Parsing, Syntax Checking and Interpreting—Part Three

By Rick Tsujimoto

THIS IS THE LAST OF A THREE PART ARTICLE, AND WE HAVE REACHED THE point where the linear expression has been sliced, diced, and weighed. What is left is handing over the pieces of data that the customer wants so that the real objective of the application can be performed. This process is referred to as interpretation.

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*

In this sense, interpretation is the execution of intermediate code that was generated from a high level language, after the translation phase. Some well-known languages that do interpretive processing are Basic, APL and Java.

For example, the following expression:

```
X = 3
```

would result in the variable X being assigned a value of 3 after interpretation.

This is where I part company with the formal definition of the term interpretation. The objectives of my tools are:

- To provide a simple means to syntactically check a linear expression.
- To hand over whatever part of that linear expression the user wants, as opposed to executing the linear expression itself.

Since no code is actually generated from the linear expression, it could be reasonably argued that the latter point (where data is handed over to the user) should have been called *extraction* instead. But, if one were to regard the syntax rules in the syntax table as a form of intermediate code, analogous to that produced by interpreters, I would argue that the processing of handing over data to the user is the direct result of interpretation.

```
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 1: OS/390 modified syntax table change

For example, if we take the same expression:

```
X = 3
```

the interpretation phase could result in presenting any, or all of the following values to the user program:

```
X
=
3
```

The values that are actually given to the user program are determined by the presence of the keyword `EXIT` in the OS/390 `@RULE` macro, or the specification of a user exit name in the C array of a structure called syntax_table.

USER EXITS

These are user-specified subroutines that are invoked during the interpretation phase. The exits can be specified on any syntax rule, except for:

OS/390

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

Non-OS/390

```
STARTRULE
```

Specifying OS/390 user exits

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, and is determined by the @RULE TYPE=INITIAL macro:

```
@RULE TYPE=INITIAL,EXITLOC=LOCAL
```

generates ACONs (this is the default), and

```
@RULE TYPE=INITIAL,EXITLOC=EXTERNAL
```

generates VCONs

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.

Invoking OS/390 user exits

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).

Specifying non-OS/390 user exits

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.

Invoking non-OS/390 user exits

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);
```

```
          .
          .
          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
            .
            .
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
            .
            .
PARMLST1 @PARSE MF=L,CMDLINE=STRING,CMDLEN=L'STRING,DELIMTB=DELIMTAB
PARMLST2 @SYNTXCK MF=L
PARMLST3 @INTRPRT MF=L,ERRMSG=IERRMSG
STRING    DS     CL80               INPUT STRING
IERRMSG   DS    CL100               ERROR MESSAGE BUFFER
            .
            .
```

Figure 2: Invoking interpreter on OS/390

When the interpreter encounters a user exit specification in a syntax rule, it invokes the user exit, passing string pointers to the token, 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).

PROGRAMMING EXAMPLES

The following examples show how to invoke the interpreter for both OS/390 and non-OS/390 platforms. In addition, the code fragments that invoke the parser and syntax checker are also included for readability's sake.

The syntax table for OS/390 (see Figure 1) shows that the user exit called MYEXIT is to be invoked if the evaluation of the syntax rule is true.

The syntax tables that are alluded to in the examples are based upon the expression ABCD=123, which had been used as the primary example in the first two articles.

The user exit in OS/390 will test if the string 123 had already been processed, or not (see Figure 2). If it has, then an error message indicating duplicate data has been encountered is returned to the interpreter. Assumedly, the user application would handle the error situation, even though the examples may not show any code to that effect.

The syntax table for non-OS/390 platforms (see Figure 3) shows that the user exit called MYEXIT is to be invoked if the evaluation of the syntax rule is true.

The user exit on non-OS/390 platforms will test if the string 123 had already been processed, or not (see Figure 4). If it has, then an error message indicating duplicate data has been encountered is returned to the interpreter. Assumedly, the user application would handle the error situation, even though the examples may not show any code to that effect.

```
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 3: Non-OS/390 modified syntax table change

```
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 4: Invoking interpreter on non-OS/390 platforms

CONCLUSION

Obviously, if the linear expression `ABCD=123` were the one and only statement that needed to be processed, then using these tools might be over-kill. Then again, if there is any chance that the expression may change, or if additional parameters or expressions may be needed in the future, it may pay in the long run to invest the time and effort to use the proper tools from the beginning.

Even though the examples used were based on linear expressions that mimicked command lines, the tools (parser, syntax checker and interpreter) could be used for other purposes as well, such as validating tag data, developing a meta-language, or even creating a poor man's calculator.

The complexity of the expressions will determine when using these tools becomes impractical, and when more sophisticated

methods are required, e.g. parse trees. But, from personal experience, I found that using these tools made it possible to process some fairly complex expressions, such as a variable length sub-parameter list, delimited by parentheses, as opposed to developing the code to pick apart those expressions.

For example, given the following expression, where free-form text is supported:

```
VOLSER = (V1, V2, …, Vn)
```

where `V1…Vn` are volume serial numbers. Would it be easier to code syntax rules and exit routines, or develop the code to process this expression?

In addition to the simplicity of this approach, the other benefits for using these tools are the consistent methods for processing linear expressions, which makes maintenance easier and, as a result, minimizes the possibility of developing error-prone code.

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.

Legislation Spotlight: NIAC continued from page 25.

direct, traceable impact is slim, especially considering the breadth of sectors that the NIAC represents. This raises the issue of how much value there is in having a single organization dedicated to so many sectors of the national infrastructure, especially when it is under the Department of Homeland Security, which is a young organization itself. The NIAC may benefit from cultivating relationships with older, more established areas of government such as the Department of Agriculture, the Department of Labor, and the Department of Defense. To find out more about NIAC, visit www.dhs.gov and type 'NIAC' into the search function.

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

Rachael Zimmermann is the editor for Technical Support magazine.