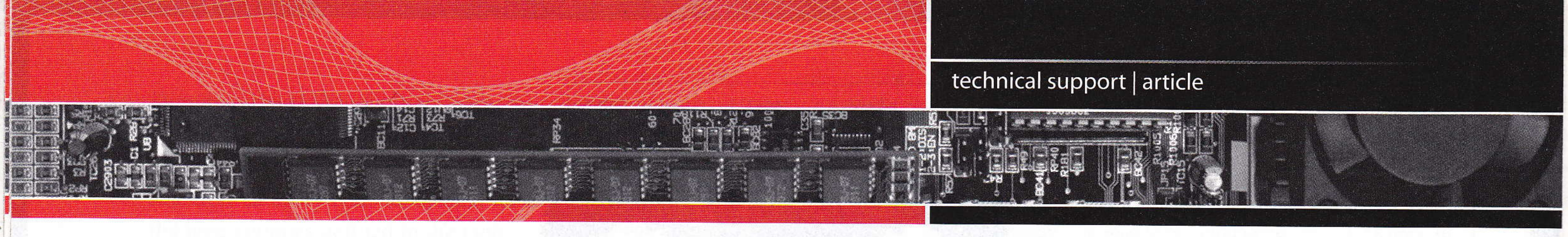


Parsing, Syntax Checking and Interpretation—Part Two

By Rick Tsujimoto

LAST MONTH I DISCUSSED HOW TO DO A LEXICAL SCAN OF A LINEAR expression, such as a command line, converting it into a series of tokens based on user-specified delimiters, keywords and strings (both numeric and alphanumeric). In addition, I also discussed how a formal parser has two major roles: lexical scanning and syntax checking, and why I chose to split these two processes into two separate callable functions. In this article, I will describe how to perform syntax checking of the linear expression after it has undergone the lexical scan phase.

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*

In the development of a compiler, these objectives seem like reasonable requirements and, as you can surmise, entail complex coding. In contrast, the approach I took with respect to syntax checking is much simpler: is the linear expression syntactically correct, or not? To determine if the syntax is correct, or not, one simply applies a grammar against the expression. Basically, the grammar is comprised of syntax rules that define the construct of valid expressions.

For example, the following is a common example of an assignment statement that is found in many computer languages:

A = B

The set of syntax rules for this expression might comprise of the following:

- A and B are names of variables.
- Variable names may consist of alphanumeric characters, the first of which must be a letter, and the names cannot be longer than n-characters.
- The statement must begin in a column other than 1.
- The equal sign must follow the first variable name.
- Spaces may be used anywhere in the expression, e.g. "A = B" is identical to "A =B".

Obviously, the more complicated the expression is, the more complicated the set of rules becomes. In a programming language, such as C for example, an assignment expression can become very complicated by parenthesizing sub-expressions, and joining them using a variety of operators, e.g. plus sign (+). The approach compiler writers take to address this issue is to store the syntax rules in a binary tree. This allows complex expressions to be checked by recursively traveling up and down the tree, applying the grammar against the expression.

```
  Token ID    Token Len.  Token Loc.
+-----------+-----------+------------+
|     0     |     4     |     1      |---------+
+-----------+-----------+------------+         |
|     3     |     1     |     5      |-------------+
+-----------+-----------+------------+         |   |
|     1     |     3     |     6      |--------------+
+-----------+-----------+------------+         |   ||
                                               V   VV
                                               ABCD=123
```

Figure 1: token entries on a FIFO queue

```
RULE1:   IF TOKENTYPE=TokenIsString and VALUE="ABCD" GOTO RULE3
RULE2:   SYNTAXERROR
RULE3:   IF TOKENTYPE=TokenIsEqual GOTO RULE5
RULE4:   SYNTAXERROR
RULE5:   IF TOKENTYPE=TokenIsNum and VALUE="123" GOTO RULE7
RULE6:   SYNTAXERROR
RULE7:   LASTRULE
```

Figure 2: set of theoretical syntax rules

```
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
         @RULE SYNTAXERR
FLUSH    @RULE TOK_IS_EOS,NEXT=DONE
         @RULE SYNTAXERR
DONE     @RULE LASTRULE
         @RULE TYPE=FINAL
```

Figure 3: OS/390 syntax table

```
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)(char token[], int tokenlen, char
msg[101]);
                      /* optional user exit */
};
```

Figure 4: Template for non-OS/390 syntax table

This method is necessary for the development of compilers, but it would have been overkill in the tool I developed.

LINEAR SYNTAX RULES

Instead of using binary trees for validating the linear expression, a simpler approach were taken, based on linear rules, similar to a decision table.

```
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"},
                     /* rule-06 */ {SYNTAXERR},
                     /* rule-07 */ {TokIsEOS,   GoTo9},
                     /* rule-08 */ {SYNTAXERR},
                     /* rule-09 */ {LASTRULE} };
```

Figure 5: Non-OS/390 syntax table

In order to illustrate the process of using a (syntax) table containing syntax rules for validating a linear expression, I think it might be helpful to reshow the results of the lexical scan that were discussed in last month's article (see Figure 1).

As you can see, a numeric identifier is associated with each token. In this example, the following associations are defined:

- 0 is assigned to character strings
- 3 is assigned to the equal sign (=)
- 1 is assigned to numeric strings

For illustration purposes, let us assign a name to each token identifier:

- TokenIsString is the same as 0
- TokenIsEqual is the same as 3
- TokenIsNum is the same as 1

Hence, a theoretical set of syntax rules that could be constructed validate ABCD=123 (see Figure 2).

Using the set of theoretical syntax rules, the syntax checking processing flow is as follows:

1. The current token is compared against a token type associated with character strings in the syntax table and the token's associated value is compared against the character string "ABCD".
2. If both matches apply, point to the next token in the queue and take the associated action, e.g. "GOTO" the specified syntax table entry, e.g. RULE3.
3. Otherwise, go to the next row in the table, e.g. RULE2, where the token type in the table is SYNTAXERROR; in this case, the lookup process ends, the return code is set to a value that indicates a syntax error and control is returned to the caller.

cole software®

Powerful, efficient development tools

800-XDC-5150
www.colesoft.com

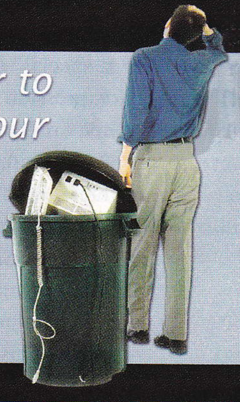
Using z/XDC, Assembler programmers can effectively find and fix bugs in applications running in nearly any z/OS environment.

But don't take our word for it.

"z/XDC is one of those software tools that makes you shudder to think of being without it - rather like someone taking away your keyboard and screen and making you use punched cards."

Rocket

Rob Scott,
Lead Developer, MXI
Rocket Software



z/XDC®

The essential Assembler programming tool

4. At RULE3, a test is made to see if the current token is an equal sign (=) and, if true, point to the next token in the queue and "GOTO" RULE5.
5. Otherwise, go to the next row in the table, e.g. RULE4, where the processing is the same as in Step 3 above.
6. At RULE5, a test is made to see if the current token is a numeric string and if it's associated value matches "123"; if true, "GOTO" RULE7.
7. Otherwise, go to the next row in the table, e.g. RULE6, where the processing is the same as in Step 3 above.
8. At RULE7, the token type in the syntax table is LASTRULE; a return code is set to indicate success and control is returned to the caller.

```
           .
           .
           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?
           BZ    CHK_SYNTAX_EXIT      NO, GET OUT
           .
           .
PARMLST1   @PARSE MF=L,CMDLINE=STRING,CMDLEN=L'STRING,DELIMTB=DELIMTAB
STRING     DS     CL80                INPUT STRING
PARMLST2   @SYNTXCK MF=L
           .
           .
```

Figure 6: Invoking syntax checker on OS/390

The use of a table containing syntax rules is easy to implement, but it is not practical in the case where an extensive grammar is required, e.g. a programming language. Yet, I believe this approach is more than adequate for the development of software that uses a limited grammar, e.g. a small set linear expressions, such as commands, or parameter data.

GRAMMAR

As mentioned earlier, a grammar is comprised of a set of syntax rules. In this case, the syntax rules are linear in nature, and are stored in a table.

The format of a syntax rule is as follows:

```
label condition <action> <string> <user 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. This is an optional parameter.
`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. This is a required parameter.
`<action>`	This is either a label or index value into the syntax table, which is "branched to" if the *condition* mentioned above is satisfied. On OS/390 it is a label and on non-OS/390 platforms it is an index value into an array (syntax table). The pointer to the current token is advanced to the next token before the "branch" is taken. This is an optional parameter.
`<string>`	The string value is used as a secondary comparison against the current token's associated value. The maximum string length is 100 characters. This is an optional parameter.
`<user exit>`	This is the address of a user routine that is called during the interpretation phase. This is an optional parameter.

It should be noted that the concept of user exits will be discussed in next month's article. In brief, user exits are used to provide context validation and processing.

On OS/390, the syntax table is generated via a macro called @RULE, which can be embedded as part of a program, or created as a separate CSECT. If the set of syntax rules becomes large, creating the syntax table as a separate CSECT makes it easier to manage and minimizes the impact on base register usage.

Using the tokens shown earlier for the expression ABCD=123 (see Figure 1), the OS/390 syntax table would contain the rules as shown in Figure 3.

The maximum number of rules that can be specified is limited to 1000 and the first array entry is must be the token type called STARTRULE.

The table must begin with @RULE TYPE=INITIAL and end with @RULE TYPE=FINAL macro. The token type value TOK_IS_EOS is a token value that is always the last token in the FIFO queue after a lexical scan. It represents the end of the list of tokens.

The coding requirements are the same as any OS/390 macro: labels must begin in column1, continuation mark must be in column 72, and so forth.

On non-OS/390 platforms, the syntax table is represented by an array of a data type known as a structure, and the template for the array is shown in Figure 4.

Again, using the tokens displayed in Figure 1 and the structure definition shown above, the syntax table would have the rules as shown in Figure 5.

The maximum number of rules that can be specified is limited to 1000 and the first array entry is must be the token type called `STARTRULE`. The limitation is artificially set due to the number of `#define` entries created for the purpose of equating a variable name to an array location. For example, GoTo5 is equated with the number 5, or the 6th array element (relative to zero). Hence, the GoTo #define values range from GoTo0 to GoTo999. If a larger array is required, the user could provide the additional #define statements.

PROGRAMMING EXAMPLES

The following examples show how to invoke the syntax checker for both OS/390 (see Figure 6) and non-OS/390 platforms (see Figure 7). In addition, the examples are based on the syntax tables discussed earlier, and the code fragments that invoke the parser are also included for readability's sake.

```
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);
     } /* end if */
          .
          .
```

Figure 7: Invoking syntax checker on non-OS/390 platforms

```
struct syntax_table syntaxtab[100] = { /* User syntax rules */
                    /* rule-00 */ {STARTRULE},
                    /* rule-01 */ {TokIsData,  GoTo06, "MYVAR"},
                    /* 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"},
                    /* rule-13 */ {TokIsData,  GoTo03},
                    /* rule-14 */ {SYNTAXERR} };
```

Figure 8: Statement continuation rules in non-OS/390 syntax table

```
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 9: Invoking SYNTBLGEN

ADDITIONAL PROGRAMMING NOTES

It was mentioned earlier that in addition to allowing free-form text, it is also possible to process linear statements that are specified on multiple inputs, by supporting statement continuation. How this is achieved will be discussed below.

In addition, an easy method for maintaining syntax tables on non-OS/390 platforms will be discussed as well.

Statement continuation

Linear expressions that span multiple inputs is often required when the statements have a complicated syntax that may, or may not, involve long data values. The user can choose any character/delimiter as the continuation character, such as a plus sign. A special token type value is provided (e.g. 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.

The user arbitrarily tests for the presence of the character chosen for the statement continuation character. 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, i.e. the status is stored, and the next syntax rule that is processed is the one immediately following CONTRULE. Hence, to provide a means for controlling the process flow, a special token value has been created which is called GOTORULE (see Figure 8). 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 associated with the keyword MYVAR:

MYVAR = aaaa (where aaaa is any string other than HELP)

or

MYVAR = HELP

If a plus sign (+) is used as a continuation character, there could be 2 places where it could be used:

Linear expressions that span multiple inputs is often required when the statements have a complicated syntax that may, or may not, involve long data values.

```
MYVAR +
      =
        +
        aaaa
        HELP
```

The degree of flexibility has a direct impact on the number of syntax rules required to support that flexibility. Since the syntax for specifying the rules on both OS/390 and non-OS/390 are nearly identical, an example of the syntax table for the non-OS/390 platform should suffice.

MAINTAINING SYNTAX TABLES FOR NON-OS/390 PLATFORMS

The main difference between syntax tables on OS/390 and non-OS/390 platforms is how they are created. Under OS/390, assembler macros are used, as opposed to an array. Macros allow the user to use labels, which are relocatable symbols that the binder resolves when it creates the executable. This makes it much easier for the programmer when the syntax table needs to be modified.

```
SYNTABLE @RULE TYPE=INITIAL
RULE010  @RULE TOK_IS_DATA,NEXT=RULE020,STRING=MYVAR,              +
               COMMENT="Start: process MYVAR"
         @RULE SYNTAXERR
RULE020  @RULE TOK_IS_EQUAL,NEXT=RULE030
         @RULE SYNTAXERR
RULE030  @RULE TOK_IS_NUM,NEXT=FLUSH,STRING=123
         @RULE SYNTAXERR,COMMENT="End:  process MYVAR"
FLUSH    @RULE TOK_IS_EOS,NEXT=DONE
         @RULE SYNTAXERR
DONE     @RULE LASTRULE
         @RULE TYPE=FINAL
```

Figure 10: Sample input file for SYNTBLGEN

```
struct syntax_table syntaxtab[MAX_RULES] = { /* Syntax rules */
 /* rule-0000 */ {STARTRULE},
 /* rule-0001 */ {TokIsData, GoTo3, "MYVAR"}, /* Start: process MYVAR */
 /* rule-0002 */ {SYNTAXERR},
 /* rule-0003 */ {TokIsEqual,GoTo5},
 /* rule-0004 */ {SYNTAXERR},
 /* rule-0005 */ {TokIsNum,  GoTo7, "123"},
 /* rule-0006 */ {SYNTAXERR},                 /* End:  process MYVAR */
 /* rule-0007 */ {TokIsEOS,  GoTo9},
 /* rule-0008 */ {SYNTAXERR},
 /* rule-0009 */ {LASTRULE},
}; /* end of syntax table */
```

Figure 11: Sample output file from SYNTBLGEN

The main difference between syntax tables on OS/390 and non-OS/390 platforms is how they are created.

In contrast, elements of an array are referenced via absolute numeric values. For example, the 11th array element (relative to zero) is referenced by specifying the number 10 in the array name, e.g. table[10]. Hence, any change to the array, such as inserting new rules, affects all other rules that reference the old 11th element, e.g. "GoTo11", as well other array elements that are displaced. This makes the task of maintaining syntax tables on non-OS/390 environments very cumbersome and, worse, prone to error.

To circumvent this obstacle, I created a C program called SYNTBLGEN (see Figure 9) that converts a source file into a file that contains the array statements, which has the user-specified syntax rules. The source file is comprised of statements, which are (almost) identical in structure to the macros used on OS/390 platforms. As a result, the programmer does not have to worry about maintaining the relationship between array element numbers and "GoTo" statements.

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.

The following examples show the macro statements (see Figure 10) that are used to define the same set of syntax rules, as in the prior examples, and the generated C arrays statements (see Figure 11).

NEXT MONTH

We have now shown how to parse a linear expression, by performing both a lexical scan and checking the syntax of the expression. The last task that needs to be discussed is how to extract the data from the linear expression, which is the ultimate objective of this process. This topic will be discussed in next month's issue.

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
Introducing the UNIX SYSTEM, Henry McGilton and Rachel Morgran, McGraw-Hill
Compiler Construction for Digital Computers, David Gries, John Wiley and Sons

Questions or comments? Please e-mail editor@NaSPA.com.

NaSPA member Richard Tsujimoto is an independant consultant specializing in MQSeries, CICS, and MVS.