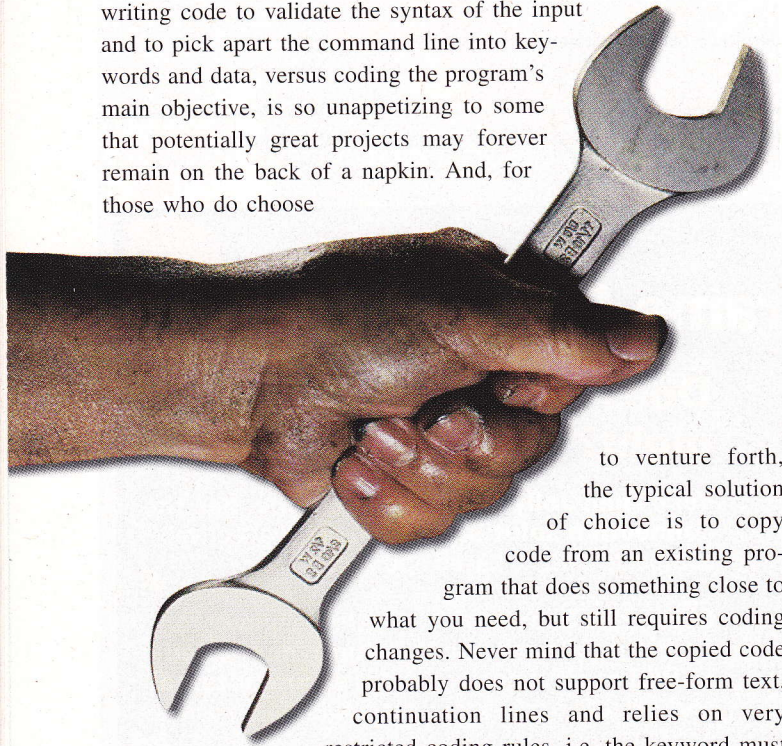


Parsing, Syntax Checking and Interpretation—Part One

By Richard Tsujimoto

AS ANYONE WHO HAS EVER DEVELOPED A TOOL, OR UTILITY PROGRAM, knows, one of the most frustrating parts of developing code is the processing of data that is used to drive the software, such as command lines, or parameters that are provided via switches, checking for correct syntax, extracting the keyword value(s) and acting on the values presented. This process is dry, repetitive and uninteresting which, as a result, often leads to code that is hard to maintain and buggy. Worse yet, the notion that more time may have to be spent on writing code to validate the syntax of the input and to pick apart the command line into keywords and data, versus coding the program's main objective, is so unappetizing to some that potentially great projects may forever remain on the back of a napkin. And, for those who do choose



to venture forth, the typical solution of choice is to copy code from an existing program that does something close to what you need, but still requires coding changes. Never mind that the copied code probably does not support free-form text, continuation lines and relies on very restricted coding rules, i.e. the keyword must begin in column 2, the equal sign must immediately follow the keyword, and the keyword value must follow right after the equal sign.

Basically what is required is a tool, or tools, to parse the input, check the syntax of the input and act on the keywords and/or data. These sorts of tools are normally used in the development of compilers and command interpreters and other software products, and would be found in commercial and/or academic software development environments. These tools are very complex and beyond the scope of this document. It should be noted that IBM does provide callable functions that are used in the construction of TSO command processors, such as IKJPARS, but they are not easy to use and are restricted to the

| Token ID | Token Len. | Token Loc. |
|----------|------------|------------|
| 0 | 4 | 1 |
| 3 | 1 | 5 |
| 1 | 3 | 6 |

ABCD=123

Figure 1: Token entries on a FIFO queue

OS/390 platform. But, despite the inherent complexity of these tools, it is still possible to leverage some of that technology to produce a reasonable solution.

Ideally, a simple mechanism is needed that deconstructs the data into subcomponents, e.g. keywords and values, provides the means to validate the syntax of the command line, and/or switch parameters, and to present the extracted data to the user program. This would offer a standard approach to handling this type of data, and it would minimize the creation of faulty code. In addition, the overall design and development phases would be simplified because this type of processing would take less time to implement.

Basically what is required is a tool, or tools, to parse the input, check the syntax of the input and act on the keywords and/or data.

I have developed software that addresses these needs and I believe it is easy to use as well. This simple mechanism consists of the following processes: parsing, syntax checking and interpretation. I would like to discuss each process as part of a three-part article.

WHEN IT ALL BEGAN

As a neophyte, I looked forward to being able to ply the skills I learned as a computer science major when I entered the working world in the summer of 1973. I worked for a major oil company and soon discovered that much of what I learned would be of little, or no value. Over time though, whenever the opportunity presented itself, I did try to incorporate what I learned in the code I produced.

My second job was at a major insurance company, where I became a CICS systems programmer and became exposed to the online world. It was here that I first encountered a group of technicians that created tools, applied custom patches to system software and dove head first into dumps. One of my coworkers wrote a macro that was used to extract the parameters of a CICS transaction. It was very basic and inflexible, but it was the first programming example that piqued my interest. Needless to say, I copied it and began to dissect it and make changes. I do not recall the original name of the macro, but I called my version PARSE.

Over the years, I would work on PARSE making additional changes, tweaking it here and there when I found spare time, or if I found a work-related requirement that PARSE could potentially satisfy, I would incorporate those changes as well. But, in the end, I soon realized that PARSE became too cumbersome to use and was soon relegated to the tool chest.

After 11 years and several jobs later, I had a consulting assignment at Bellcore, the research arm of the baby bell companies after the breakup of AT&T. Unlike my prior jobs and assignments, this was my first experience working in a high-tech environment. One of the team members created a prototype for a service provisioning

```
DELIMTAB @DELIM MF=GEN,
PTPOUND=NO,
PTDQUOTE=TOGGLE
```

Figure 2: Delimiter list

system for 800 services that was based on rules; in effect, the rules were the program. This was a very interesting approach to a problem.

After my stint at Bellcore, I gradually took what I learned from there and began to rethink how to change PARSE, and started to incorporate those ideas in a new version of the software. After a number of years, I created a new version of the software in OS/390 assembler, and in the C language for Windows/2000, AIX, HP-UX and AS/400. What originally started out as a single assembler macro evolved into 3 programs and a number of macros.

WHAT IS PARSING?

One 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

From the software perspective, a parser converts the linear expression, e.g. command line, into tokens, a process that is known as lexical scanning, and checks that the linear expression is syntactically correct based on a given grammar.

I chose to separate parsing into two distinct callable functions. The first, which I refer to as the parser, is nothing more than a lexical scanner. The second function is what I refer to as syntax checking. I took this approach because I thought lexical scanning could be useful on its own. For example, you could take input, convert it into tokens, and count the number of tokens, similar to the Unix wc function.

TAKE CONTROL OF YOUR APPLICATION PORTFOLIO



Use the Edge Portfolio Analyzer To:

- ◆ Plan Your LE Migration
- ◆ Guide Implementation of New Compilers
- ◆ Troubleshoot and Debug Programs
- ◆ Populate Change Management Database
- ◆ Identify Language Impediments to CICS Upgrades

What's
in Your
Production
Library?

Available NOW!

Version 1 Release 6.02 Supports:

- Operating Systems through z/OS 1.7
- IBM Enterprise COBOL through V3.4
- IBM Enterprise PL/I through V3.5
- And much, much more....

Get Your FREE
General Information Manual
GO TO www.edge-information.com
or CALL 847-297-2020

TOKENIZING DATA

Lexical scanning takes the input and breaks it up into tokens, or atomic expressions, which are then stored in a data structure, such as a push down stack. The atomic expressions could be a string of characters, numbers or delimiters.

For example, in the sentence "The child is 8 years old.", the tokens would be: *The, child, is, 8, years, old,* and *.* (*period*). Typically, the blanks are discarded and not converted into tokens.

The approach I took is slightly different. First, unlike a parser for a compiler which works with a predefined set of delimiters, I chose to give the user the option of providing a user-specified list of punctuation marks, which is used to distinguish what is, or is not to be regarded as a delimiter. Hence, if a user wants to accept the pound sign (#) as a valid character in a name, that particular punctuation mark would be flagged as such in the list of delimiters passed to the parser.

Second, the tokens are stored in a first-in, first-out (FIFO) queue, as opposed to a push down stack. A push down stack is needed for a compiler (or interpreter) when it performs expression analysis, which is beyond the requirements of my parser.

Last, the user can specify when a blank is to be converted into a token, and when it can be ignored. This is achieved by letting the user specify in the list of delimiters passed to the parser, when to start/stop tokenizing blanks. For example, the user can specify that when a double quote is encountered, blanks will be tokenized, and when another double quote is encountered, blanks will not be tokenized.

As lexical scanning takes place, an entry is created in a FIFO queue for each token (see Figure 1), where:

- ▼ **Token identifier** 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** is the length of the token.
- ▼ **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.

MODIFYING THE LIST OF DELIMITERS

The user has the choice of either using the default list of delimiters, or the user could

```
char userdelim[33][2] = { /*User-defined delimiters for parse */
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, /* could be used in file names */
PTAT,YES,
PTPLUS,YES,
PTSLASH,YES,
PTPERCENT,YES,
PTSTAR,YES,
PTSCOLON,YES,
PTCOLON,YES,
PTQUOTE,YES,
PTDQUOTE,TOGGLE, /*make sure blanks between double quotes are tokenized */
PTTILDE,YES,
PTBAR,YES,
PTQUEST,YES,
PTPERIOD,YES,
PTEXCLAM,YES,
PTBSLASH,YES,
PTDOLLAR,YES,
PTRVQUOT,YES,
PTLBRACKET,YES,
PTRBRACKET,YES,
PTNULL):
```

Figure 3: Non-OS/390 delimiter list example

```
@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 4: Invoking parser on OS/390

```
#include <parse.h>

main() {
char inbuff[MAXSTRINGLEN+1];
char userdelim[33][2] = { /* user-defined delimiters for parse */
PTPAD,YES,
.
PTNULL);

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 5: Invoking parser on non-OS/390 platforms

provide a customized list of delimiters. The default list of delimiters for a given platform treats all punctuation marks as delimiters.

The following examples show a delimiter list that treats the pound sign (#) as a character and not a delimiter. In addition, when a double quote is first encountered, it will cause blanks to be tokenized and when it is encountered again, the tokenization of blanks will cease. In effect, the double quote acts as a toggle switch for tokenizing blanks. It should be noted, any delimiter could be used to toggle the tokenization of blanks.

OS/390 delimiter list example

In this example, an assembler macro called @DELIM is coded, and the values for the pound sign (#) and double quote are modified (see Figure 2).

Non-OS/390 delimiter list example

In this example, a little more work is required because an array declaration must be modified, as opposed to a macro (see Figure 3).

The user must:

1. Copy the file dfltdelim.h into the program.
2. Change the array name from dfltdelim to userdelim. (or some other name of the user's choosing)
3. Modify the settings for the pound sign (#) and double quote.

PROGRAMMING EXAMPLES

The following examples show how to invoke the parser for both OS/390 and non-OS/390 platforms.

Invoking parser on OS/390

On OS/390 the parser is invoked using the @PARSE macro (see Figure 4).

Invoking parser on non-OS/390 platforms

On non-OS/390 platforms, the parser is invoked by calling the function called parse (see Figure 5).

NEXT MONTH

At this point, I have shown how the data gets parsed and stored in a FIFO queue. The data is now in a state that makes it easy to check if it is syntactically correct or not. Next month I will discuss how to setup the syntax rules and invoke the syntax checker.

REFERENCE MATERIAL

ESA/390 Principles of Operation, SA22-7201
HLASM VIR4 Language Reference, SC26-4940
HLASM VIR4 Programmer's Guide, SC26-4941

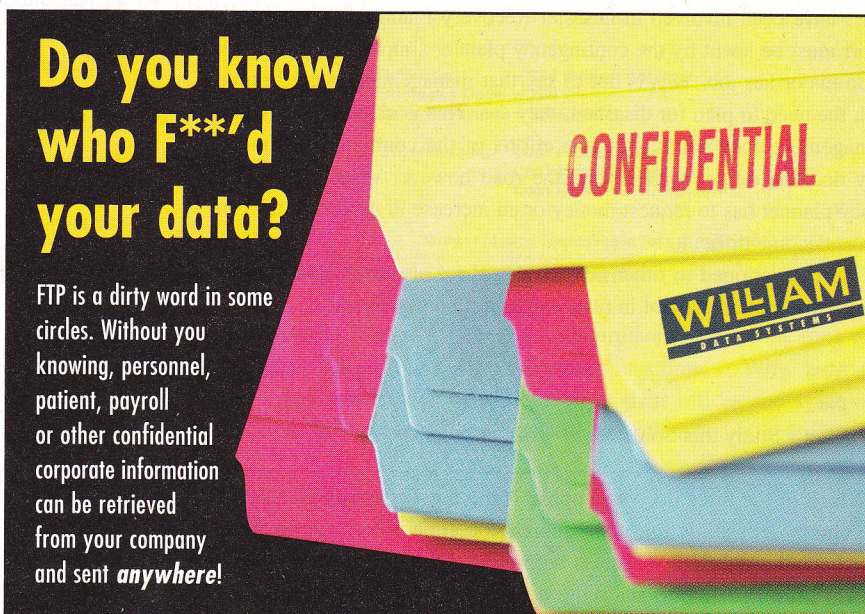
ILE C for AS/400 Programmer's Guide, SC09-2712

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 independent consultant specializing in M0Series, CICS, and MVS.



Do you know who F'd your data?**


FTP is a dirty word in some circles. Without you knowing, personnel, patient, payroll or other confidential corporate information can be retrieved from your company and sent *anywhere!*

This can all be achieved with the universal File Transfer Protocol (FTP), often used, easily abused but rarely managed.

FREE FTP ACTIVITY CHECKER:
Download your free copy of the z/OS FTP activity checker from www.willdata.com/ts.htm

ftpalert from William Data Systems lets you control who can ftp files to and from your zSeries eServer. *ftpalert* installs in minutes, secures all FTP activity and, to satisfy regulatory compliance, incorporates comprehensive, real time audit and logging facilities for all FTP usage.

Evaluate *ftpalert* today. You have nothing to lose but your data!

ftpalert 

Call (877) 723-0008 or (703) 674-2200
or visit www.willdata.com/ts.htm

IBM and ServerProven are trademarks of International Business Machines Corporation in the United States, other countries, or both.