Parsing, Syntax Checking and Interpreting

User's Guide

Version 2.0

November 13, 2012

Richard Tsujimoto, Inc.

Approval

Version:	1.0
Date:	June 7, 2004
Status:	Draft
Copy no.:	Uncontrolled
Prepared by	Richard Tsujimoto
Reviewed by:	
Approved by:	
Approver's name:	
Approver's title:	
File Location	

Revision History

Date	Ver.	Author	Comments
12-21-2004	1.0	Richard Tsujimoto	Correct some typos, and add a section that describes Statement Continuation
07-14-2005	2.0	Richard Tsujimoto	Add sections on new software for MVS that are compatible with C-versions; also changes where made to the C-versions to provide compatibility with OS/390 V2 changes
07-25-2005	2.0	Richard Tsujimoto	Change parameters for @DELIM macro from OFF to NO, and ON to YES
10-18-2005	2.0	Richard Tsujimoto	Add TOOGLE as delimiter state as a means to tokenize blanks; default action is to not tokenize blanks
12-19-2005	2.0	Richard Tsujimoto	Add description for SYNTBLGEN
02-13-2006	2.0	Richard Tsujimoto	Fix some erroneous, unclear statements
02-14-2006	2.0	Richard Tsujimoto	Add section on building applications
11-07-2012	2.0	Richard Tsujimoto	Add reference to Linux support

Contents

Table Of Figures

1 Introduction

This document is a guide that describes the text processing tools that support parsing, syntax checking and interpretation.

These tools support:

- A consistent method for "tokenizing data"
- A rules-based method for validating the data
- A simple means for extracting data

A separate section is devoted to each tool, and how it is implemented on various platforms.

The tools are available for the following platforms:

- OS/390
- AS/400
- Windows/2000/XP
- AIX
- HP-UX
- Linux[1]

Programming examples are provided, showing how to prepare the parameters/data structures that are required by these tools, and how to invoke these services.

In addition, instructions on how to include these tools during the build of an executable are provided.

1.1 Latest changes

The tools for OS/390 now include software that is wholly compatible with the tools that exist for the distributed platforms. This version of software for OS/390s referred to as Version 2 (V2), as is the software for the distributed platform.

In addition, changes were also made to the C-version to incorporate some improvements made to the OS/390 V2 software.

1.2 Reference Material

ESA/390 Principles of Operation, SA22-7201

HLASM V1R4 Language Reference, SC26-4940

HLASM V1R4 Programmer's Guide, SC26-4941

ILE C for AS/400 Programmer's Guide, SC09-2712

C: The Complete Reference, Herbert Schildt, Osborne McGraw-Hill

[1] Red Hat on Intel 32-bit processor

2 PARSING

2.1 What is Parsing?

One academic perspective of parsing is as follows:

> "Parsing is the process of structuring a linear representation in accordance with a given grammar. The definition has been kept abstract on purpose, to allow as wide an interpretation as possible. The "linear representation" may be a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in a ritual behaviour, in short any linear sequence in which the preceding elements in some way restrict the next element." – *Parsing Techniques, A Practical Guide* by Dick Grune and Ceriel Jacobs

Rather than get into a lengthy discussion on grammar and linguistics, let it suffice to say that grammar is a set of rules that describe a language. There are several ways to represent the set of rules. Compiler writers create parse trees, which are an efficient data structure that supports the validation of complex expressions. But, this level of complexity is beyond the needs for most software tools. A set of linear rules contained in a table should be adequate, even if the functionality it supports is more limited than parse trees.

In order to structure "a linear expression in accordance with a given grammar", the expression must be broken down into its constituent parts, or tokens. The component that preprocesses linear expression, or input characters, into tokens is called a lexical scanner. In this case, the parser is actually nothing more than a lexical scanner. We use a simple a list of punctuation marks and other symbols, e.g. equal sign, which can be used as delimiters, making it easy to extract the elements of the linear expression into tokens.

For example, in the sentence "the ticket costs $4.00", the tokens could be: *the*, *ticket*, *costs*, *$*, and *4.00*. In most implementations, each white space is discarded, or ignored, as opposed to being converted into a token.

The true functions of a parser are to "tokenize" the linear representation, and to validate the input against grammar rules. In our implementation, this process is performed in two separate processes, which we call parsing and syntax checking. The latter process is discussed in a separate section.

2.2 Parsers

The source code and include files (if applicable) for each platform are as follows:

Platform	**File**
OS/390	RTI.MACLIB(@PARSE)
	RTI.SOURCE(@PARSE)
	RTI.SOURCE(T@PARSE)
Non-OS/390[2]	parse.h
	parse.c
	TestPARSE.c
	dfltdelim.h

Figure 1: Parsing source code

2.3 Tokenizing Data

The process of tokenizing data is essentially the same across platforms. As the lexical scanning takes place, an entry is stored on a First-in First-out (FIFO) queue for each token, where each entry contains the following information (see Figure 2: Example of token stack entries**)**.:

[2] AIX, AS/400, HP-UX and Windows

- **Token identifier**[3] - this is a numeric value that represents the type of token. For example, if the expression is a string of characters, e.g. ABCD, it might be assigned 0, whereas if it were an equal sign, it might be assigned 3.
- **Token length** - this is the length of the expression, not including the trailing NULL for C strings.
- **Location of the token** - on OS/390, this would be an actual memory address, whereas on non-OS/390 platforms, this would be an offset into a character string.

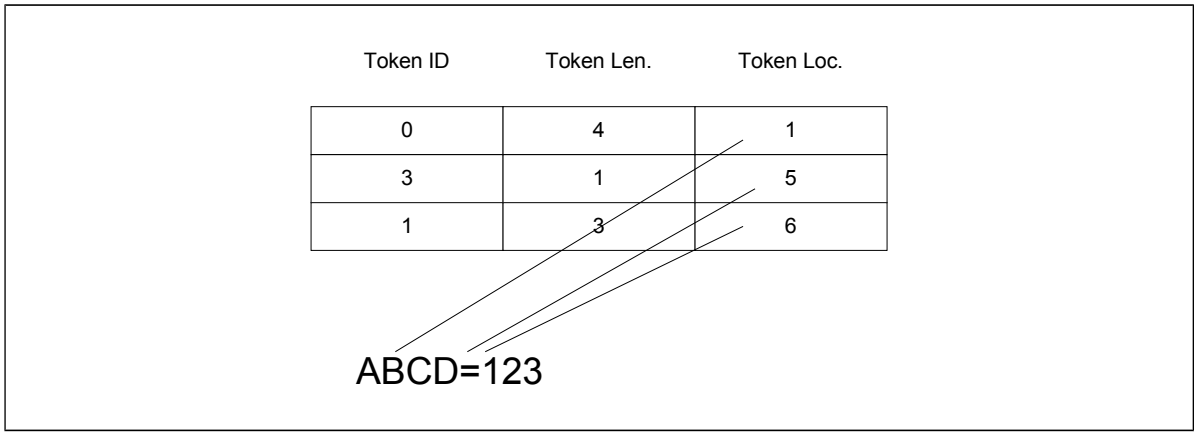

Figure 2: Example of token stack entries

2.3.1 Delimiters

The list of characters that constitute the set of valid delimiters can be modified by the calling application, which gives the caller the ability to treat one, or more delimiters, as ordinary characters. For example, periods are used as part of an OS/390 data set name. Therefore, it may be desirable to treat a period as part of the data set name. If no changes are made, a default set of delimiters is used (see APPENDIX A. Default Delimiters**)**.

2.3.2 Modifying the List of Delimiters

2.3.2.1 OS/390

To modify the list of delimiters for OS/390:

- Code the **`@DELIM`** macro, specifying which special characters are to be treated as delimiters, as opposed to ordinary data.
- By default, each special character is set to **`YES`**, e.g. treat the character as a delimiter. To have a delimiter treated as data, specify **`NO`**.
- Optionally, the value **`TOGGLE`** can be specified. This is the same as **`YES`**, plus it serves as a means to tokenize blanks on and off. For example, a user may want to treat embedded blanks as data, such as a string that is enclosed in double quotes. In this particular case, the user would code **`PTDQUOTE=TOGGLE`**. In effect, this would cause the first double quote to establish that any subsequent blanks to be tokenized until another double quote is encountered.

[3] The actual value assigned on OS/390 is different from that assigned on non-OS/390.

```
DELIMTAB @DELIM MF=GEN,                                                +
               PTDASH=NO,                                              +
               PTUBAR=NO,                                              +
               PTPOUND=NO,                                             +
               PTSLASH=NO,                                             +
               PTCOLON=NO,                                             +
                   .
                   .
                   .
                   .
```

Figure 3: Modified delimiter list on OS/390

2.3.2.2 Non-OS/390

To modify the delimiter list for non-OS/390 platforms:

- Copy **`dfltdelim.h`** into your program
- Change the array name from **`dfltdelim`** to **`userdelim`**.
- By default, each special character is set to **`YES`**, e.g. treat the character as a delimiter. To have a delimiter treated as data, specify **`NO`**.
- Optionally, the value **`TOGGLE`** can be specified. This is the same as **`YES`**, plus it serves as a means to tokenize blanks on and off. For example, a user may want to treat embedded blanks as data, such as a string that is enclosed in double quotes. In this particular case, the user would code **`TOGGLE`**. In effect, this would cause the first double quote to establish that any subsequent blanks to be tokenized until another double quote is encountered.

```
char userdelim[33][2] = { /* User-defined delimiters */
   PTPAD,YES,
   PTCOMMA,YES,
   PTEQUAL,YES,
   PTLPAREN,YES,
   PTRPAREN,YES,
   PTLT,YES,
   PTGT,YES,
   PTLBRACE,YES,
   PTRBRACE,YES,
   PTDASH,YES,
   PTUBAR,YES,
   PTAND,YES,
   PTPOUND,NO,
   PTDQUOTE,TOGGLE,
      .
      .
      .
 }
```

Figure 4: Modified delimiter list on non-OS/390 platforms

2.4 Application Programming Interface

2.4.1 OS/390

The API to invoke the V2 parser on OS/390 is an assembler macro.

```
&LABEL   @PARSE &CMDLINE=,        ADDRESS OF COMMAND LINE STRING       +
                &CMDLEN=,         LENGTH OF COMMAND LINE STRING        +
                &DELIMTB=,        ADDRESS OF DELIMITER ARRAY           +
                &MF=,             MACRO FORMAT                         +
                &DOC=NO           DOCUMENTATION ONLY
Description
Mandatory input parameters: CMDLINE, CMDLEN, DELIMTB
Optional input parameters:  DOC
Output parameters:          n/a

Keyword Values
MF=DSECT      Generate a DSECT of the calling parameter list
MF=L          Generate the calling parameter list
MF=(E,reg)    Generate the execute form of a program call
MF not coded  Generate the inline form of a program call


Registers
R15 = 0       SUCCESS
    = 4       ADDRESS FOR COMMAND STRING IS ZERO
    = 8       STRING HAS INVALID LENGTH
    = 12      ADDRESS FOR DELIMITER ARRAY IS ZERO
```

Figure 5: OS/390 parser API

2.4.2 Non-OS/390

The API to invoke the parser on non-OS/390 platforms is a C function call. See APPENDIX B. parse.h for additional information.

```
extern int parse(char cmdline[],            /* String to be parsed */
                 char delimiterlist[][2]);  /* List of delimiters  */

Description
Mandatory input parameters: cmdline
Optional input parameters:  delimiterlist

Function return values
   0  = Success
   1  = One, or more, parameters is NULL
   2  = cmdline has an invalid length
```

Figure 6: non-OS/390 parser API

2.5 Examples

2.5.1 OS/390

The following example invokes the parser and passes a list of modified delimiters.

```
         @TOKEN   MF=ALL
$PARSMAP @PARSE   MF=DSECT
           .
           .
         LA    R3,PARMLST1        POINT TO PARM LIST
         SPACE 1
         @PARSE MF=(E,R3)
         SPACE 1
         LTR   R15,R15            OK ?
         BZ    PARSE_CMD_EXIT     YES, GET OUT
           .
           .
PARMLST1 @PARSE MF=L,CMDLINE=STRING,CMDLEN=L'STRING,DELIMTB=DELIMTAB
STRING   DS    CL80               INPUT STRING
```

Figure 7: Example of parsing on OS/390

2.5.2 Non-OS/390

The following example invokes the parser, passing a list of modified delimiters.

```
#include <parse.h>
           .
           .
main() {
   char inbuff[MAXSTRINGLEN+1];
           .
           .
   rc = parse(inbuff, userdelim);

   if (rc) { /* Error detected by Parser */
      printf(">>> Error detected by Parser.  rc=%d\n", rc);
      return(ERROR);
   } /* end if */
} /* end main */
```

Figure 8: Example of parsing on non-OS/390 platforms

3 Syntax Checking

3.1 What is Syntax Checking?

One definition of syntax checking that I came across is:

> "A compiler will typically perform syntax checking, which includes type checks, scoping rule enforcement, amongst other checks; and other processes such as static binding, instantiation of templates, and optimization." – *Wikipedia*

As you can see, the definition includes a of number tasks, beyond simply validating the syntactical correctness of an expression. But, in our case, that is all we are interested in, i.e. syntax checking is simply the application a grammar against a linear expression, or input characters. As mentioned earlier, in our implementation, the two functions of a parser, lexical scanning and syntax checking, were divided into two separate processes. The latter is discussed in this section.

Unlike a formal parser, our syntax checker:

- Does not construct or utilize parse trees
- Does not use recursion to find a matching syntax rule
- Does not convert the linear expression into a form suitable for expression analysis, e.g. Reverse Polish Notation (RPN)

The complexity and development time required to implement the full functionality of a true parser are beyond the requirements of most user applications. Hence, a simpler approach was taken.

Instead of using parse trees for validating the linear expression, we employed a simpler approach based on linear rules, similar to a decision table.

Basically, the processing flow is as follows:

1. The current token is compared against a token type in the syntax table and,
2. If it matches, point to the next token in the FIFO queue and take the associated action, e.g. "GOTO" the specified syntax table entry and repeat the process shown in Step 1
3. Otherwise, go to the next row in the table and repeat the process shown in Step 1
4. If the token type is `SYNTAXERROR`, the lookup process ends, the return code is set to a value that indicates a syntax error and control is returned to the caller
5. Otherwise, the process (steps 1-3 above) are repeated until the input data is exhausted; in this case a return code is set to indicate success and control is returned to the caller

Using a decision table-like mechanism is easy to implement, but it does not provide the capability needed for an extensive grammar, e.g. programming language. Yet, this approach is more than adequate for the development of software that uses a limited grammar, such as a small set of commands, or parameters that are used in most user applications.

3.2 Syntax Checkers

The source code and include files (if applicable) for each platform are as follows:

Platform	File
OS/390	RTI.MACLIB(@SYNTXCK)
	RTI.SOURCE(@SYNTXCK)
	RTI.SOURCE(T@SYNTBL)
	RTI.SOURCE(T@SYNTXC)
	RTI.SOURCE(T@UEXITS)
Non-OS/390	syntaxchk.c
	syntaxchk.h
	syntblgen.c
	syntblgen.h

Figure 9: Syntax checking source code

3.3 Grammar

The set of syntax rules, or grammar, is collectively referred to as the syntax table in this document. On OS/390, this is a group of constants generated by the `@RULE` macro, which can be embedded as part of a program, or created as a separate `CSECT`. On non-OS/390 platforms, the syntax table is represented by the multiple instantiations of a C struct variable of type `syntax_table` within an array.

The declaration of the variable type `syntax_table` is in the file `syntaxchk.h`. See APPENDIX C. syntaxchk.h for more details. In addition, the maximum number of syntax rules for non-OS/390 is 1000. This is an arbitrary value, which can be expanded as needed. The user simply has to provide the additional `#define` statements to define the values beyond 1000.

3.3.1 Syntax rule format

The format of a syntax rule is as follows:

`label condition <action> <string> <user exit> <validation exit>`

where:

`label` On OS/390 this is an assembler statement label, but on other platforms this is an index value into the syntax table.

`condition` This is either a token type value, which is used to compare against the current token's type value, or it is a special value, e.g. it denotes the end of a linear expression, or the end of a subset of rules. For details see APPENDIX C. syntaxchk.h and APPENDIX E. OS/390 V2 Token types

`<action>` This is a "GOTO" instruction. On OS/390 it is an actual address of a grammar rule, but on non-OS/390 platforms it is an index value into the array that represents the syntax table. The file `syntaxchk.h` contains 1000 `#define` constants, e.g. `GoTo00` – `GoTo999`, which can be used to simplify the construction of rules. The pointer to the current token is advanced to the next token before the "branch" is taken. This is an optional parameter.

`<string>` If a string value is specified, it is used as a secondary comparison against the current token, after the token type test is satisfied. This is an optional parameter. The maximum length of a string is 100 characters.

<user exit> This is the address of user routine that is called during the interpretation phase. This is an optional parameter.

3.4 Syntax table examples

Using the grammar rules as described above, an example of a syntax table for each version and/or environment is presented.

The examples are based on the following linear expression:

```
ABCD=123
```

and the string value assigned to the keyword `ABCD` is to be extracted by the user exit called `MYEXIT`.

3.4.1 OS/390

```
SYNTABLE @RULE TYPE=INITIAL
RULE010  @RULE TOK_IS_DATA,NEXT=RULE020,STRING=ABCD
         @RULE SYNTAXERR
RULE020  @RULE TOK_IS_EQUAL,NEXT=RULE030
         @RULE SYNTAXERR
RULE030  @RULE TOK_IS_NUM,NEXT=FLUSH,STRING=123,EXIT=MYEXIT
         @RULE SYNTAXERR
FLUSH    @RULE TOK_IS_EOS,NEXT=DONE
         @RULE SYNTAXERR
DONE     @RULE LASTRULE
         @RULE TYPE=FINAL
```

Figure 10: Example of an OS./390 syntax table

3.4.2 Non-OS/390

```
struct syntax_table syntaxtab[100] = { /* User syntax rules */
                    /* rule-00 */ {STARTRULE},
                    /* rule-01 */ {TokIsData,  GoTo3, "ABCD"},
                    /* rule-02 */ {SYNTAXERR},
                    /* rule-03 */ {TokIsEqual, GoTo5},
                    /* rule-04 */ {SYNTAXERR},
                    /* rule-05 */ {TokIsNum,   GoTo7, "123", &MYEXIT},
                    /* rule-06 */ {SYNTAXERR},
                    /* rule-07 */ {TokIsEOS,   GoTo9},
                    /* rule-08 */ {SYNTAXERR},
                    /* rule-09 */ {LASTRULE} };
```

Figure 11: Example of a non-OS/390 syntax table

3.5 Statement continuation

The continuations of input statements is a common practice when commands tend to have a complicated syntax that may, or may not, involve long data values. Any token can be chosen as the continuation character, such as a plus sign. A special token type value is provided (**`CONTRULE`**), which instructs the syntax checker that the statement will be continued and that it should record where syntax checking is to resume when the next input string is processed.

3.5.1 OS/390 and Non-OS/390

The user tests for the presence of the character chosen for the statement continuation character. The character can be any character that is valid for the given platform. Once a match has been made, a rule is branched to that has the special keyword **CONTRULE**, followed by the statement where syntax checking is to resume for next statement. Obviously, the placement and usage of a continuation character can be as liberal as one chooses, but the trade-off is added complexity to the syntax rules.

Since the continuation rule uses the "GOTO" location associated with the **<action>** value for a future purpose, e.g. the location of the syntax rule where syntax checking is to resume is stored, a mechanism is required to control the logic flow after **CONTRULE**. Hence, a special token value is available that meets this need, which is called **GOTORULE**. The purpose of this token type is to provide the "GOTO" capability that was not provided for in the **CONTRULE** token type. This special token type **must** follow the **CONTRULE** token type.

For example, the following statements show a command that supports two types of values, and calls **MYEXIT** to extract the values:

```
ABCD = aaaa          (where aaaa is any string other than HELP)
ABCD = HELP
```

If a plus sign (+) is used as a continuation character, there could be 2 places where it could be used (assuming no blank statements are used):

```
ABCD +
       =
         +
         aaaa
         HELP
```

It should be pointed out that the degree of flexibility has a direct impact on the number of syntax rules required to support that flexibility.

Since the example of the syntax table that demonstrates support for statement continuation for the above expression is nearly identical for both OS/390 and non-OS/390 platforms, only the non-OS/390 example is shown.

```
struct syntax_table syntaxtab[100] = { /* User syntax rules */
                    /* rule-00 */ {STARTRULE},
                    /* rule-01 */ {TokIsData,  GoTo06, "ABCD"},
                    /* rule-02 */ {SYNTAXERR},
                    /* rule-03 */ {TokIsEOS,   GoTo05},
                    /* rule-04 */ {SYNTAXERR},
                    /* rule-05 */ {LASTRULE},
                    /* rule-06 */ {TokIsEqual, GoTo12},
                    /* rule-07 */ {TokIsPlus,  GoTo09},
                    /* rule-08 */ {SYNTAXERR},
                    /* rule-09 */ {CONTRULE,   GoTo12},
                    /* rule-10 */ {GOTORULE,   GoTo03},
                    /* rule-11 */ {SYNTAXERR},
                    /* rule-12 */ {TokIsData,  GoTo03, "HELP", &MYEXIT},
                    /* rule-13 */ {TokIsData,  GoTo03, NULL,   &MYEXIT},
                    /* rule-14 */ {SYNTAXERR} };
```

Figure 12: Example of syntax rules for Statement Continuation on non-OS/390 platforms

3.6 Application Programming Interface

The following sections describe how to invoke the syntax checker, and the parameters that are required for each version and/or environment.

3.6.1 OS/390

The API to invoke the syntax checker on OS/390 is an assembler macro.

```
&LABEL   @SYNTXCK &SYNTXTB=,      ADDRESS OF SYNTAX TABLE              +
                  &MF=,           MACRO FORMAT                         +
                  &DOC=NO         DOCUMENTATION ONLY

Description
Mandatory input parameters: SYNTXTB
Optional input parameters:  DOC
Output parameters:          Address of token in error[4]
                            Length of token in error
                            Address of bad rule

Keyword Values
MF=DSECT      Generate a DSECT of the calling parameter list
MF=L          Generate the calling parameter list
MF=(E,reg)    Generate the execute form of a program call
MF not coded  Generate the inline form of a program call

Registers
R15 = 0       SUCCESS
    = 4       ADDRESS OF USER'S SYNTAX TABLE IS ZERO
    = 8       SYNTAX ERROR DETECTED
    = 12      NEXT RULE INDEX IS ZERO
```

Figure 13: OS/390 syntax checker API

3.6.2 Non-OS/390

The API to invoke the syntax checker on non-OS/390 platforms is a C function call. See APPENDIX C. syntaxchk.h for additional information.

```
extern int syntaxchk(char cmdline[],      /* String to be checked     */
                     struct syntax_table *syntaxtab,  /* Syntax rules */
                     int *tokenloc,       /* Index value of bad token */
                     int *tokenlen,       /* Length of bad token      */
                     int *syntaxloc);     /* Index of bad rule        */

Description
Mandatory input parameters: cmdline, syntaxtab
Optional input parameters:  n/a
Output parameters:          tokenloc, tokenlen, syntaxloc

Function return values
   0  = success
```

[4] See DSECT of OS/390 calling parameter list

```
   2  = syntax error detected
   3  = One, or more parameters is NULL
   4  = "Next Rule" index is zero
   5  = STARTRULE not found as first table entry[5]
```

Figure 14: non-OS/390 syntax checker API

3.7 Programming examples

The following examples will be based on syntax checking the following linear expression:

```
ABCD  =   123
```

where, embedded spaces are allowed, and a user exit called **MYEXIT** is to be called during the interpretation phase to process the assigned data.

It is assumed that the parser successfully processed the input characters in an earlier step.

3.7.1 OS/390

```
         @TOKEN   MF=ALL
$PARSMAP @PARSE   MF=DSECT
$SYNTMAP @SYNTXCK MF=DSECT
           .
           .
         LA    R3,PARMLST2         POINT TO @SYNTXCK PARM LIST
         LA    R5,SYNTABLE         POINT TO SYNTAX TABLE
         SPACE 1
         @SYNTXCK MF=(E,R3),SYNTXTB=(R5)
         SPACE 1
         LTR   R15,R15             ANY ERRORS?
         BZ    CHK_SYNTAX_EXIT     NO, GET OUT
           .
           .
PARMLST2 @SYNTXCK MF=L
           .
           .
SYNTABLE @RULE TYPE=INITIAL,EXITLOC=LOCAL
RULE010  @RULE TOK_IS_DATA,NEXT=RULE020,STRING=ABCD
         @RULE SYNTAXERR
RULE020  @RULE TOK_IS_EQUAL,NEXT=RULE030
         @RULE SYNTAXERR
RULE030  @RULE TOK_IS_DATA,NEXT=FLUSH,EXIT=MYEXIT
         @RULE SYNTAXERR
FLUSH    @RULE TOK_IS_EOS,NEXT=DONE
         @RULE SYNTAXERR
DONE     @RULE LASTRULE
         @RULE TYPE=FINAL
```

Figure 15: Example of syntax checking on OS/390

[5] STARTRULE is used to occupy the first element in the syntax table, which is an array in C. This prevents using array position zero as a valid "GOTO" location. This is not an issue on OS/390.

3.7.2 Non-OS/390

```
#include <parse.h>
#include <syntaxchk.h>
extern int MYEXIT(char token[], int tokenlen);

main()
{
  char inbuff[MAXSTRINGLEN+1];
  int  toklen;
  int  tokloc;

  struct syntax_table syntaxtab[100] = { /* User syntax rules */
                   /* rule-00 */ {STARTRULE},
                   /* rule-01 */ {TokIsData,  GoTo03, "ABCD"},
                   /* rule-02 */ {SYNTAXERR},
                   /* rule-03 */ {TokIsEqual, GoTo05},
                   /* rule-04 */ {SYNTAXERR},
                   /* rule-05 */ {TokIsNum,   GoTo07, NULL, &MYEXIT},
                   /* rule-06 */ {SYNTAXERR},
                   /* rule-07 */ {TokIsEOS,   GoTo09},
                   /* rule-08 */ {SYNTAXERR},
                   /* rule-09 */ {LASTRULE} };
           .
           .
     rc = syntaxchk(inbuff, syntaxtab, &tokloc, &toklen);

     if (rc) { /* syntax error found */
        printf(">>> Syntax error in column %d token length = %d\n",
               tokloc + 1, toklen);
     } /* end if */
           .
           .
```

Figure 16: Example of syntax checking on non-OS/390 platforms

3.8 Maintaining syntax tables for non-OS/390 platforms

The major difference between the specification of syntax rules between OS/390 and non-OS/390 environments is the use of a macro language on OS/390 vs. using an array in a C program. The macro language allows one to specify labels, which are treated as relocatable symbols. Hence, if the syntax rules need changes, it becomes a trivial task with respect to editing source code. But, in the case of C programs, since each rule is, in effect, an entry in an array, the impact to the relationship between array locations and "GoTo" statements is significant. For example, the insertion of a single rule in an array, other than at the end of the array, affects the relative location of each array entry from that point on, and on any rule that references the affected entries via a "GoTo" statement. Obviously, the effort to maintain a syntax table in a C program could potentially outweigh the benefits of using the in-house tools.

As a means to alleviate the effort of maintaining a syntax table in a C program, two options are presented.

3.8.1 DUMMYRULE

A special token type called `DUMMYRULE` is available, which is used to reserve space in the event of future changes. Since entries in C struct variables are non-relocatable, reserving space minimizes the impact of

making changes later on. This approach may be desirable if the syntax table is small, and not prone to changes.

The following example shows the same syntax rules used in prior examples, except for a few DUMMYRULE's. Note that the "GoTo" statements had to be modified, so that the offset is referenced.

```
extern int MYEXIT(char token[], int tokenlen);

main()
{
  char inbuff[MAXSTRINGLEN+1];
  int  toklen;
  int  tokloc;

  struct syntax_table syntaxtab[100] = { /* User syntax rules */
                    /* rule-00 */ {STARTRULE},
                    /* rule-01 */ {TokIsData,  GoTo04, "ABCD"},
                    /* rule-02 */ {SYNTAXERR},
                    /* rule-03 */ {DUMMYRULE},
                    /* rule-04 */ {TokIsEqual, GoTo07},
                    /* rule-05 */ {SYNTAXERR},
                    /* rule-06 */ {DUMMYRULE},
                    /* rule-07 */ {TokIsNum,   GoTo09, NULL, &MYEXIT},
                    /* rule-08 */ {SYNTAXERR},
                    /* rule-09 */ {TokIsEOS,   GoTo11},
                    /* rule-10 */ {SYNTAXERR},
                    /* rule-11 */ {LASTRULE} };
            .
            .
     rc = syntaxchk(inbuff, syntaxtab, &tokloc, &toklen);

     if (rc) { /* syntax error found */
        printf(">>> Syntax error in column %d token length = %d\n",
               tokloc + 1, toklen);
     } /* end if */
            .
            .
```

Figure 17: Using DUMMYRULE to reserve space in the syntax table

3.8.2 SYNTBLGEN

The second option that can be used to maintain syntax tables for C programs is to use another in-house tool called SYNTBLGEN. This tool is a C program that converts a macro-like source file into a file that contains an array, which has the user-specified syntax rules. As a result, the programmer does not have to worry about maintaining the relationship between specific array entries and "GoTo" statements.

The macro-like source is based on the macro statements used in the OS/390 environment. In effect, if the same rules are used on OS/390 and other non-OS/390 platforms, a single set of rules could be maintained on OS/390 itself, with a few minor changes.

The differences between the macro language used on OS/390 and non-OS/390 are as follows:

- The first rule must be **@RULE TYPE=INITIAL** without any other parameters

- OS/390-specific token types are not supported, e.g. `TOK_IS_NOT` (the not sign).

- The parameter **`COMMENT="…"`** is only supported for non-OS/390. This parameter provides a way for a programmer to specify a comment for a rule that is also generated as a comment for an array element.

3.8.2.1 Command syntax for SYNTBLGEN

```
syntblgen –i infile –o outfile


where infile is the input source file that contains the macro statements
      outfile is the output file that contains the C array statements
```

Figure 18: Command syntax for SYNTBLGEN

3.8.2.2 Example using SYNTBLGEN

The following examples show the macro statements that are used to define the same set of syntax rules, as in the prior examples, and the generated C arrays statements.

```
SYNTABLE @RULE TYPE=INITIAL
RULE010  @RULE TOK_IS_DATA,NEXT=RULE020,STRING=ABCD,                  +
               COMMENT="Start: process ABCD"
         @RULE SYNTAXERR
RULE020  @RULE TOK_IS_EQUAL,NEXT=RULE030
         @RULE SYNTAXERR
RULE030  @RULE TOK_IS_DATA,NEXT=FLUSH,EXIT=MYEXIT
         @RULE SYNTAXERR,COMMENT="End:  process ABCD"
FLUSH    @RULE TOK_IS_EOS,NEXT=DONE
         @RULE SYNTAXERR
DONE     @RULE LASTRULE
         @RULE TYPE=FINAL
```

Figure 19: Sample input file for SYNTBLGEN

The above example is a copy of the macro statements taken from the OS/390 example, with the following changes:

- The **`@RULE TYPE=INITIAL`** statement does not have the **`EXITLOC=LOCAL`** parameter

- **`COMMENT="…"`** parameters has been added.

- A plus sign (+) has been added, which is used to denote a continuation statement.

```
struct syntax_table syntaxtab[MAX_RULES] = { /* Syntax rules */
 /* rule-0000 */ {STARTRULE},
 /* rule-0001 */ {TokIsData, GoTo3, "ABCD"}, /* Start: process ABCD */
 /* rule-0002 */ {SYNTAXERR},
 /* rule-0003 */ {TokIsEqual,       GoTo5},
```

```
 /* rule-0004 */ {SYNTAXERR},
 /* rule-0005 */ {TokIsData,        GoTo7, NULL, &MYEXIT},
 /* rule-0006 */ {SYNTAXERR},                /* End:  process ABCD */
 /* rule-0007 */ {TokIsEOS,         GoTo9},
 /* rule-0008 */ {SYNTAXERR},
 /* rule-0009 */ {LASTRULE},
}; /* end of syntax table */
```

Figure 20: Output file generated by SYNTBLGEN

The above example shows the result of using SYNTBLGEN[6]. Note that **MAX_RULES** has been inserted as the size of the array. **MAX_RULES** is a **#define** variable set to 1000. The programmer can change this to a smaller value, if necessary.

[6] The lines were shifted left so that the comment lines are not split across two lines. This was to make the example more readable for this document. In a real C program, this would not be an issue.

4 Interpreting

4.1 What is interpreting?

The explanation of this term could be derived from the understanding of what an interpreter does.

> "A program that executes instructions written in a high-level language. There are two ways to run programs written in a high-level language. The most common is to compile the program; the other method is to pass the program through an interpreter.
>
> An interpreter translates high-level instructions into an intermediate form, which it then executes. In contrast, a compiler translates high-level instructions directly into machine language." – *Wikipedia*

An example of an interpreter is the BASIC language, which is widely known as an interpretive language[7].

But, in our particular case, interpreting is a process that is designed to provide a means for giving the application access to tokens once the linear expression has been tokenized and checked for syntactical correctness. Unlike a formal interpreter, the linear expression is **no**t stored in an intermediate form. But the syntax rules that pertain to that linear expression can be viewed as the intermediate format of the linear expression itself.

The same grammar that was used as input to the syntax checker is also used as input to the interpreter. But, unlike the syntax checker, the interpreter uses the syntax rules for the linear expression as instructions to be processed, or followed, looking for rules that contain requests to invoke user exits. The location and length of the token are then passed to the user exit which, after local processing, returns control to the interpreter, indicating if it had encountered any context-related errors. For example, the interpreter passes a value of 100 to the user exit, but the user exit discovers that value exceeds the maximum allowed for the token.

4.2 Interpreters

The source code and include files (if applicable) for each platform are as follows:

Host	File
OS/390	RTI.MACLIB(@INTRPTR)
	RTI.SOURCE(@INTRPTR)
Non-OS/390	interpret.h
	interpret.c

Figure 21: Interpreter source code

4.3 Application Programming Interface

4.3.1 OS/390

The API to invoke the interpreter on OS/390 is an assembler macro.

```
&LABEL   @INTRPRT &SYNTXTB=,      ADDRESS OF USER'S SYNTAX TABLE       +
                  &ERRMSG=,       ADDRESS OF 100-BYTE MESSAGE BUFFER   +
                  &MF=,           MACRO FORMAT                         +
                  &DOC=NO         DOCUMENTATION ONLY

Description
Mandatory input parameters: SYNTXTB
```

[7] Some implementations of BASIC languages have the option of being compiled as well as interpreted.

```
Optional input parameters:  DOC
Output parameters:          ERRMSG


Registers
R15 =  0 - SUCCESS
       4 - ADDR OF USER'S SYNTAX TABLE IS ZERO, OR
           ADDR OF USER'S MESSAGE BUFFER IS ZERO
       8 - ERROR DETECTED BY USER'S EXIT
```

Figure 22: OS/390 interpreter API

4.3.2 Non-OS/390

The API to invoke the interpreter on non-OS/390 platforms is a C function call. See APPENDIX D. interpret.h for additional information.

```
extern int interpret(char cmdline[],      /* String to be interpreted */
                     struct syntax_table *syntaxtab); /* Syntax rules */

Description
Mandatory input parameters: cmdline, syntaxtab
Optional input parameters:  n/a
Output parameters: n/a

Function return values
   0           success
   1           context error detected by user exit
   2           Parameter is null
```

Figure 23: non-OS/390 interpreter API

4.4 User exits

User exits are subroutines that are invoked when the linear expression is passed through the syntax table and the interpreter encounters a user exit specification for a given syntax rule. This provides the means by which the user code can access data from a linear expression and process it.

4.4.1 OS/390 user exits

4.4.1.1 Syntax rule specifications

The exits can be specified on any syntax rule, except for:

```
@RULE TYPE=INITIAL
@RULE TYPE=FINAL
```

The format for specifying user exits on a syntax rule is as follows:

```
@RULE TYPE=token_type,NEXT=label[,STRING=s…s],EXIT=e…e
```

where `e…e` is the subroutine's name. The address constant that is generated in the syntax table is either an `ACON`, or a `VCON`, is determined by the `@RULE TYPE=INITIAL` macro:

`@RULE TYPE=INITIAL,EXITLOC=LOCAL` generates `ACON`s (this is the default), and

`@RULE TYPE=INITIAL,EXITLOC=EXTERNAL` generates `VCON`s

Note, if the user exit is part of a separate `CSECT`, but is not the main entry point, then an `ENTRY e…e` assembler statement must be specified in that `CSECT`.

For an example, see Figure 10: Example of an OS./390 syntax table.

4.4.1.2 Calling the user exit

When the interpreter encounters a user exit specification in a syntax rule, it branches to the user exit, passing a parameter list via register 1.

Register 1 points to the following 3-word parameter list:

Address of the token
Length of the token
Address of a 100 byte message buffer

The message buffer is provided to the user exit in the event it chooses to store an informational/error message.

The user exit signifies if its processing is to be regarded as successful, or not, by setting register 15 to 0 (success), or 8 (error).

4.4.2 Non-OS/390 user exits

4.4.2.1 Syntax rule specifications

The exits can be specified on any syntax rule, except for:

```
STARTRULE
```

The format for specifying user exits on a syntax rule is as follows:

```
TokenType, GoToxx [, "s…s"], &e…e
                  [, NULL]
```

where `s…s` is a string value and `e…e` is the name of the user exit. And, in the case where there is no string to compare, `NULL` serves as a placeholder and ensures that an empty string pointer is to be generated in the syntax table.

For an example, see Figure 11: Example of a non-OS/390 syntax table.

4.4.2.2 Calling the user exit

User exit names must be defined by specifying an external function prototype statement for the user exit.

For example:

```
extern int MYEXIT(char token[], int tokenlen, char *errmsg);
```

When the interpreter encounters a user exit specification in a syntax rule, it invokes the user exit, passing string pointers to the token and a 100-byte message buffer (not including the null terminator), and an integer value that reflects the length of the token (again, not including the null terminator).

The message buffer is provided to the user exit in the event it chooses to store an informational/error message.

The user exit signifies if its processing is to be regarded as successful, or not, by setting the parameter in the `return()` function to 0 (success), or 1 (error).

4.5 Programming examples

The following examples show how to invoke the interpreter, along with a simple user exit. The code that invokes the parser and syntax checker is also included for readability sake. In addition, the syntax tables are not shown, but it can be assumed that the tables being used are the same one as ones described in the section Syntax table examples**.**

4.5.1 OS/390

```
#PARSMAP @PARSE MF=DSECT
#SYNTMAP @SYNTXCK MF=DSECT
#INTRMAP @INTRPRT MF=DSECT
#EXITMAP @INTRPRT MF=EXITPARM
         @TOKEN MF=ALL
           .
           .
         LA    R3,PARMLST1        POINT TO PARM LIST
         SPACE 1
         @PARSE MF=(E,R3)
         SPACE 1
         LTR   R15,R15            OK?
         BNZ   PARSE_CMD_ERR      NO, CONTINUE
         SPACE 1
         LA    R3,PARMLST2        POINT TO @SYNTXCK PARM LIST
         LA    R5,SYNTABLE        POINT TO SYNTAX TABLE
         SPACE 1
         @SYNTXCK MF=(E,R3),SYNTXTB=(R5)
         SPACE 1
         LTR   R15,R15            ANY ERRORS?
         BNZ   SYNTAX_ERR         YES, CONTINUE
         SPACE 1
         LA    R3,PARMLST3        POINT TO @INTRPRT PARM LIST
         LA    R5,SYNTABLE        POINT TO SYNTAX TABLE
         SPACE 1
         @INTRPRT MF=(E,R3),SYNTXTB=(R5)
         SPACE 1
         LTR   R15,R15            ANY ERROR?
         BZ    CHK_INTRPRT_EXIT   NO, GET OUT
           .
           .
PARMLST1 @PARSE MF=L,CMDLINE=INBUFF,CMDLEN=L'INBUFF,DELIMTB=DELIMTAB
PARMLST2 @SYNTXCK MF=L
PARMLST3 @INTRPRT MF=L,ERRMSG=IERRMSG
INBUFF   DS    CL80               INPUT  BUFFER
           .
           .
MYEXIT   DS    0H
         PUSH  USING
         SAVE  (14,12)            SAVE CALLER'S REGS
```

```
         SPACE 1
         LR    BASEREG,R15         PRIME BASE REG
         USING MYEXIT,BASEREG      SET ADDR
         SPACE 1
         ST    R13,SUBSAVE+4       SAVE PTR TO CALLER'S REG. SAVE AREA
         LA    R13,SUBSAVE         PRIME SAVE AREA PTR
         LR    R7,R1               PARM ADDR
         USING #EXITMAP,R7         SET ADDR
         SPACE 1
         CLC   NUM123,=CL3' '      DUPLICATE?
         BNE    ITSADUP            YES, CONTINUE
         SPACE 1
         L     R2,@IXTOKAD         ADDR OF TOKEN
         L     R4,@IXTOKLN         TOKEN LENGTH
         BCTR  R4,R0               MACHINE LENGTH
         EX    R4,COPY123          COPY IT
         XR    R15,R15             SET GOOD RC
         B     EXIT                GET OUT
         SPACE 1
ITSADUP  DS    0H
         L     R2,@IXERRMG         ADDR OF MESSAGE BUFFER
         MVC   0(L'DUPMSG,R2),DUPMSG  STORE MESSAGE
         LA    R15,8               SET BAD RC
         SPACE 1
EXIT     DS    0H
         L     R13,SUBSAVE+4       PT TO CALLER'S REG SAVEAREA
         SPACE 1
         RETURN (14,12),RC=(15)    GO BACK TO @INTRPRT
         SPACE 1
NUM123   DC    CL3' '              123
DUPMSG   DC    C'>>> DUPLICATE 123 <<<'
         SPACE 1
         DROP  R7
         POP   USING
```

Figure 24: Example of OS390 V2 user exit

4.5.2 Non-OS/390

```
#include <parse.h>
#include <syntaxchk.h>
#include <interpret.h>

extern int MYEXIT(char token[], int tokenlen, char *errmsg);

char num123[4] = "   ";

main()
{
  char inbuff[MAXSTRINGLEN+1];
  int  toklen;
  int  tokloc;
          .
          .
  rc = parse(inbuff, userdelim);
```

```
  if (rc) { /* Error detected by Parser */
     pritnf(“>>> Error detected by Parser, rc=&d\n”, rc);
     return(ERROR);
  } /* end if */

  rc = syntaxchk(inbuff, syntaxtab, &tokloc, &toklen);

  if (rc) { /* syntax error found */
     printf(">>> Syntax error in column %d token length = %d\n",
            tokloc + 1, toklen);
     return(ERROR);
  } /* end if */

  rc = interpret(inbuff, syntaxtab);

  if (rc) { /* context error encountered */
     printf("+++ rc returned by interpret = %d\n", rc);
     return(ERROR);
  } /* end if */
           .
           .
int MYEXIT(char token[], int tokenlen, char *errmsg)
{
  int i;

  for (i = 0; i < tokenlen; i++) {
     if (num123[i] != ' ') {
        strcat(errmsg, “>>> DUPLICATE 123 <<<”;
        return(ERROR);
     } /* end if */
  } /* end for */

  return(SUCCESS);

} /* end of MYEXIT */

} /* end main */
```

Figure 25: Example of non-OS/390 user exit

5 Installation and build instructions

This section describes how to build the tools and sample programs, and how to include these tools in an application.

NOTE: due to problems detected in the packaging of these tools by NaSPA, the CBT tape should not be used. Instead the packaging has been redone and is available on a CD which can be obtained by contacting the author by sending an email to:

rtsujimoto@nyc.rr.com

5.1 Installation

Follow the installation instructions on the author-supplied CD

5.2 Building the tools and sample programs

The platform-specific readme file also describes how to build the tools and sample programs.

APPENDIX A. Default Delimiters

Delimiter	Name	Comments
	Space	
,	Comma	
=	Equal sign	
(	Left parenthesis	
)	Right parenthesis	
<	Left angle bracket	
>	Right angle bracket	
{	Left brace	
}	Right brace	
-	Dash	
_	Under-bar	
&	Ampersand	
#	Pound sign	
@	At sign	
+	Plus sign	
/	Slash	
*	Asterisk	
;	Semi-colon	
:	Colon	
‘	Single-quote	
“	Double-quote	
¬	Not-equal	OS/390 only
~	Tilde	
|	Bar	
?	Question mark	
.	Period	
!	Exclamation point	
¢	Cent sign	OS/390 only
\	Back slash	
$	Dollar sign	
¦	Split bar	OS/390 only
`	Reverse quote	
[	Left bracket	Non-OS/390 only
]	Right bracket	Non-OS/390 only

APPENDIX B. parse.h

```
/*****************************Documentation Start*******************************

  NAME: parse.h  - Header file for parser.c and user code that invokes parser.c


  DESCRIPTION:

  This file contains definitions used by parser.c, and callers of parser.c


  NOTE: parser.c must specify #define PARSE  1 to ensure the global variables it
        uses are exposed.


  HISTORY:

  Date        Who Description
  ----------- --- ------------------------------------------------------------
  2004-Jun-01 RXT Created
  2005-Jul-08 RXT Add specific return codes for each error type

*****************************Documentation End********************************/

#define PARSE_PARM_IS_NULL          1
#define PARSE_STRING_LEN_INVALID    2

#define YES             'Y'
#define NO              'N'
#define TOGGLE          'T'
#define TOGGLE_ON       1
#define DEFAULT_DELIMS  dfltdelim
#define MAXSTRINGLEN    100
#define MAX_KEYWORD_LEN 100

#define PTDATA          100
#define PTNUM           101
#define PEOS            255
#define PTNULL          '\0'
#define PTPAD           ' '
#define PTCOMMA         ','
#define PTEQUAL         '='
#define PTLPAREN        '('
#define PTRPAREN        ')'
#define PTLT            '<'
#define PTGT            '>'
#define PTLBRACE        '{'
#define PTRBRACE        '}'
#define PTDASH          '-'
#define PTUBAR          '_'
#define PTAND           '&'
#define PTPOUND         '#'
#define PTAT            '@'
#define PTPLUS          '+'
#define PTSLASH         '/'
#define PTPERCENT       '%'
#define PTSTAR          '*'
#define PTSCOLON        ';'
#define PTCOLON         ':'
#define PTSQUOTE        '\''
```

```
#define PTDQUOTE        '"'
#define PTTILDE         '~'
#define PTBAR           '|'
#define PTQUEST         '?'
#define PTPERIOD        '.'
#define PTEXCLAM        '!'
#define PTBSLASH        '\\'
#define PTDOLLAR        '$'
#define PTRVQUOT        '`'
#define PTLBRACKET      '['
#define PTRBRACKET      ']'

#if PARSE != 1
extern char dfltdelim[33][2];

extern struct parsework {  /* Stack of parsed tokens */
          int tokentype;   /* token type flag        */
          int tokenlen;    /* token length           */
          int tokenloc;    /* token offset in string */
} tokenstack[MAXSTRINGLEN+1];
#endif

extern int parse(char   cmdline[],                  /* String to be parsed */
                 char   delimiterlist[][2]);        /* List of delimiters  */
```

APPENDIX C. syntaxchk.h

```
/************************Documentation Start***********************************

  NAME: syntaxchk.h  - Header file for syntaxchk.c and user code that invokes
                       syntaxchk.c


  DESCRIPTION:

  This file contains definitions used by syntaxchk.c, and callers of syntaxchk.c


  NOTE: syntaxchk.c must specify #define SYNTAXCHK  1 to ensure the global
        variables it uses are exposed.


  HISTORY:

  Date        Who Description
  ----------- --- ------------------------------------------------------------
  2004-Jun-01 RXT Created

************************Documentation End*************************************/

#define TokIsData              100
#define TokIsNum               101
#define TokIsEOS               255
#define TokIsBlank             0
#define TokIsComma             1
#define TokIsEqual             2
#define TokIsLeftParen         3
#define TokIsRightParen        4
#define TokIsLessThan          5
#define TokIsGreaterThan       6
#define TokIsLeftBrace         7
#define TokIsRightBrace        8
#define TokIsDash              9
#define TokIsUnderBar          10
#define TokIsAnd               11
#define TokIsPound             12
#define TokIsAt                13
#define TokIsPlus              14
#define TokIsSlash             15
#define TokIsPercent           16
#define TokIsAsterisk          17
#define TokIsSemiColon         18
#define TokIsColon             19
#define TokIsSingleQuote       20
#define TokIsDoubleQuote       21
#define TokIsTilde             22
#define TokIsBar               23
#define TokIsQuestion          24
#define TokIsPeriod            25
#define TokIsExclamation       26
#define TokIsBackSlash         27
#define TokIsDollar            28
#define TokIsReverseQuote      29
#define TokIsLeftBracket       30
#define TokIsRightBracket      31
```

```
#define DUMMYRULE               48059   /* Used to reserve space for a rule set
for future changes */
#define CONTRULE                52428   /* Indicates a statement continuation
delimiter encountered*/
#define SYNTAXERR               56797   /* Indicates a syntax error found
*/
#define LASTRULE                61166   /* Marks the successful end of a rule
"path"               */

/* "Next Rule" markers
   These values reflect the location of the next "rule" to be applied during
syntax checking.  In effect,
   the values reflect the indexed location of the struct entry that contains the
"rule".  Since these
   are hard-coded offsets, care should be taken when changes are made.
   NOTE: 1. use DUMMYRULE to reserve space for future changes
         2. add more "next rule" markers as needed. */

#define GoTo00           0
#define GoTo01           1
#define GoTo02           2
   .
   .
   .
#define GoTo599        599
#define GoTo0600       600

struct syntax_table { /* Table of syntax rules       */
   int tokentype;     /* token type flag             */
   int nextrule;      /* next syntax rule to process */
   char *keyword;     /* optional keyword string     */
   int (*userexit)(); /* optional user exit          */
};

#if SYNTAXCHK == 1
int contrule_loc = -1;  /* Loc of syntax rule to execute due to a statement
continuation */
#else
extern int syntaxchk(char cmdline[],              /* String to be checked     */
                     struct syntax_table *syntaxtab, /* Table of syntax rules */
                     int *tokenloc,               /* Index value of bad token */
                     int *tokenlen);              /* Length of bad token      */
#endif
```

APPENDIX D. interpret.h

```
/************************Documentation Start***********************************

  NAME: interpret.h  - Header file for interpret.c and user code that invokes
                       interpret.c


  DESCRIPTION:

  This file contains definitions used by interpret.c, and callers of interpret.c


  NOTE: interpret.c must specify #define INTERPRET  1 to ensure the global
        variables it uses are exposed.


  HISTORY:

  Date        Who Description
  ----------- --- ------------------------------------------------------------
  2004-Jun-01 RXT Created

************************Documentation End*************************************/

#if INTERPRET == 1
int fail_rc;            /* Error rc returned from user exit */
int contrule2_loc = -1; /* Loc of syntax rule to execute due to a statement
                           continuation */

int interpret(char cmdline[],                 /* String to be interpreted */
              struct syntax_table *syntaxtab); /* Table of syntax rules    */
#else
extern int interpret(char cmdline[],          /* String to be interpreted */
                     struct syntax_table *syntaxtab); /* syntax rules    */
#endif
```

APPENDIX E. OS/390 V2 Token types

```
*
*   Special token values
*
GOTORULE
CONTRULE
SYNTAXERR
LASTRULE
*
*   Normal token values
*
TOK_IS_DATA
TOK_IS_NUM
TOK_IS_EOS
*
TOK_IS_BLANK
TOK_IS_COMMA
TOK_IS_EQUAL
TOK_IS_LPAREN
TOK_IS_RPAREN
TOK_IS_LT
TOK_IS_GT
TOK_IS_LBRACE
TOK_IS_RBRACE
TOK_IS_DASH
TOK_IS_UBAR
TOK_IS_AND
TOK_IS_POUND
TOK_IS_AT
TOK_IS_PLUS
TOK_IS_SLASH
TOK_IS_PERCENT
TOK_IS_STAR
TOK_IS_SCOLON
TOK_IS_COLON
TOK_IS_SQUOTE
TOK_IS_DQUOTE
TOK_IS_NOT
TOK_IS_TILDE
TOK_IS_BAR
TOK_IS_QUEST
TOK_IS_PERIOD
TOK_IS_EXCLAM
TOK_IS_CENT
TOK_IS_BSLASH
TOK_IS_DOLLAR
TOK_IS_SBAR
TOK_IS_RVQUOTE
```