To: Distribution

From: Robert A. Freiburghouse

Subject: A Unified Command Language

Date: February 6, 1974

This document defines a command language and command processor that is intended to be a user selected alternative to the current Multics command processor. The language is suitable for use as an interactive or absentee job control language, and it also is a suitable language in which to perform simple calculations.

Design Objectives

1. Provide a single unified language containing the essential functions of calc, abbrev, exec_com and the current Multics command language.

2. Provide a command language that can call subroutines and functions written in standard languages in a natural manner passing arguments and receiving values having any of the scalar data types of the standard languages. Any procedure whose arguments and return values are scalars can be invoked from the command processor exactly as it would be invoked from another procedure, thus eliminating the need for active functions and commands to be written in a nonstandard style.

3. Provide a language whose implementation will perform a given operation using less CPU time and storage than used by the existing command processor and related facilities to perform the equivalent operation.

General Concepts

The command language is a very simple algorithmic language whose largest syntatic unit is a <command>. Each <command> is a conditional or unconditional imperative statement which can contain references to named variables, expressions, and other <command>s. Expressions are the familiar parenthesized infix and prefix expressions of Fortran or PL/I.

The command processor is an interpreter that executes a sequence of <command>s. Interpretation of each <command> is performed as a two stage process. During the first stage, the <command> is processed as a sequence of characters without regard to its syntatic construction or purpose as a command. It is during this first stage that abbreviations and parameters are replaced as described later. During the second stage of interpretation, each <command> is parsed (identified) and executed.

The command processor can be called by a <command>. Each invocation creates a new set of arguments, a new set of local variables, a new command input file, and a new "current" abbreviation file.

```
cp path s1 s2...sn
          or
cp path
          or
cp
```

where path is the pathname of the command input file, and s1 s2...sn are the strings which are arguments of the new invocation of the command processor. If path is omitted, commands are read from user_input.

Parameter Substitution and Abbreviation Replacement

Before each <command> is executed, the following steps are performed:

1. the <command> is isolated from the rest of the command file by scanning to the first ; or <newline> not contained in a <quoted string>.

2. If the current abbreviation file designator is not null, the <command> is examined as a sequence of characters. Each <pattern> is replaced by the replacement string defined for that <pattern> in the abbreviation file. After replacement of a given <pattern>, the replaced text is scanned. Scanning and replacement occurs from left-to-right.

3. Each &k, where k is an <integer>, is replaced by the kth argument to this invocation of the command processor. If no such argument exists, the &k is removed.

4. If the <command> does not begin with ., it is translated as an <execute>.

Note that by using abbreviations the user can eliminate the . required on each command and can change the syntax of commands to a limited extent.

Note that this proposal does not include the string iteration currently performed by the Multics command processor. If that type of string processing is to be done, it should be done as a part of step 4 above.

The Syntax and Semantics of Commands

<command line>::=<command>[;<command>]...<newline>

<command>::=<define>|<delete>|<use>|<list>|<while>|<if>|
<let>|<call>|<exit>|<print>|<on>|<for>|<execute>

<define>::=.define <pattern>::=<expression>

<pattern>::=<expression>

causes the value of <pattern> to be entered as an abbreviation in the current abbreviation file. Both the <pattern> and the <expression> must yield string values.

<delete>::=.delete <pattern>

causes the value of <pattern> to be deleted from the current abbreviation file. The <pattern> must yield a string value that is defined in the current abbreviation file.

<use>::=.use[<expression>]

causes the file identified by the value of <expression> to become the current abbreviation file. If the <expression> is omitted or yields a null string as its value, the current abbreviation file designation is set null - inhibiting the replacement of abbreviations.

<list>::=.list

causes the content of the current abbreviation file to be listed on user_output.

<while>::=.while(<expression>)<group>

<group>::=<command>|(<command>[;<command>]...)

If <expression> is true, the <group> is evaluated; otherwise, it is not. Upon completion of the <group>, the <while> is repeated. The <expression> must yield a logical value.

<if>::=.if(<expression>)<group>

If <expression> is true, the <group> is evaluated; otherwise, it is not. The <expression> must yield a logical value.

<let>::=let <name> be <expression>

causes <name> to be defined as a local variable allocated in the current stack frame. The value of the variable is the value produced by evaluation of the <expression>.

<call>::=
.call <expression>([<expression>[,<expression>]...])

If the first <expression> is a <name> that has not been defined as a local variable, it is translated into "<name>", and evaluated as a string; otherwise, evaluation of the first <expression> must yield a string. In both cases, the string must be a pathname that identifies an object segment entry point.

The argument <expression>s are evaluated and converted to conform to the data types specified by the entry definition of the object segment as described later.

<exit>::=.exit

causes control to return from the current invocation of the command processor.

<print>::=print <expression>[,<expression>]...

causes the value of each <expression> to be written on user_output in a suitable format.

<on>::=.on <expression> do <group>

causes the <group> to be established as an on-unit for the condition identified by the string value of the <expression>. The <expression> must yield a string value.

<for>::=.for <name>=<expression>[,<expression>]...do<group>

Let n be the number of <expression>s. For k=1,2,...n, the kth <expression> is evaluated and its resulting value assigned to the local variable <name>, and the <group> is evaluated. A <for> defines its <name> as a local variable just like a <let>.

<execute>::=<string> [<string>]...

An <execute> is translated into

.call "<string>"(["<string>"[,"<string>"]...])

except

1. If the original <string> is a <quoted string>, no additional quotes are used in the translation.
2. If the original <string> has the form .<name>, no quotes are used in the translation and the translation is <name>.

The <execute> is a special form of the <call> that is designed to be easy to type and compatible with the existing command language. It is translated into a call as the last step of the string processing stage of interpretation.

<name>::=<letter>[<letter>|<digit>|_]...

<string>::=<quoted string>|<unquoted string>

<quoted string>::="<char>..."

<char>::=""|Any ASCII character except "

<unquoted string>::=<notend>...

<notend>::= Any ASCII character except ; <newline> blank or tab

<newline>::=ASCII new line 012

<expression>::=<infix>|<prefix>|<basic>

<infix>::=<expression><infix-op><expression>

<infix-op>::=+|-|*|/|**|=|^=|>|>=|<=|<|>|&|||

<prefix>::={+|-|^}<expression>

<basic>::=(<expresison>)|<name>|<constant>|<function>

<constant>::=<quoted string>|<integer>
|<real>|true|false|null

<integer>::=<digit>...

<real>::={<integer>.[<integer>]|.<integer>}[e[+|-]<integer>]

<function>::=<expression>([<expression>[,<expression>]...])

A function works like a call, except that a return value is expected and is converted to the corresponding command language data type.

Variables and Data Types

A local variable is allocated in the stack frame of the command processor. Each variable is capable of possessing values of any data type.

The possible data types are:

integer	(fixed bin(35))
real	(float dec(18))
logical	(bit(1))
string	(char(256) varying)
address	(pointer, pointer)

These data types are designed to accommodate all PL/I and Fortran data types except complex numbers. The conversions between these types and PL/I types are given in the following section.

A variable is defined by the appearance of its <name> in a <let> or <for>. Because the command language has no concept of multiple scopes of names and no declared attributes, no declarative statements are required. The type of a variable is the type of the value it currently possesses.

Argument Conversion

If an entry definition specifies no parameters, the arguments, if any, are passed without conversion.

If the entry definition specifies a single one-dimensional array, the arguments are converted to the data type of the array and each argument is transformed into an element of the array. The lower bound of the array descriptor is set to 1 and the upper bound is set to n, where n is the number of arguments given. Using this scheme, a PL/I procedure can easily receive a variable number of arguments while remaining within the standard language.

If the entry definition specifies one or more scalar arguments, each argument is converted to the data type of its corresponding parameter. If an argument is a reference to a local variable, it is passed by-reference; otherwise, it is

passed by-value. When an argument is passed by-reference, it is converted to conform to the data type of the corresponding parameter, and upon return it is converted back to the original type of the argument.

If the expected data type of a called procedure is any kind of PL/I arithmetic data, both integer and real can be converted to the expected type. On return, all PL/I arithmetic types, except complex, can be converted either to integer or real. Large decimal values are rounded and a warning produced.

Aggregate values cannot be passed or received.

PL/I bit strings, other than bit(1), are converted to character strings.

Excessively long (>256) character strings are truncated with a warning.

Because the command language stores addresses as pointer pairs, it can hold pointer, offset, label, entry, format, file, and area values as address values.