL CPGSTK(PCB, EXPRL, IWORD1, 4)
        CALL CPSET(PCB, SEMRES, IWORD1)
C
999   FORMAT('  = ', I8)
      END
```

H. EXTERNAL GRAMMAR EXAMPLE

As a final example, we will split the grammar and semantic routines for evaluating the expressions into a separate external package which could then be used in implementing the grammar of the previous examples.

The grammar for the expressions is:

```
procedure Exprsem for Operation, Copy_Top;

goal <Expression> entry "EXPR"
       is Opnd:<Term> [Op:<Addop> Opnd:<Term>
                        Opnd:#Operation]...;
<Term> is *blanks* Opnd:<Factor> *blanks*
                 [Op:<Multop> Opnd:<Term> Opnd:#Operation]...;
<Factor> is  Expr:*Integer* #Copy_Top
         or  "'" Expr:*Hex_Integer* "'" #Copy_Top
         or  "(" Expr:<Expression> ")" #Copy_Top;

<Addop> is "+" or "-";
<Multop> is "*" or "/";
```

This is just the "expression" productions from the previous grammar. <Expression> is now a goal, however, and a new semantic routine has been introduced.

The program for the expressions is as follows. This can be compiled independently of any programs that may use it. The procedure Exprsem will evaluate expressions parsed by goal <Expression> and leave the result on the semantic stack.

Expression Semantic Routine

```
%Title := "Expression Semantics";
/* Include some standard SOURCELIB definitions */
%Include(Numeric_Types, String_Types);
%Include(Integer_To_Varying);

/* Include the parser declarations that are needed. */
%Include(Semantic_Procedure_Type,
   Parse_String_Type, Parse_Set, Parse_Get_Stack);
```

```
%Include(Parse_Goal_Type);
type Exprsem_Action_Type is (Operation, Copy_Top);
type Expression_Label_Type is (Expr, Op, Opnd);
variable Expression is Parse_Goal_Type external "EXPR    ";
procedures Exprsem is Semantic_Procedure_Type;

definition Exprsem
   Success := True;

   select Semantic_Action from
   case Operation:
      /* As in previous example
           ...
      */

   case Copy_Top:
      /* As in previous example
           ...
      */
   end select;
end;
```

A possible grammar to use this package might be:

```
procedure Semant for Print_Result, Stop_Command,
   Mts_Command;

goal <Command> is "ST"|"OP"| #Stop_command
               or "MT"|"S"| #Mts_command
               or "HE"|"LP"| *Help_Command*
               or "TRACE" *blanks* { "ON" *Trace(1)*
                                   or "OFF" *Trace(0)*
                                   or "FULL" *Trace(2)*}
               or Res:<Expression> <Nothing> #Print_Result
               or correct;

<Expression>  is external "EXPR";
<Nothing> is *End_Of_Line*
          or correct;
```

<Expression> is now defined as an external production. The parser will invoke the other grammar at the appropriate points, and call its semantic routine as necessary.

The corresponding main program and semantic routines are straightforward.

APPENDIX C - ERROR MESSAGES AND CODES

The parser variables Last_Error_Code and Last_Error_Message may be set by various error conditions that occur during parsing. They will always be set before the parser returns, and may be set during semantic processing for certain system terminals.

Error codes >= 100 indicate serious errors, which make successful continuation unlikely. For these, the error message will normally be written to Sercom. This can be prevented by setting the parser variable Print_Errors.

The currently defined codes are as follows:

0 ""

The parse succeeded.

1 "Parse failed"

This message occurs if the input cannot be matched by the specified goal production.

2 "Correction cancelled"

This means CANCEL was entered in response to an error recovery prompt.

3 "Correction suppressed"

This message occurs if a "CORRECT" item is encountered at some point in the parse, but the value of the parser variable Error_Correction suppresses error correction.

4 "Can’t correct - more than one input line"

Error correction will not be attempted when parser is reading input if the "error string" crosses input lines.

5 "Fenced production failed"

This occurs if an alternate which contains FENCE fails. The parser abandons the parse in this case.

6 "QUIT executed"

This means the parser terminated by executing a QUIT term.

7 "READ terminated by attention"

This means an attention interrupt was generated while the parser was attempting to read more input.

10 "Integer out of range"

This may be set by the *Integer* system terminal.

11 "Line number has too many fractional digits"
12 "Line number out of range"

The above two may be set by the *Line_Number* system terminal.

13 "String longer than 32767 characters"

This may be set by the *String*, *Quoted_String*, *Primed_String* system terminals.

14 "REAL number out of range"

This may be set by the *Real* system terminal.

101 "Invalid table format"

The syntax tables are invalid. This may indicate that they have been clobbered by stray stores from the program.

102 "System service not available"

This may occur if error correction is requested when the required services haven't been defined.

110 "Unable to expand parser stack"
111 "Unable to expand semantic stack"
112 "Unable to get space for semantics"
114 "Unable to get space for internal buffer"

These all indicate a nonzero return code from the Getspace subroutine.

APPENDIX D - SUBROUTINE CALLING SEQUENCES

This section describes the linkage and parameter passing used by the parser routines.

Most of the routines are available in two versions. One version uses a Fortran/OS linkage with standard S-type parameter list. The other uses subroutine linkage providing a stack, known as the "coding-conventions (CC) linkage". These routines also use R-type parameters. The coding-conventions versions should be used from PLUS programs, and may also be used with Assembler programs.

A. LINKAGE CONVENTIONS

For all coding-conventions routines, at entry to a procedure, R13 must contain the address of a stack to be used by that procedure. R11 contains the address of the global pseudo-register vector. The pseudo-register is not used by any of the parser routines, and so does not necessarily have to be provided. The parser will pass on to the semantic routines whatever value is in R11 when it is called.

For all OS-linkage routines, at entry, R13 must contain the address of an 18-word save area. The OS-linkage initialization routines CPINIT and CPINM allocate a stack whose address is saved in the parser control block. The other routines use the same stack, retrieved from the parser control block.

On calls to S-type semantic routines, R13 will point to a save area.

In both kinds of routines, the procedure is called by:

```
            L     R15,=V(procname)
            BALR  R14,R15
```

B. PARAMETER PASSING

For the coding-conventions R-type routines, parameters are passed in general registers R0-Rn.

For the OS-linkage S-type routines, at entry to the procedure, general register R1 points to a list of addresses, which in turn point to the actual parameters.

For both types of routines, results (if any) are returned in general register R0.

The following descriptions indicate the parameters and results expected for each of the routines. The correspondence is not totally consistent. In some cases a parameter to the R-type routine is the same as an address from the parameter list for the S-type routine. In other cases, the R-type parameter corresponds to the word pointed to from the parameter list.

The Attention-Testing Routine

Attention-testing routines always take S-type parameters. The call always provides a stack, hence is valid for either OS or CC linkage.

S-type Parameters:

0(R1)	address of a fullword containing the user psect pointer specified in the call to Parse_Initialize.

Result:	R0	a Boolean. 1 means attention occurred; 0 means no attention.

Command Text

CC External Name: CPCTXTCC

R-type Parameters:

R0	address of a parser control block.
R1	address of a 258-byte area in which the result is returned as a halfword length followed by a variable number of characters.
R2	a Boolean value indicating whether to return upper-case (¬0) or mixed-case (=0) form.

Note that the PLUS declaration of this procedure is such that it actually appears to be a function returning a character string, rather than being passed a pointer to one as a parameter.

There is no OS-linkage/S-type version of this routine.

Current Position

CC External Name: CPCURPCC

R-type Parameters:

R0	address of a parser control block.

OS External Name: CPCURP

S-type Parameters:

0(R1)	address of a fullword containing address of a parser control block.

```
Result:  R0      integer specifying position.
```

<u>Last Terminal Text</u>

```
CC External Name:   CPLTTCC

R-type Parameters:
         R0      address of a parser control block.
         R1      address  of  a  258-byte  area  in which the
                 result is  returned  as  a  halfword  length
                 followed by a variable number of characters.
         R2      a Boolean value indicating whether to return
                 upper-case (¬0) or mixed-case (=0) form.

         Note  that the PLUS declaration of this procedure is
         such that it  actually  appears  to  be  a  function
         returning  a  character  string,  rather  than being
         passed a pointer to one as a parameter.

There is no OS-linkage/S-type version of this routine.
```

<u>Parse</u>

```
CC External Name:   CPARSECC

R-type Parameters:
         R0      address of a parser control block.
         R1      address of the goal; V(goalsym).
                 Note  that  this   is   a   PLUS   reference
                 parameter,   so   for   a   PLUS   program,
                 Address(...) is automatically supplied.
         R2      address of first byte of text to be parsed.
         R3      integer length of text to be parsed.

OS External Name:   CPARSE

S-type Parameters:
          0(R1)  address of a fullword containing the address
                 of a parser control block.
          4(R1)  address of a fullword containing V(goalsym).
          8(R1)  address of first byte of text to be parsed.
         12(R1)  address of a fullword containing the integer
                 length of text to be parsed.

Result:  R0      a Boolean  value  indicating  whether  parse
                 succeeded (=0) or failed (=1).
```

<u>Parse Get</u>

```
CC External Name:   CPGETCC
```

R-type Parameters:

R0	address of a parser control block.
R1	integer code for item to get.
R2	address of first byte of area to return value. Note that for a PLUS program the Address(...) is automatically supplied.
R3	integer length of space provided for returning item.
R4	(when required) a Boolean value indicating whether to return upper-case (¬0) or mixed-case (=0) form.

OS External Name: CPGET

S-type Parameters:

0(R1)	address of fullword containing the address of a parser control block.
4(R1)	address of a fullword containing an integer code for item to get.
8(R1)	address of first byte of area to return value.
12(R1)	address of a fullword containing the integer length of space provided for returning item.
16(R1)	(when required) address of a fullword value indicating whether to return upper-case (¬0) or mixed-case (=0) form.

Result: R0   integer specifying actual length of item requested.

<u>Parse Get Stack</u>

CC External Name: CPGSTKCC

R-type Parameters:

R0	address of a parser control block.
R1	integer code for label of element to get.
R2	address of first byte of area to return value. Note that for a PLUS program the Address(...) is automatically supplied.
R3	integer length of space provided for returning item.
R4	(when required) a Boolean value indicating whether to return upper-case (¬0) or mixed-case (=0) form.

OS External Name: CPGSTK

S-type Parameters:

0(R1)	address of fullword containing the address of a parser control block.
4(R1)	address of a fullword containing an integer code for the label of the element to get.

8(R1)	address of first byte of area to return value.
12(R1)	address of a fullword containing the integer length of space provided for returning item.
16(R1)	(when required) address of a fullword value indicating whether to return upper-case (¬0) or mixed-case (=0) form.

Result: R0 integer specifying actual length of item requested.

<u>Parse Get Stack Size</u>

CC External Name: CPGSSCC

R-type Parameters:

R0	address of a parser control block.
R1	integer code for label of element to get.

OS External Name: CPGSS

S-type Parameters:

0(R1)	address of fullword containing the address of a parser control block.
4(R1)	address of a fullword containing an integer code for the label of the element to get.

Result: R0 integer specifying length in bytes of item requested.

<u>Parse Help</u>

CC External Name: CPHELPCC

R-type Parameters:

R0	address of a parser control block.
R1	address of help item in form of halfword length followed by variable number of characters. Note that for a PLUS program the Address(...) is automatically supplied.

OS External Name: CPHELP

S-type Parameters:

0(R1)	address of a fullword containing the address of a parser control block.
4(R1)	address of help item in form of halfword length followed by variable number of characters.

```
Result:  R0      a Boolean.  True  (=1)  indicates  requested
                 item  was found; False (=0) indicates it was
                 not found.
```

<u>Parse Initialize</u>

```
CC External Name:   CPINITCC

R-type Parameters:
         R0      arbitrary fullword (usually a psect address)
                 to pass to semantic routine.

OS External Name:   CPINIT

S-type Parameters:
          0(R1)  address  of  a  fullword  containing  an
                 arbitrary value (usually a psect address) to
                 pass to semantic routine.

Result:  R0      address of a parser control block.
```

<u>Parse Initialize Nonmts</u>

```
CC External Name:   CPINMCC

R-type Parameters:
         R0      arbitrary fullword (usually a psect address)
                 to pass to semantic routine.
         R1      address of a getspace routine.
         R2      address of a freespace routine.

OS External Name:   CPINM

S-type Parameters:
          0(R1)  address  of  a  fullword  containing  an
                 arbitrary value (usually a psect address) to
                 pass to semantic routine.
          4(R1)  address of a fullword containing the address
                 of a getspace routine.
          8(R1)  address of a fullword containing the address
                 of a freespace routine.

Result:  R0      address of a parser control block.
```

<u>The Parser Input Routine</u>

The input routines always take S-type parameters. The call always provides a stack, hence is valid for either OS or CC linkage.

```
S-type Parameters:
          0(R1)  address of a fullword containing the address
                 of a parser control block.
          4(R1)  address of fullword containing the user
                 psect pointer.
          8(R1)  address of area in which to return next line
                 of input.
         12(R1)  address of a fullword integer specifying the
                 number of bytes that may be used for input
                 line.

Result:  R0 integer specifying number of bytes returned.
```

<u>Parse Reset</u>

```
CC External Name:   CPRSETCC

R-type Parameters:
         R0      address of a parser control block.

OS External Name:   CPRSET

S-type Parameters:
          0(R1)  address of a fullword containing the address
                 of a parser control block.

Result:  R0      a Boolean. False (=0) if anything went
                 wrong.
```

<u>Parse Set</u>

```
CC External Name:   CPSETCC

R-type Parameters:
         R0      address of a parser control block.
         R1      integer code for item to set.
         R2      address of first byte of area containing
                 value. Note that for a PLUS program the
                 Address(...) is automatically supplied.
         R3      (when required) length of item passed in R2.

OS External Name:   CPSET

S-type Parameters:
          0(R1)  address of a fullword containing the address
                 of a parser control block.
          4(R1)  address of a fullword containing an integer
                 code for the item to set.
          8(R1)  address of first byte of area containing
                 value.
         12(R1)  (when required) address of fullword
                 containing the length of item passed as
                 parameter 3.
```

```
Result:  R0      a Boolean. True (=1) if item set
                 successfully; False (=0) if not.
```

<u>Parse Terminate</u>

```
CC External Name:   CPTERMCC

R-type Parameters:
         R0      address of a parser control block.

OS External Name:   CPTERM

S-type Parameters:
          0(R1)  address of a fullword containing the address
                 of a parser control block.

Result:  R0      a Boolean. False (=0) if anything went
                 wrong.
```

<u>Production Text</u>

```
CC External Name:   CPPTXTCC

Parameters:
         R0      address of a parser control block.
         R1      address of a 258-byte area in which the
                 result is returned as a halfword length
                 followed by a variable number of characters.
         R2      a Boolean value indicating whether to return
                 upper-case (¬0) or mixed-case (=0) form.

         Note that the PLUS declaration of this procedure is
         such that it actually appears to return a character
         string, rather than being passed a pointer to one as
         a parameter.
```

There is no OS-linkage/S-type version of this routine.

<u>The Semantic Procedures</u>

A semantic procedure is called as a CC-linkage/R-type, unless the declaration in the grammar specifies "S-TYPE".

```
R-type Parameters:
         R0      address of a parser control block.
         R1      psect pointer as passed to Parse_Initialize.
         R2      numeric code for semantic action to be
                 applied.

S-type Parameters:
          0(R1)  address of a fullword containing the address
                 of a parser control block.
          4(R1)  address of a fullword containing the psect
```

```
                pointer as passed to Parse_Initialize.
         8(R1)  address  of  a fullword containing a numeric
                code for semantic action to be applied.

   Result:  R0  a Boolean. True (=1) if semantic  action  is
                to succeed; False (=0) if it is to fail.
```

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX