TO: MSPM Distribution
FROM: Karolyn Martin
DATE: November 6, 1968
SUBJECT: Multics Command Language - BX.1.00

A number of minor changes to the Multics Command Language have been made and the writeup has been completely rewritten to present the ideas more clearly. These changes simplify implementation or incorporate wanted features.

1) Arguments to commands are passed as non-varying character strings, rather than varying character strings.

2) Arrays of varying character strings (formerly called lists) are no longer valid arguments to commands.

3) The delimiters for various syntax elements have changed. The summary of the syntax should be studied closely (page 6).

4) The concept of iteration has been added.

Published: 10/29/68
(Supersedes: BX.1.00, 09/15/66;
BX.1.00, 05/27/66)

Identification

Multics Command Language
W. H. Southworth, G. Schroeder, R. Sobecki, D. Eastwood

Purpose

The Multics Command Language provides the user with a concise means of expressing his wishes to the Multics system. The language and implementation are based on the work and suggestions of L. Peter Deutsch, E. L. Glaser, R. M. Graham, C. N. Mooers, J. H. Saltzer, and C. Strachey.

Introduction

Most time-sharing systems provide the user with various services which may be invoked from his console by means of "commands" to the operating system. A small number of time-sharing systems go beyond this and allow any user to define (with varying degrees of ease) his own commands, which may be used in exactly the same way as the system supplied commands. Nearly all command programs, whether user defined or system provided, require additional information, which must be supplied by the user, before they can complete their function. Some systems require that every command program directly interrogate the user for the additional information it needs. A more general method is to accept "arguments", in addition to the command name, at the time the command is issued by the user. These arguments are then passed on to the command program by the system. This approach permits great flexibility in the design of command programs. Information may be supplied by arguments, interrogation or a combinaton of both.

Issuing a command is analogous to executing a function or subroutine call in a language such as PL/1 or FORTRAN. With this view, the name of the command is simply the name of a command program to be either interpreted, if it is a user defined macro (see BX.1.01 for a description of the Macro facility), or executed if it is the name of an entry point in an executable segment which conforms to Multics standards (as defined in BD.7.02). The arguments are either used in the expansion of the macro or passed to the executed procedure in the standard manner. The link between the issuance of the command by a user and the calling of the command program is the command language

interpreter. It performs many of the functions of a compiler, principally, parsing the command (which is initially a character string) into its basic elements (e.g., command name and arguments) and formatting the arguments for use by the command program. Finally, the command language interpreter calls the macro expander if the command name is the name of a macro or calls the command program directly.

The Command

A command is a sequence of zero or more elements. The first element is interpreted as the name of the command program and any additional elements are arguments. The normal command program will expect input arguments which are fixed length character strings, and return a value which is a varying character string. If the command program does not expect arguments in this form the command language interpreter, the Shell, will convert the type of the arguments according to information contained in the symbol table for the command program. The elements of a command are separated by spaces and terminated by a semicolon or a new line character. For example, to change the name of a file from "a" to "b", using the change_name command, a user would type

(1) change_name a b

Elements

The simplest type of element is a string of characters not containing any spaces or other characters reserved by the command language. The semicolon and new line characters are reserved. Other reserved characters will be identified as they are encountered. The three elements in example 1, at the end of the previous section, are the simple character strings "change_name", "a", and "b".

Any element (or part of an element) may be a command. The user can tell the interpreter to evaluate an element as a command by surrounding the element with brackets (which are reserved characters). For example in

(2) change_name [oldestfile] b

the second element is a command. When an element is evaluated as a command, the result of that evaluation (i.e., the returned value of the function) replaces the original element. Suppose that the command "oldestfile" returns

as its value the name of the oldest file in the users' directory, then example 2 changes the name of the oldest file to "b". In this case, the first argument to the command program change_name is the character string returned by the command program "oldestfile" and the second argument is the character string "b".

Note that the spaces before and after the brackets are necessary to indicate that the result of "oldestfile" is an element and not a portion of an element. Suppose that the user had a program "me" which returned as its value his default working directory, (e.g., "me" would return a character string of the form ">user_dir_dir>Southworth.MAC"). While working in some other directory the user might link to a file in his default directory by typing:

(3) link [me]>test.epl

This command consists of two elements, the character string "link" and the result of the command program "me" concatenated with the character string ">test.epl". Similarly a command of the form

(4) link [me]>[filename 17]

would consist of only two elements, where the second element is formed using the values returned by the two command programs "me" and "filename". Note that the command "filename" has one argument, the character string "17".

Sometimes it is necessary to use a reserved character without its special meaning. For example, a command name might contain an imbedded space. The characters quote and left and right accent are reserved for this purpose. Reserved characters within any string of characters surrounded by quotes or accents, will be treated as ordinary characters. For example,

(5)
```
`change name' a b
change" "name a b
"change name" a b
change` 'name a b
```

are all acceptable methods for executing a command whose name contains an imbedded space. Also, since quotes and accents are reserved characters it may be desirable to suppress the special meaning of one or the other. This may be done by surrounding quotes with accents, and accents with quotes. For example, the operator could issue the command

```
broadcast "Don't do anything!"
```

Active, Neutral, and Empty Commands

There are three types of commands which may appear in elements; active, neutral and empty commands. In order to understand how these three types differ it is necessary to have a basic knowledge of the scanning and interpretation algorithm of the shell. A command line is scanned from left to right. The shell maintains a pointer which indicates its current position in the line. Whenever a command has been completely scanned it is evaluated (i.e., the command program is executed). For example in

(6) change_name a b ;
↑

when the pointer has reached the indicated location the shell will recognize that the end of a command has been reached and call the command program "change_name". In the case of an element which is a command, nothing to the right of the element will be scanned until after the command element has been executed and its value has been inserted back into the command line. For example, in

(7) change_name [oldestfile] b;
↑

when the pointer has reached this point the command program "oldestfile" will be called. If the returned value is "xyz", the transformed command line is

(8) change_name xyz b;
↑

Note that the pointer is set to the beginning of the inserted value. This is important because the returned value will now be scanned in the same manner as the original command element. If "oldestfile" had returned the string "[myoldest]", then the scanning pointer would have encountered this string as a command because of the brackets and executed the command "myoldest".

The type of command in the previous examples, which returns a value which is rescanned we will define as an "active command". The command language recognizes two other kinds, a "neutral command" and an "empty command". A command preceded by "|[" rather than "[" is said to be "neutral". Its value is not rescanned. This is particularly useful in defining certain macros. If our previous example with "oldestfile" had read

(9) change_name |[oldestfile] b;

and "oldestfile" had returned the string "[myoldest]", then this value would have been inserted into the command line and the scan pointer set to the next character after

the inserted character string, i.e.,

(10) change_name [myoldest] b;
↑

In this case the inserted string would not be recognized as a command and "change_name" would be called with "[myoldest]" as its first argument. An "empty" command is preceded by "|||[". After an empty command is executed its value is thrown away.

The vertical bar is a reserved character only in the context of "|[" or "|||[". An easy way to remember the three types of commands is to think of a command as performing the three actions: evaluation, insertion, and reevaluation. A single vertical bar suppresses reevaluation leaving evaluation and insertion, while a double vertical bar suppresses reevaluation and insertion leaving only the first evaluation.

Iteration

Sometimes the user wishes to repeat a command with one or more elements changed. The iteration facility of the command language is provided for economy of typing in this case. The "iteration set" is a set of zero or more elements enclosed by parentheses (parentheses are reserved characters). If it contains no elements it is ignored. Otherwise, each element of the set will, in turn, replace the entire set in the command line. For example

(11) print (a b c).epl

is equivalent to the three commands

```
print a.epl; print b.epl; print c.epl
```

More than one iteration set may appear in a command. All possible combinations will be executed. For instance, the compound command

```
(print delete) xyz(.epl .eplbsa);
```

would expend into the commands:

```
print  xyz.epl
print  xyz.eplbsa
delete xyz.epl
delete xyz.eplbsa
```

Nested iteration sets behave in exactly the same manner as unnested sets. Evaluation of parentheses occurs from the outside in. The principle use of nested iteration sets is to reduce typing when subsets of an element are repeated. For example

(12)
```
make_directory >user_dir_dir>((Southworth Martin).MAC Stone.GE)
```

would make three directories

```
>user_dir_dir>Southworth.MAC
>user_dir_dir>Martin.MAC
>user_dir_dir>Stone.GE
```

Summary of Command Language Syntax

The Multics Command Language contains the following syntactical elements.

command line	the character string representation of a command or sequence of commands.
command	a sequence of zero or more elements separated by spaces (in the command line). The first element is taken as the command name and additional elements as arguments.
iteration set	a sequence of zero or more elements, enclosed by parentheses, which are inserted in turn in the command for evaluation.
command program	either a defined macro to be recognized and expanded by a macro expander program, or executable machine instructions whose name represents an entry point in a segment which conforms to multics standards (as defined in BD.7.02).
element	the basic component of a command; it may represent a command name, or argument.
active command	a sequence of zero or more elements surrounded by brackets (it is not necessary to enclose the character string typed in at the user's console with brackets, in this case brackets are assumed). The character string within the brackets is treated as a command - it is evaluated, its value is inserted into the command line and its value is rescanned as part of the command line.

neutral command	a sequence of zero or more elements surrounded by "|[" on the left and "]" on the right which is evaluated, as a command line, but is not rescanned.
empty command	a sequence of zero or more elements surrounded by "||[" on the left and "]" on the right which is evaluated. Its value is thrown away.
literal string	a character string surrounded by quotes or balanced left and right accents. Its value should be taken literally, i.e., reserved characters within the string should not be recognized for their special meaning.
semicolon	denotes the end of a command and the beginning of another command of the same type.
new line	new line characters within the command string passed to the command language interpreter are ignored if encountered in the scan of the command line. This should not be confused with the fact that the new line character may serve as a delimiter for whatever program called the interpreter.

Implementation

The command language interpreter, the Shell, is normally driven by the Listener. The shell provides the necessary parsing to process a character string as a command. The Listener can be conceptually described as

```
[[read_line]]; listener
```

Its function is to listen for requests in the form of command lines typed in at the user console. In the above command language description, the listener reads in a line from the console, evaluates the line as a command, and re-calls itself to repeat the function. In actuality this is usually accomplished by a multics procedure which calls the shell which accepts as its single argument a character string (fixed length or varying) to be evaluated as a command.

Formal Description of the Multics Command Language

The following Backus-Naur description formally defines the syntactic components of the Command Language. For simplicity we have not provided definitions for the ascii characters, based on the assumption that the alphabet is not open to design changes. If a construct is enclosed in parentheses it is to be interpreted as zero or more occurrences of that construct.

<command sequence> ::= <command> (<semicolon> <command>)

<command> ::= <element list>

<element list> ::= <element> (<spaces> <element list>)

<element> ::= <element component> (<element>)

<element component> ::= <function> | <iteration set> |
<literal string> | <unreserved character>

<function> ::= <active function> | <neutral function> |
<empty function>

<active function> ::= <left bracket> <command> <right bracket>

<neutral function> ::= <vertical bar> <left bracket>
<command> <right bracket>

<empty function> ::= <vertical bar> <vertical bar> <left
bracket> <command> <right bracket>

<iteration set> ::= <left paren> <element list> <right paren>

<literal string> ::= <quoted string> | <balanced quoted string>

<quoted string> ::= <quote> <balanced quoted string> <quote> |
<quote> <unquoted character string> <quote>

<balanced quoted string> ::= <left accent> <balanced quoted
substring> <right accent>

<balanced quoted substring> ::= <character string not containing
` or '> | <balanced quoted
string>