L10 PROGRAMMING GUIDE
(User Guide)

Augmentation Research Center
Stanford Research Institute
Menlo Park, California 94025

For Systems Programmer's Guide, see 705-2,
## CONTENTS

<table>
<thead>
<tr>
<th>Title</th>
<th>Statement #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 1. INTRODUCTION TO L10</td>
<td>(3)</td>
</tr>
<tr>
<td>Introduction</td>
<td>(3a)</td>
</tr>
<tr>
<td>CONVENTIONS USED IN DESCRIPTION OF L10</td>
<td>(3b)</td>
</tr>
<tr>
<td>DEFINITIONS</td>
<td>(3c)</td>
</tr>
<tr>
<td>Section 2. PROGRAM STRUCTURE AND PROCEDURES</td>
<td>(4)</td>
</tr>
<tr>
<td>Introduction</td>
<td>(4a)</td>
</tr>
<tr>
<td>USER PROGRAM STRUCTURE</td>
<td>(4b)</td>
</tr>
<tr>
<td>Section 3. ELEMENTS OF L10</td>
<td>(5)</td>
</tr>
<tr>
<td>Introduction</td>
<td>(5a)</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>(5b)</td>
</tr>
<tr>
<td>OPERATORS</td>
<td>(5c)</td>
</tr>
<tr>
<td>PRIMITIVES</td>
<td>(5d)</td>
</tr>
<tr>
<td>EXPRESSIONS</td>
<td>(5e)</td>
</tr>
<tr>
<td>Section 4. DECLARATIONS</td>
<td>(6)</td>
</tr>
<tr>
<td>Introduction</td>
<td>(6a)</td>
</tr>
<tr>
<td>GLOBAL DECLARATIONS</td>
<td>(6b)</td>
</tr>
<tr>
<td>REFERENCE DECLARATIONS</td>
<td>(6c)</td>
</tr>
<tr>
<td>LOCAL DECLARATIONS</td>
<td>(6d)</td>
</tr>
</tbody>
</table>
Section 5. STATEMENTS
ASSIGNMENT
DIVIDE
BLOCK
CONDITIONAL
ITERATIVE
TRANSFER
NULL STATEMENT

Section 6. STRING TEST AND MANIPULATION
Introduction
CURRENT CHARACTER POSITION (CCPOS) AND TEXT POINTERS
PATTERNS - the FIND statement and CONTENT ANALYSIS patterns
STRING CONSTRUCTION

Section 7. CONTENT ANALYSIS AND SEQUENCE GENERATOR PROGRAMS
Introduction
THE CREATION OF USER WRITTEN PROGRAMS
THE CONTEXT OF USER WRITTEN PROGRAMS
Section 8. INVOCATION OF USER FILTERS AND PROGRAMS

Introduction

SIMPLE CONTENT ANALYSIS PATTERNS

PROGRAMS SUBSYSTEM

INDEX

(10)

(10a)

(10b)

(10c)

(11)
Section 1. INTRODUCTION TO L10

Introduction

This document describes a subset of the L10 programming language used at ARC on the PDP10. The language contains some high level features for operations such as string analysis and manipulation which are implemented in the language as calls on library routines. In addition, L10 has basic constructions such as local variables which have been particularly useful. The L10 compiler was written using the compiler-compiler system Tree Meta.

The subset presented is offered primarily to satisfy the needs of the novice programmer interested in producing user programs for use in the analyzer formatter system of the NLS portrayal generator.

The portrayal generator, its NLS relative the sequence generator, and the NLS commands used to compile users' programs and establish them as the filters used by the system are described in Section 7 and 8 below.
CONVENTIONS USED IN DESCRIPTION OF L10

The following conventions (syntax) are used in the description of the features of L10.

If there is more than one alternative allowed in any syntax rule, they are separated by slashes (/).

Each alternative consists of a sequence of elements.

All elements in the sequence must occur in the specified order.

Any element enclosed in square brackets, [ and ], is optional.

The elements may be any of the following:

- the name of a rule;
- a call on a basic recognizer which tests the input for one of the following:
  - ID - recognizes a lower case identifier,
  - NUM - recognizes a number,
  - SR - recognizes a string enclosed in quotes ("),
  - SRL - recognizes a single character preceded by an apostrophe (')
  - CHR - recognizes any character;
  - a string enclosed in quotes (");
  - a single character string indicated by an apostrophe (') followed by the character;
  - a list of alternatives enclosed in parentheses;
  - a dollar sign ($) followed by an element, which means an arbitrary number of occurrences (including zero) of the element.

Comments are enclosed in percent signs (%) and may be embedded anywhere in the rule.
Rules are terminated by a semicolon (;).
DEFINITIONS

**identifier**

a symbolic name used to identify procedures, executable statements, and variables. (When used to identify executable statements, identifiers are referred to as labels.) In L10 identifiers consist of any number of lowercase letters and/or digits the first of which must be a letter.

**label**

an executable statement identifier enclosed in parentheses and followed immediately by a colon.

**variable**

an identifier which represents a quantity whose value was previously defined, is not yet defined, or may change through the course of the program. L10 variables must be explicitly defined in program declaration statements, in procedure argument lists or LOCAL statements, or must be available as NLS globals.

**indexed variable**

a multi-element variable or array. L10 permits arrays of one dimension only.

**global**

pertaining to a variable whose address in memory is known and accessible throughout all parts of a program. Global variables may be declared in a program or be NLS globals, which the NLS environment defines and which are valid for any L10 program. Through the compiler's knowledge of the correspondence between the identifier and the memory address (contained in the system symbol table), the contents of the memory cell may be changed by program instructions.

**local**

pertaining to a variable whose address in memory is known only to a specific portion of a program, i.e., local to a procedure.
constant

A program element whose value remains unchanged through the programming process. A constant may be a number or literal text (string).

string

A variable or constant consisting of any number of characters enclosed in double quotation marks or a single character preceded by a single quotation mark.

comments

Information enclosed in percent signs (%) which may appear anywhere in the program and are ignored when the program is compiled and executed.

expression

In general, any variable, constant or combination of these joined by operators. L10 also provides some special expression constructions that are peculiar to L10. An expression always has a value.

statement

The basic unit of L10 procedures. L10 statements may consist of many parts: expressions, L10 reserved words, other statements, etc. Unlike expressions, statements do not necessarily have values. L10 statements may be labeled or unlabeled.

execute

To carry out an instruction or "run" a program.
Section 2. PROGRAM STRUCTURE AND PROCEDURES

Introduction

The structure of an L10 program is ALGOL like in its block arrangement. The formal syntax equations for the structure of L10 user programs described below are:

\[
\begin{align*}
\text{program} &= \text{header} \ parts \ "\text{FINISH}"; \\
\text{header} &= \ "\text{PROGRAM}" \ ID; \\
\text{where} \ ID &= \text{the identifier of the first procedure to be executed}; \\
\text{parts} &= \text{procedure} / \text{declare}; \\
\text{procedure} &= \ '(\ ID \ ') \ "\text{PROCEDURE}" \ ['(\ \text{arglist} \ ') \ '] \ body; \\
\text{arglist} &= \ ID \ $(', \ ID); \\
\text{body} &= \$('\ "\text{LOCAL}" \ \text{loco} \ '; / "\text{REF}" \ \text{idlist} ';) \ \text{labeled} \ $('; \ \text{labeled} \ "\text{END}" ;) \\
\text{labeled} &= \ '(\ID'); \ "\text{statement}; \\
\text{idlist} &= \ ID \ $(', \ ID); \\
\text{declare} &= \ (\text{decl/ext/equ/regdec/record/pgdec/refd}) \ '; \\
\text{decl} &= \ "\text{DECLARE}" \ ["\text{EXTERNAL}"] \\
\text{(field} / \text{string} / \text{tp} / \text{stores} / \text{items}); \\
\text{loco} &= \ "\text{STRING}\ 1\text{str} \ $(', \ 1\text{str}) / "\text{TEXT}"\text{"POINTER"} \ \text{idlist} / \ \text{loco} \ $(', \ \text{loco}); \\
\text{1str} &= \ .ID \ /[\ NUM \ '] ;
\end{align*}
\]
NUM gives the maximum length of the local string being declared

loc = .ID /' .NUM '/;

Local declaration of an array of NUM words or a simple variable
USER PROGRAM STRUCTURE

A user program in the NLS environment consists of various procedures and declarations that are prefaced and followed by statements that define the boundaries of the program's text. These elements of the L10 program, which must be arranged in a definite manner with strict adherence to syntactic punctuation, are:

The header -

a statement consisting of the word "PROGRAM" followed by the ID of a procedure in the program. (Program execution will begin with a call to this procedure.) No punctuation occurs between the header and the program body.

The body -

consists of any number of the following in any order:

declaration statements which specify information about the data to be processed by the procedures in the program and cause the data identifiers to be entered into the program's symbol table.

procedures which specify certain execution tasks. Each procedure must consist of -

the procedure identifier enclosed in parentheses followed by the word "PROCEDURE" and optionally an argument list containing names of variables that are passed by the calling procedure for referencing within the called procedure. This statement must be terminated by a semicolon.

the body of the procedure which may consist of LOCAL, REF, and/or statements which may optionally be labeled.

LOCAL is used for declaring data which is to be used only within the current procedure.

REF specifies that the named data elements contain references to other data and when used, the referenced data itself will actually be used.
The procedure terminal statement which consists of the word "END" followed by a period (.).

The program terminal statement which consists of the word "FINISH".
Section 3. VARIABLES, OPERATORS, PRIMITIVES AND EXPRESSIONS

Introduction

This section contains a discussion of the basic elements of the L10 language which when combined with the L10 reserved word commands discussed in the next section, are the building blocks of the L10 statements and hence of L10 programs.

VARIABLES

Five types of variables are described in this document: global, local, referenced, unreferenced, and text pointers.

GLOBAL VARIABLES

A global variable is represented by an identifier and refers to a cell in memory which is known and accessible throughout the program. Global variables are defined in the program's declaration statements or in the NLS system environment.

A global variable may be indexed, i.e., declared as an array. In this case the user must specify the number of elements of the array by following the ID with an expression in square brackets. For example, in a declaration statement sam[10] specifies an array of 10 elements. In an expression however, sam[10] specifies the tenth element of the array sam.

LOCAL VARIABLES

A local variable is represented by an identifier and refers to a cell in memory which is known and accessible only to the procedure in which it appears. Local variables must appear in a procedure argument list or be declared in a procedure's LOCAL declaration statement.

Local variables in the different procedures may have the same identifier without conflict. A global identifier may not be declared as a local identifier and a procedure identifier may be used as neither. In such cases the ID is considered to be multiply defined and an error results.
A local variable may be indexed, i.e., declared as an array. In a local array declaration the user must specify the number of elements of the array by following the ID with an expression in square brackets. For example, `odd[6]` specifies an array of 6 elements.

**REFERENCED VARIABLES**

A variable which represents a pointer to something rather than the thing itself may be passed as an argument to a procedure. If, in the called procedure, one wishes to access the data referenced by the pointer, the pointer identifier may be declared to be a reference using the `REF` construction.

A pointer to a cell in memory may be passed by a calling procedure. A convenient way to access the contents of the cell is to declare the variable to be "referenced" in the procedure through the use of the "REF" construction.

If a variable has been `REF'd`, within the scope of the reference (usually a procedure in which it occurs, although a variable may be `REF'd` through an entire file if desired), whenever the variable is used, that which is pointed to will actually be used.

**UNREFERENCED VARIABLES**

If it is desired to use again a pointer to a variable which has been `REF'd`, one may "unref" it by prefacing the relevant ID with an ampersand (`&`).

**TEXT POINTERS**

A text pointer is an L10 feature used in string manipulation constructions. It is a multi-word entity which provides information for pointing to particular locations within text whether free standing strings or strings which contain the text for an NLS file statement. A text pointer consists of a string identifier and a character count. A string may be a declared string, literal string, or a string which contains text of an NLS statement or an NLS file.
The text pointer points between two characters in a
statement or string. By putting the pointers between
characters a single pointer can be used to mark both
the end of one substring and the beginning of the
substring starting with the next character thereby
simplifying the string manipulation algorithms and
the way one thinks about strings.
Variables, Operators, Primitives and Expressions

Operators

Logarithmic operators

Every numeric value also has a logical value. A numeric value not equal to zero has a logical value of true; a numeric value equal to zero has a logical value of false.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Evaluation</th>
</tr>
</thead>
</table>
| OR       | a OR b = true if a = true or b = true  
           | = false if a = false and b = false     |
| AND      | a AND b = false if a = false or b = false 
           | = true if a = true and b = true         |
| NOT      | NOT a = false if a = true               
           | = true if a = false                     |

Relational operators

A relational operator is used in an expression to compare one quantity with another. The expression is evaluated for a logical value. If true, its value is 1; if false, its value is 0.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
<td>4+1 = 3+2 (true, =1)</td>
</tr>
<tr>
<td>#</td>
<td>not equal to</td>
<td>6#8 (true, =1)</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>6&lt;8 (true, =1)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>8&lt;=6 (false, =0)</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>3&gt;8 (false, =0)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>8&gt;=6 (true, =1)</td>
</tr>
</tbody>
</table>

NOT may precede any other relational operator.
Interval operators

The interval operators permit one to check whether the value of a primitive falls in or out of a particular interval.

\[
\text{IN} \quad \text{intrel}
\]

\[
\text{OUT} \quad \text{intrel} \quad \text{%equivalent to NOT IN%}
\]

\[
\text{intrel} = (\text{' ( / '}) \text{ opexp '}, \text{ opexp (')} / '))
\]

The opexp\(s\) are values separated by operators against which the operand is tested to see whether or not it lies within (or outside of) a particular interval. Each side of the interval may be "open" or "closed". Thus the values which determine the boundaries may be included in the interval (by using a square bracket) or excluded (by using parentheses).

Example:

\[
x \quad \text{IN} \quad (1,100)
\]

is the same as

\[
(x \geq 1) \quad \text{AND} \quad (x < 100)
\]

Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary +</td>
<td>positive value</td>
</tr>
<tr>
<td>unary -</td>
<td>negative value</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>integer division (remainder not saved.)</td>
</tr>
<tr>
<td>MOD</td>
<td>a , \text{MOD} , b , \text{gives the remainder of} , a , / , b</td>
</tr>
</tbody>
</table>

**Page Dimensions:** 613.4x793.1

**ARC & APR 72 9246**

**Variables, Operators, Primitives and Expressions**
.V  a .V b = bit pattern which has 1's wherever either an a or b had a 1 and 0 elsewhere.

.X  a .X b = bit pattern which has 1's wherever either an a had a 1 and b had a 0, or a had a 0 and b had a 1, and 0 elsewhere.

.A  a .A b = bit pattern which has 1's wherever both a and b had 1's, and 0 elsewhere.
PRIMITIVES

Primitives are the basic units which are used as the operands of LILO expressions. There are many types of elements that can be used as LILO primitives; each type returns a value which is used in the evaluation of an expression.

Each of the following is a valid primitive:

- variable -
  any valid variable identifier

- procname args -
  a procedure call with argument list

- variable ' + exp -
  an assignment statement

- pointer -
  a pointer, possibly a text pointer or a reference to any other type of array

- literal -
  a numeric constant or character constant

- string = '* stringname '*' / .SR;

  It is possible to compare variable or literal strings.

- charclass -
  provides a simple way to test the common classes of characters; described in detail below

- "MIN" ' ( exp $'(', exp ')' )
- "MAX" ' ( exp $'(', exp ')' )

  Select the minimum or maximum, respectively, of the values of a list of expressions.

- "READC" -
a character is read from the current character position and in the direction as set by the last scan. This facility is described later in this document under string manipulation.

"CCPOS" -

the value of the index of the character to the right of the current character position. This facility is described later in this document under string manipulation.

"FIND" stringstuff -

used to test text patterns and load text pointers for use in string construction (see the STRING MANIPULATION section); return the value TRUE or FALSE depending on whether or not the string tests within it succeed.

"POS" posrel -

may be used to compare two text pointers

Procedure Calls

When a procedure call is used as a primitive, the value is that of the leftmost result returned by the procedure.

procname args

Where

procname =

ID, a procedure identifier

args =

'( (exp $(', exp) /': var $('. var)) ')';

exp =

any valid L10 expression. A set of expressions separated by commas constitute the argument list for the procedure.
variables. All but the leftmost variables are used to store the results of the procedure.

The argument list consists of an arbitrary number of expressions separated by commas. It is not necessary for the number of arguments to equal the number of formal parameters for the procedure (although this is generally a good idea). The argument expressions are evaluated in order from left to right.

Following the arguments there may be a list of locations for multiple results to be returned. The list of variables for multiple results is separated from the list of argument expressions by a colon. The number of locations for results need not equal the number of results actually returned. If there are more locations than results, then the extra locations get an undefined value. If there are more results than locations, the extra results are simply lost.

Example:

If procedure p ends with the statement

RETURN (a,b,c)

then the statement

q = p(r,s)

results in (q,r,s) + (a,b,c).

Assignments

An assignment can be used as a primitive.

The form a = b has the effect of storing b into a and has the value of b as its value.

Pointers

A string or an identifier preceded by a dollar sign ($) represents a pointer to that string or the variable represented by the identifier.

pointer = '$' (ID / SR)
Literals

A literal is a constant which returns a numerical value. A literal may be any of the following:

- **NUM**
- "TRUE"
- "FALSE"
- **char**

There are several ways in which numeric values may be represented. A sequence of digits alone or followed by a **D** is interpreted as base ten. If followed by a **B** then it is interpreted as base eight. A scale factor may be given after the **B** for octal numbers or after a **D** for decimal numbers. The scale factor is equivalent to adding that many zeros to the original number.

**Examples:**

\[
\begin{align*}
64 & = 100B = 1B2 \\
144B & = 100 = 1D2
\end{align*}
\]

The words **TRUE** and **FALSE** are equivalent to the numbers 1 and 0 respectively.

Characters may be used as literals as they are represented internally by numeric values. The following are synonyms for commonly used characters:

- **SR1** - any single character preceded by an apostrophe e.g. 'a represents the code for the character a and is equal to 1h1b.
- "ENDCHR" - end character as returned by **READC**
- "SP" - space
- "EOL" - Tenex's version of CR LF
- "ALT" - Tenex's version of altmode or escape (=33b)
- "CR" - carriage return
"LF" - line feed
"TAB" - tab
"BC" - backspace character
"BW" - backspace word
"C." - center dot

CA - Command Accept
CD - Command Delete;

Character classes

charclass =

"CH" /
  %any character%

"ULD" /
  %uppercase letter or digit%

"LLD" /
  %lowercase letter or digit%

"LD" /
  %lowercase or uppercase letter or digit%

"NLD" /
  %not a letter or digit%

"UL" /
  %uppercase letter%

"LL" /
  %lowercase letter%

"L" /
  %lowercase or uppercase letter%

"D" /
  %digit%

"PT" /
  %printing character%
"NP"
%nonprinting character%;

Example:

char = LD

is true if the variable "char" contains a value which is a letter or a digit.

MIN and MAX

These primitives return the lowest/highest value expression in the expression list specified.

Example: if a = 3, b = 2, c = 4 at time MIN and MAX called, then MIN(a, b, c) = b (=2) and MAX(a, b, c) = c (=4).

READC

The primitive READC is a special construction for reading characters from NLS statements or strings.

A character is read from the current character position in the scan direction set by the last CCPOS statement or string analysis FIND statement or expression. This feature is explained in detail later in this document, under String Manipulation.

Attempts to read off the end of a string in either direction result in a special "endcharacter" being returned and the character position is not moved. This endcharacter is included in the set of characters for which system mnemonics are provided and may be referenced by the identifier "ENDCHR".

Example:

to sequentially process the characters of a string

CCPOS *str*;
UNTIL (char + READC) = ENDCHR DO process(char).

(Note: READC may also be used as a statement if it is desired to read and simply discard a character).
When used as a primitive, CCPOS has as its value the index of the character to the right of the current character position. CCPOS is more commonly used to set the current character position for use in text pattern matching. This is discussed in detail in section 6 (7b) below.

Examples:

If str = "glarp", then after CCPOS *str*, the value of CCPOS is 1 and after CCPOS SE(*str*) the value of CCPOS is 6 (one greater than the length of the string).

To sequentially process the first n characters of a string (assumed to have at least n characters)

CCPOS *str*;
UNTIL CCPOS > n DO process(READC).

Text Pointer Comparisons

posrel =

pos ["NOT"] ('=' / '#' / '>=' / '<=' / '>' / '<') pos;

This may be used to compare two text pointers.

The pos is a character position pointer (text pointer) in a form discussed in (7b) below.

If the pointers refer to different statements then all relations between them are false except "not equal" which is written '# or "NOT" '='. If the pointers refer to the same statement, then the truth of the relation is decided on the basis of their location within the statement with the convention that a pointer closer to the front of the statement is "less than" a pointer closer to the end.
Introduction

An expression is any constant, variable, special expression form, or combination of these joined by operators and parentheses as necessary to denote the order in which operations are to be performed. Special L10 expressions are: the FIND expression which is used for string manipulation; the conditional IF and CASE expressions which may be used to give alternative values to expressions depending on tests made in the expressions. Expressions are used where the syntax requires a value. While certain of these forms are similar syntactically to L10 statements, when used as an expression they always have values.

ORDER OF OPERATOR EXECUTION—BINDING PRECEDENCE

The order of performing individual operations within an equation is determined by the hierarchy of operator execution (or binding precedence) and the use of parentheses.

Operations of the same hierarchy are performed from left to right in an expression. Operations in parentheses are performed before operations not in parentheses.

The order of execution hierarchy of operators (from highest to lowest) is as follows:

- unary -, unary +
- .A
- .V, .X
- *, /, MOD
- +, -
- relational tests (e.g., >, <, =, =, #, IN, OUT)
- NOT relational tests (e.g., NOT >)
- NOT
- AND
- OR
CONDITIONAL EXPRESSIONS

IF Expressions

IF testexp THEN expl ELSE exp2

testexp is tested for its logical value. If testexp is true then expl will be evaluated. If it is false, then exp2 is evaluated.

Therefore, the result of this entire expression is EITHER the result of expl of exp2.

Example:

\[ y = \begin{cases} % \text{if } x = 1, 2, \text{ or } 3 & y + x; \\ & \text{otherwise } y + 4 \end{cases} \]

CASE Expression

This form is similar to the above except that it causes any one of a series of expressions to be evaluated and used as the result of the entire expression.

\[ \text{CASE testexp OF } \begin{cases} % \text{relist ' : exp ' ; } \end{cases} \text{ "ENDCASE" exp ' ; } \]

relist = RELOP exp \$(', RELOP exp):

Where RELOP = any relational operator

In the above, the testexp is evaluated and used with the operator RELOPs and their respective exps in a relist to test for a value of true or false. If true in any instance the companion exp on the right of the colon is executed and taken to be the value of the whole expression. A value of false for a set of relist tests causes the next relist in the CASE expression to be tested against the testexp. If all relists are false, the ENDCASE expression is taken to be the value of the whole expression.

Example:

\[ \text{CASE } x1 \text{ OF } \]

VARIABLES, OPERATORS, PRIMITIVES AND EXPRESSIONS

<4: xl+1;
=4: xl+2;
=5: xl;
ENDCASE xl*2;

<table>
<thead>
<tr>
<th>Value of Xl</th>
<th>Value of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

STRING EXPRESSIONS

L10 also provides several expression forms which are used for string manipulation and evaluation. These are identical to the string manipulation statements discussed in Section 6 of this document (7). Note that when using string manipulation statement forms as expressions, parentheses may be necessary to prevent ambiguities.
Section 4. DECLARATIONS

Introduction

Ll0 declarations are necessary to provide information to the compiler about the nature of the data that is to be accessed. Declarations are non-executable.

There are various types of declarations available; only the most frequently used are discussed here: DECLARE, REF, and LOCAL.

Program level declarations (DECLARE and REF) may appear anywhere in the program. However, procedure level declarations (LOCAL and REF inside a procedure) must appear before any executable statements in the procedure.

GLOBAL DECLARATIONS

Variables specified in these declarations are global (i.e., outside any procedure) and may be used by all procedures in the program. There are four versions depending on the type of entity to be defined: scalars, arrays, strings, and text pointers. The scalar, array, and string declarations allow the user to initialize the value of the variable(s) specified.

Declaring Scalar Variables

A scalar variable that is to be used throughout a program must be declared in a declaration at the program level. The quantity represented by the scalar variable may be a numeric value, a string, or an address. Optionally, the user may specify the initial value of the variable being declared. If a scalar variable is not initialized at the program level, it should be initialized in the first executed procedure in which it appears.

To declare a scalar variable only: .Grab=6

"DECLARE" ID ';

To declare and initialize a scalar variable:

"DECLARE" ID ' = CONSTANT ';

L10 Programming Guide  Section 4  (page 33)
Where ID = the name of the variable being declared.

CONSTANT =

the initial value of ID. It may be any of the following:

- a numeric constant optionally preceded by a unary minus sign (-)
- a string enclosed in quotation marks
- another identifier (causing the latter's address to be used as the value of the ID being declared)

Examples:

```
DECLARE x1;  %x1 is not initialized
DECLARE x2=5;  %x2 contains the value 5
DECLARE x3="OUT";  %x3 contains the word OUT
DECLARE xx=x1;  %xx contains the address of x1
```

Declaring Array Variables

If the user intends to use any array variables throughout the program, he must specify the number of elements of the array at the program level. Optionally, he may specify the initial value of each element of the array. If array values are not initialized at the program level, they should be initialized in the first executed procedure in which the array is used.

To declare an array variable only:

```
"DECLARE" ID '{ NUM '} ;
```

To declare and initialize an array variable:

```
"DECLARE" ID ' = ' ( CONSTANT $ ( ',CONSTANT ) ' ) ' ;
```

Where ID = the name of the variable being declared.

NUM = the number of elements in the array

if the array is not being initialized.
CONSTANT = the initial value of each element of the array. The number of constants implicitly define the number of elements in the array. They may be any of the following:
- a numeric constant optionally preceded by a unary minus (-)
- a string enclosed in quotation marks
- another identifier (causing the latter's address to be used as the value of the ID being declared)

Note: there is a one-to-one correspondence between the first constant and the first element, the second constant and the second element, etc.

Examples:

DECLARE sam[10];
%declares an array named sam containing 10 elements which are not initialized%

DECLARE nums=(1,2,3);
declares an array named nums containing 3 elements which are initialized such that:
    nums = 1
    nums(1) = 2
    nums(2) = 3

DECLARE motley=(10,words);
declares an array named motley containing 2 elements which are initialized such that:
    motley = 10
    motley(1) = the address of the variable words
Declarations

Declaring Many Scalars and/or Arrays in One Statement

One may avoid putting several individual declarations of items (i.e., several statements each beginning with the word DECLARE) by putting items and arrays to be declared, initialized or not, in a list in one statement following a single DECLARE separated by commas and terminated by a semi-colon.

Example:

DECLARE x, y[10], z = (1, 2, -5);

Declaring Strings

The DECLARE STRING enables the user to declare a global string variable by initializing the string and/or declaring its maximum character length. Any number of strings may be declared in the same statement.

To declare a number of strings:

"DECLARE STRING" ID '('ID') $(',')$('ID')' ';

To declare and initialize a number of strings:

"DECLARE STRING" ID'=STRING $(',')$'STRING')';

Where ID = the name of the string being declared

NUM = the maximum number of characters allowed for the string

STRING = a string constant enclosed in double quotation marks. The length of this string defines the maximum length of the corresponding ID.

Strings have two associated values, maximum length and current length. When strings are simply declared, maximum length is specified by NUM and current length is 0; when strings are initialized in a declaration statement, maximum length is equal to current length.

These numbers may be accessed by specifying the name of the string followed by a period and the letters M or L respectively.
Examples:

DECLARE STRING lstring[100];

declares a string named lstring with a maximum length of 100 characters and a current length of 0 characters

DECLARE STRING message="RED ALERT", warn="WARNING", help[50];

declares three strings message, warn, and help such that:

message has an actual and maximum length of 9 characters and contains the text "RED ALERT"

warn has an actual and maximum length of 7 characters and contains the text "WARNING"

help has a actual length of 0 and a maximum length of 50 characters, i.e. help.M = 50 and help.L = 0

Declaring Text Pointers

The DECLARE TEXT POINTER declaration enables the user to declare global variables as text pointers that are used in string manipulation and construction.
REFERENCE DECLARATIONS

Unlike the other declarations discussed here, the REF statement does not allocate storage; it simply defines the use of the variable(s) specified as references.

A variable which contains a pointer to something rather than the thing itself may be passed as an argument to a procedure. If, in the called procedure, one wishes to access the thing itself, the pointer identifier may be declared to be a reference using the REF construction.

If a variable has been REF'd, within the scope of the reference (usually a procedure in which it occurs, although a variable may be REF'd through an entire file if desired) when the variable is accessed as a normal variable, the value of the cell being pointed to is actually used.

Example:

If x contains the address of y and x has been REF'd, then:

\[
\begin{align*}
z & \leftarrow x; \quad (=z+Y) \\
x & \leftarrow z \quad (=y+z)
\end{align*}
\]

This is equivalent (without REF'ing) to:

\[
\begin{align*}
z & \leftarrow [x]; \\
[x] & \leftarrow z;
\end{align*}
\]

Referenced variables may be "unreferenced" by preceding their identifiers by the ampersand character "&". Unreferencing a variable causes it to be interpreted as a pointer. Thus, any variable name may serve a dual function of pointing to an address as well as designating the contents at that address.

"REF" ID $(',ID)';

Local variables may be declared as references by a REF declaration among declarations in a procedure (see below).
LOCAL DECLARATIONS

The LOCAL declaration consists of several forms that are equivalent to those of the global DECLARE forms except that variables declared in a LOCAL declaration may be used only by the procedure in which they appear. Also, LOCAL declarations do not provide for the initialization of variables.

Any LOCAL declarations must precede the executable statements in a procedure.

To declare a local scalar variable only:

"LOCAL" ID ';

To declare a local array variable only:

"LOCAL" ID ' [ NUM ' ] ';

Again lists of items separated by commas may be declared locally.

To declare a local string only:

"LOCAL STRING" ID ' ( NUM ' ) $ ( ' , ID ' ( NUM ' ) ) ';

To declare a local text pointer:

"LOCAL TEXT POINTER" ID $ ( ' , ID ) ';
ASSIGNMENT

ASSIGN STATEMENT

In the ASSIGN statement the expression on the right side of the "=" is evaluated and stored in the variable on the left side of the statement.

```plaintext
var ' = exp ';
```

where var = any global, local, referenced or unreferenced variable.

MULTIPLE ASSIGN STATEMENT

In the MULTIPLE ASSIGN statement the expressions are evaluated and the values pushed on a stack provided by the system. Then the values are popped from the stack and stored into the appropriate left hand side. The order of evaluation of the expressions is left to right.

```plaintext
'( var $(' , var ) ') ' = ' ( exp $(' , exp ) ');
```

where var = any global, local, referenced or unreferenced variable.

Naturally, the number of expressions must equal the number of var’s.

Example:

```plaintext
(a, b) = (a+b, a-b)
```

the expression a+b is evaluated and stacked, expression a-b is evaluated and stacked, the value of a-b is popped and stored into b, and finally, the value of a+b is popped and stored into a.
DIVIDE STATEMENT

The divide statement permits both the quotient and remainder of a division to be saved. The syntax for the divide statement is as follows:

"DIV" exp ' ', quotient ' ', remainder

The central connective in the expression must be '/'. Quotient and remainder are the identifiers in which the respective values will be saved upon the division.

BLOCK

The BLOCK construction enables the user to group several (labeled) statements into one syntactic statement entity. A block construction of any length is valid where a statement is required.

"BEGIN" $( statement ';' ) "END"

Where statement = any executable LIO statement, labeled or unlabeled.

Example:

BEGIN
  a←b;
  c←d+5;
  xx+yy;
  (nono):d+a+c;
END

is equivalent to:

a←b;
  c←d+5;
  xx+yy;
  (nono):d+a+c;

but may be used in an instance in which the syntax requires one statement. (See, for example, the LOOP construction below.)
There are two types of conditional statements described below-- the common IF statement with optional ELSE and the CASE statement.

**IF Statement**

This form causes execution of a statement (which may be a block) if a tested expression is true. If it is false and the optional ELSE part is present, the statement following the ELSE is executed. If no ELSE part is present, control passes to the statement immediately following the IF statement.

```
"IF" testexp "THEN" labeledstatement ["ELSE" labeledstatement]
```

testexp is tested for its logical value. If testexp is true then the statement following the THEN will be executed. If it is false and an optional ELSE part is present, then the statement following the ELSE will be executed; otherwise the next statement after the IF statement will be executed.

**CASE Statement**

This form is similar to the above except that it causes any one of a series of statements to be executed depending on the result of a series of tests.

```
CASE testexp OF $( relist ': labeledstat ');
"ENDCASE" labeledstat ';
```

relist = RELOP exp $(', RELOP exp);

where RELOP = any relational operator (>=, <, =, IN, etc.)

The CASE-statement provides a means of executing one statement out of many. The expression after the word "CASE" is evaluated and the result left in a register. This is used as the left-hand side of the binary relations at the beginning of the various cases. Several relations may be listed at the start of a single statement; the statement will be executed if any of the relations is satisfied. If none of the relations is satisfied, the statement following the word "ENDCASE" will be executed.
Example:

CASE c OF
    = a, <d: x + y; %Executed if c = a or c < d%
    > b: (x, y) + (x+y, x-y); %Executed if c > b%
ENDCASE v + x; %Executed otherwise

ITERATIVE

The statements described here enable the user to alter the normal sequence of execution within a procedure and/or to cause the repeated execution of a set of statements until some condition is met.

LOOP STATEMENT

The statement following the word "LOOP" is repeatedly executed until control leaves by means of some transfer instruction within the loop.

"LOOP" statement:

where statement = any executable LLO statement (including a block), labeled or unlabeled.

Example:

LOOP
BEGIN
a := a * a + 1;
b := a + b;
IF a > 200 THEN EXIT;
END;

It is assumed that a and b have been initialized before entering the loop. The EXIT construction is described below.
WHILE...DO STATEMENT

This statement causes a statement (or block of statements) to be repeatedly executed as long as the expression immediately following the word WHILE has a logical value of true or control has not been passed out of the DO loop by some explicit transfer.

"WHILE" exp "DO" statement

exp is evaluated and if true the statement following the word DO is executed; exp is then reevaluated and the statement continually executed until exp is false. In this event control will pass to the next sequential statement.

Example:

WHILE alpha DO
BEGIN
  zygo + b+b;
  alpha + alpha = 1;
END;

If alpha has a value of +5 (logically true) when this statement is executed, the statement following "DO" will be executed 5 times as alpha is decremented by one each time the statement is executed. Once alpha is equal to zero (false) the next statement will be executed.
UNTIL...DO STATEMENT

This statement is similar to the WHILE...DO statement except that statement(s) following DO are executed until exp is true. As long as exp has a logical value of false the statement(s) will be executed repeatedly.

"UNTIL" exp "DO" statement
DO...UNTIL/WHILE STATEMENT

This statement is like the preceding statement, except that the logical test is made after the statement has been executed rather than before.

"DO" statement ("UNTIL" / "WHILE") exp;

Thus the specified statement is always executed at least once (the first time, before the test is made).
FOR STATEMENT

The FOR statement causes the repeated execution of the statement following "DO" until a specific terminal value is reached.

"FOR" var ['+ expl] ("UP" / "DOWN") [exp2]
"UNTIL" (relop) exp3 "DO" statement;

where var = the variable whose value in incremented/decremented each time the FOR statement is executed

expl = an optional initial value for var. If expl is not specified, the current value of var is used.

exp2 = an optional value by which var will be incremented (if UP specified) or decremented (if DOWN specified). If exp2 is not specified, a value of one will be assumed.

relop = any relational operator

exp3 = when combined with relop determines whether or not another iteration of the FOR statement will be performed.

Note that exp2 and exp3 are recomputed on each iteration.

Example:

FOF k + n UP j UNTIL > m*3 DO x[k] + k;

is equivalent to

k + n;
GOTO test;
(loop): k = k + j;
(test): IF k > m*3 THEN GOTO out;
x[k] = k;
GOTO loop;
(out):
These statements in general cause the unconditional transfer of control from one part of a program to another part.

PROCEDURE CALL STATEMENT

This statement is used to direct program control to the procedure specified.

`procname args`

Where `procname` = ID, a procedure identifier

`args = '(' [exp & ('
','exp')] ':
var & ('
','var')]')';`

`exp` = any valid L10 expression. The set of expressions separated by commas is the argument list for the procedure.

`var` = any variable. The set of variables is used to store the results of the procedure if there is more than one result.

The argument list consists of an arbitrary number of expressions separated by commas. It is recommended (although not necessary) for the number of arguments to equal the number of formal parameters for the procedure. The argument expressions are evaluated in order from left to right.

Following the arguments there may be a list of locations for multiple results to be returned. The list of variables for multiple results is separated from the list of argument expressions by a colon. The number of locations for results need not equal the number of results actually returned. If there are more locations than results, then the extra locations get an undefined value. If there are more results than locations, the extra results are simply lost.

Example:

If procedure `p` ends with the statement

`RETURN (a,b,c)`
then the statement

q = p(r,s);

results in (q,r,s) + (a,b,c).

A procedure call may just exist as a statement alone without returning a value:

z();
RETURN STATEMENT

This statement causes a procedure to return an arbitrary number of results. The order of evaluation of results is from left to right.

"RETURN" ['( exp $(', exp ')')]

GOTO STATEMENT

Goto provides for unconditional transfer of control to a new location.

"GO""TO" ID

The ID is the name of a label elsewhere in the program.

EXIT STATEMENT

This construction provides for forward branches out of CASE or iterative statements. The optional number (NUM) specifies the number of lexical levels of CASE or iterative statements respectively that are to be exited. If a number is not given then 1 is assumed. All of the iterative statements (LOOP, WHILE, UNTIL, DO, FOR) can be exited by the EXIT LOOP construct.

"EXIT" ("CASE" [NUM] / ["LOOP"] [NUM])

EXIT and EXIT LOOP have the same meaning.

Examples:

LOOP
BEGIN
 ..........  
IF test THEN EXIT;
%the EXIT will branch out of the LOOP%
 ..........  
END;
UNTIL something DO
BEGIN
• • • • • • • •
WHILE test1 DO
BEGIN
• • • • • • • •
IF test2 THEN EXIT;
%the EXIT will branch out of the WHILE%
• • • • •
END;
• • • • • • • •
END;
UNTIL something DO
BEGIN
• • • • • • • •
WHILE test1 DO
BEGIN
• • • • • • • •
IF test2 THEN EXIT 2;
%the EXIT 2 will branch out of the UNTIL%
• • • • •
END;
• • • • • • • •
END;
CASE exp OF
=something: BEGIN
• • • • • • • •
IF test THEN EXIT CASE;
%the EXIT will branch out of the CASE%
• • • • •
END;
• • • • • • • •
END;
7f5b1b
7f5b1c
7f5b1d
REPEAT STATEMENT

This construction provides for backward branches to the front of CASE or conditional statements. The optional number (NUM) has the same meaning as in the EXIT statement.

"REPEAT" ["LOOP" [NUM] / ["CASE"] [NUM] [ '(' exp ')']]

If an expression is given with the REPEAT CASE, then it is evaluated and used in place of the expression given at the head of the specified CASE statement. If the expression is not given, then the one at the head of the CASE statement is reevaluated.

It is worth noting that the availability of EXIT and REPEAT statements has resulted in clearer programs which are generally without labels and GOTO's. The EXIT and REPEAT replace GOTO's to the start or end of the most common compound forms. By providing implicit labels in these positions for use with EXIT or REPEAT, explicit labels are avoided.

REPEAT and REPEAT CASE have the same meaning.

Examples:

```ll
CASE expl OF
    =something:
    BEGIN
        .........
        IF test1 THEN REPEAT;
        %REPEAT with a reevaluated expl%
        .........
        IF test2 THEN REPEAT(exp2);
        %REPEAT with exp2%
        .........
        END;
        .........
LOOP
BEGIN
    .........
    IF test THEN REPEAT LOOP;
    %REPEAT LOOP will go to the top of the LOOP%
    .........
    END;
```
NULL STATEMENT

The NULL statement may be used as a convenience to the programmer. It is a no-op.

```
null = "NULL";
```

7g

7g1

7g1a
Section 6. STRING TEST AND MANIPULATION

INTRODUCTION

The following special statements allow for complex string analysis and construction. The three basic elements of string manipulation discussed here are the Current Character Position (ccpos) and text pointers which allow the user to delimit substrings within a string, patterns that cause the system to search the string for specific occurrences of text and set up pointers to various textual elements, and actual string construction.

The content analysis facility of NLS may be invoked using similar search patterns without the pointer-loading capabilities.

CURRENT CHARACTER POSITION (CCPOS) AND TEXT POINTERS

The Current Character Position is similar to the TNLS CM (current marker) in that it specifies the location in the string at which subsequent operations are to begin. All L10 string tests start their search from the current character position.

"CCPOS" (pos / '* stringname '*/ [ '/ exp ']));

pos is a position in a statement or string that may be expressed as any of the following:

A previously declared and set text pointer ID

The scan direction over the text will remain unchanged. The direction of scanning may be set implicitly using the string front of string end facilities or explicitly using the direction setting "<" or ">" in an earlier pattern. (See "Other parameters" under PATTERNS below.)

String Front -- left of the first character

"SF(" stspec ")"

When SF is specified scanning will take place from left to right within the string.
"stspec" is a string specification that may be expressed as a previously declared text pointer ID or previously declared string ID enclosed in asterisks.

String End -- right of the last character

"SE(" stspec ")"

When SE is specified scanning will take place from right to left within the string.

A text pointer points between two characters in a string.

The variable holding a text pointer is declared by a DECLARE TEXT POINTER or LOCAL TEXT POINTER statement. There is a special declaration for these because text pointers require more than a single word of storage. The identifier used as a text pointer may be such a variable or a reference, defined by a REF statement, to such a variable.

If a text pointer is given after COPOOS, then the character position is set to that location.

If a stringname ("* stringname") is given after COPOOS, then the position is moved to that string. The scan direction is set left to right.

Indexing the stringname (by specifying '/ exp '/') simply specifies a particular position within the string. Thus "str*[3]" puts the current character position between the second and third characters of the string "str". If the scan direction is left to right, then the third character will be read next. If the direction is right to left, then the second will be read next.

If no indexing is given, then the position is set to the left of the first character in the string. This is equivalent to an index of 1.
PATTERNS - the FIND statement and CONTENT ANALYSIS patterns

FIND Statements and Expressions

This statement specifies a string pattern to be tested and text pointers to be manipulated and set starting from the current character position. If the test succeeds the character position is moved past the last character read. If the test fails the character position is reset to the position prior to the test and the values of all text pointers set within the pattern will be reset.

"FIND" $strentity;

FINDs may be used as expressions as well as free-standing elements. If used as an expression, for example in IF statements, it has the value "TRUE" if all pattern elements within it are true and the value "FALSE" if one of the elements is false.

Content Analysis Patterns

Content analysis patterns are simply string pattern entities followed by a semi-colon. When placed in an NLS file and "compiled" using the Execute Content Analyzer command, the pattern may be invoked using a special viewspec to search through an NLS file for statements satisfying the patterns. (The process is described in detail in sections 7 and 8 below.)

Implicit in Content Analysis patterns is the notion that they will start a pattern matching search at the beginning of each NLS text statement.

Certain of the arguments are valid only in the context of complete L10 programs. These are noted below.

Because text pointers may not be loaded in Content Analysis patterns and because strings may not be reconstructed in them, they may only be used effectively in relatively simple cases. In more complex situations, full L10 programs are necessary.

String pattern entities-- ($strentities)
A string entity (strenity) may be any valid combination of the following: logical operators, testing arguments, and other non-testing parameters which in general cause repositioning within the current string.

Logical Operators-- These combine and delimit groups of patterns. Each compound group is considered to be a single pattern with the value TRUE or FALSE. If text pointers are set within a test pattern and the pattern is not true, the values of those text pointers are reset to the values they had before the test was made. (See examples below.)

"OR" -
Either of the two separated groups must be true for the pattern to be true.

"AND" -
Both of the two separated groups must be true for the pattern to be true.

"NOT" -
The following pattern group must not be true for the pattern to be true.

"I" -
Either of the two separated groups must be true for the pattern to be true. Has lower precedence than OR, i.e., binds less tightly than "OR".

Pattern Matching Arguments-- (each of these can be true or false)

These may appear in Content Analysis patterns:

SR

string constant, e.g. "ABC"
It should be noted that if the scan direction is set right to left the pattern string constant pattern should be reversed. In the above example, one would have "CBA".

char

any character

charclass

look for a character of a specific class (see primitives for a list of character classes) If found, = true, otherwise false.

'( strentity ')

look for an occurrence of the pattern specified by strentity. If found, = true, otherwise false.

'=' parameter

True only if the parameter following the dash does not occur.

'/' strentity '

true if the pattern specified by strentity can be found anywhere in the remainder of the string. First searches from current position. If the search failed, then the current position is incremented by one and resets. Incrementing and searching continues until the end of the string. The value of the search is false if the testing string entity is not matched before the end of the string is reached.

NUM argument

find (exactly) the specified number of occurrences of the argument.
NUM1 'S' NUM2 argument

Tests for a range of occurrences of the argument specified. If the argument is found at least NUM1 times and at most NUM2 times, the value of the test is true. Either number is optional. The default value for NUM1 is zero. The default value for NUM2 is 10000. Thus a construction of the form "$3 CH" would search for any number of characters (including zero) up to and including three.

"ID" ('#/=') UID

if the string being tested is the text of an NLS statement then the identifier of user who created the statement is tested by this construction.

"SINCE" datim

if the string being tested is the text of an NLS statement, this test is true if the statement was created after the date and time (datim, see below) specified.

"BEFORE" datim

if the string being tested is the text of an NLS statement, this test is true if the statement was created before the date and time (datim, see below) specified.

These may not appear in Content Analysis patterns:

'*' stringname '

string variable

"BETWEEN" pos pos ( strentity ')

Search limited to between positions specified. Scan character position is set to first position before the pattern is tested.
Format of date and time for pattern matching

datim = ' ( date time )'

Acceptable dates and times follow the forms permitted by the TENEX system's IDTIM JSYS described in detail in the JSYS manual. It accepts "most any reasonable date and time syntax."

Examples of valid dates:

17-APR-70
APR-17-70
APR 17 70
APRIL 17, 1970
17 APRIL 70
17/5/1970
5/17/70

Examples of valid times:

1:12:13
123h
16:30 (4:30 PM)
123h:56
1:56AM
1:56-EST
1200NOON
12:00:00AM (midnight)
11:59:59AM-EST (late morning)
12:00:01AM (early morning)

Other Arguments-- (these do not involve tests; rather, they involve some execution action. They are always TRUE for the purposes of pattern matching tests.)

These may appear in simple Content Analysis Patterns:

'<' -

set scan direction to the left
In this case, care should be taken to specify patterns in reverse, that is in the order which the computer will scan the text.

'>' -

set scan direction to the right

"TRUE" -

has no effect; it is generally used at the end of FIND when a value of true is desired even if all tests fail.

These may not appear in simple Content Analysis Patterns:

`pos -`

set current character position to this position. If the SE pointer is used, set scan direction from right to left. If the SF pointer is used, set scan direction from left to right.

`↑ ID -`

store current scan position into the textpointer specified by the identifier

`↑ [NUM] ID -`

back up the specified text pointer by the specified number (NUM) of characters. Default value for NUM is one. Backup is in the opposite direction of the current scan direction.
STRING CONSTRUCTION

String constructions allow the replacement of one string (substring) by another string.

"ST" (pos / substr) '← stlist /

'* stringname '* / exp "TO" exp'J) '← stlist;

The string to which pos or stringname refers is replaced by the string specified to the right of the arrow. A substring is replaced if a substr or an indexed stringname is specified.

Examples:

ST p1 p2 + string;
is equivalent to
ST p1 + SF(p1) p1, string, p2 SE(p2);

*str*/lower TO upper/ + string;
is equivalent to
*str* + *str*/l TO lower-l, string, *str*/upper+l TO str.L/;

stlist = stprim $(', stprim);

stprim =

"NULL" /

represents the zero length string

SR /

for string constant, e.g. "ABC"

substr /

substring

'← substr /

substring capitalized

'= substr /

substring in lower case
If it is preceded by a dollar sign ($), then the substring is copied without moving any associated markers to the new position. This element is relevant only if the string is the text of an NLS statement.

for string variables

for character variables

substring by indices

A construction of the form \texttt{*\textquoteleft\textquoteleft str\textquoteleft\textquoteleft[i TO j]} refers to the substring starting with the \textit{i}th character in the string up and including the \textit{j}th character. Thus \texttt{*\textquoteleft\textquoteleft str\textquoteleft\textquoteleft[i TO 1:10]} is the eleven character substring starting with the \textit{i}th character of \textit{str}. and \texttt{*\textquoteleft\textquoteleft str\textquoteleft\textquoteleft[i TO str.L]} is the string \textit{str} with the first \textit{i}-1 characters deleted.

value of a general L10 expression taken as a character; i.e., the character with the ASCII code value equivalent to the value of the expression

\texttt{"STRING\textquoteleft\textquoteleft ( exp \textquoteleft\textquoteleft , exp\textquoteleft\textquoteleft )\textquoteright\textquoteright\textquoteright};

\texttt{substr = pos pos;}

This is the substring bounded by the two positions.
Example:

Let a "word" be defined as an arbitrary number of letters and digits. The two statements in this example delete the word pointed to by the text pointer "t", and if there is a space on the right of the word, it is also deleted. Otherwise, if there is space on the left of the word it is deleted.

The text pointers x and y are used to delimit the left and right respectively of the string to be deleted.

LD is true if the character is a letter or a digit, and SP is true if the character is a space.

FIND t < $LD ↑x t > $LD (SP ↑y / ↑y x < (SP ↑x / TRUE));
ST x y + NULL;

The reader should work through this example until it is clear that it really behaves as advertised.

The new string or substring is specified as a concatenation of string primaries, with the primaries separated by commas.
Section 7. CONTENT ANALYSIS AND SEQUENCE GENERATOR PROGRAMS

Introduction

NLS provides a variety of commands for file manipulation and viewing. All of the editing commands, and the print command with associated viewspecs (like line truncation and statement numbers) provide examples of these manipulation and viewing facilities.

But occasionally one may need more sophisticated view controls than those available with the viewspec and viewchange features in NLS.

For example, one may want to see only those statements that contain a particular word or phrase.

Or one might want to see one line of text that compacts the information found in several longer statements.

One might also wish to perform a series of routine editing operations without specifying each of the NLS commands over and over again.

The Network Information Center at ARC uses the ability to create text using the information from several different statements (and even different files) and the ability to insert this new text into a file to produce catalogues and indices.

User written programs enable one to tailor the presentation of the information in a file to his particular needs. Experienced users may write programs that edit files automatically.

CREATION OF USER WRITTEN PROGRAMS

User written programs must be coded in L10. They may call other user written routines and various procedures in the NLS program itself.

User programs that control the way material is portrayed take effect when NLS presents a sequence of statements in response to a command like Print Group.
In processing a command such as Print NLS looks at a sequence of statements, examining each statement to see if it falls within the range specified in the Print command and if it satisfies the viewspecs. At this point NLS may also pass the statement to a user written program to see if it satisfies the requirements specified in that program. If the user program returns a value of true, the (passed) statement is printed and the next statement in the sequence is tested; if false, the next statement in the sequence is tested.

User programs that modify files usually gain control at the same point in processing as those that control the view.

Typically, one wants such a program to operate on a sequence of statements chosen by a user when he decides to run the program. In addition, one usually wants to see the results of such an automated series of editing operations immediately after it happens.

Although a user program may be called explicitly (using a special purpose NLS command), it is usually invoked when one asks to view a part of the file.

CONTEXT OF USER WRITTEN PROGRAMS -- THE PORTRAYAL GENERATOR

Generally, the user written program runs in the framework of the portrayal generator. It may be invoked in several ways, described below, whenever one asks to view a portion of the file, e.g., with a Print command in TNLS, with any of the output to printer commands, and with the Jump command in DNLS.

All of the portrayal generators in NLS have at least two sections -- the formatter and the sequence generator; if the user invokes a program of his own, the portrayal generator will have at least one, and possibly two, additional parts -- a user filter program and a user sequence generator.

FORMATTER

The formatter section arranges text passed to it by the sequence generator (described below) in the style specified by the user. The formatter observes viewspecs such as line truncation, length and indenting; it also formats the text in accord with the requirements of the output device.
The formatter works by calling the sequence generator, formatting the text returned, then repeating this process until the sequence generator decides that the sequence has been exhausted or the formatter has filled the desired area (e.g. the display).

SEQUENCE GENERATOR

The sequence generator looks at statements one at a time, beginning at the point specified by the user. It observes viewspecs like level truncation in determining which statements to pass on to the formatter.

For example, the viewspecs may indicate that only the first line of statements in the two highest levels are to be output. The default NLS sequence generator will return pointers only to those statements passing the structural filters; the formatter will further truncate the text to only the first line.

When the sequence generator finds a statement that passes all the viewspec requirements, it returns the statement to the formatter and waits to be called again for the next statement in the sequence.

One of the viewspecs that the sequence generator pays particular attention to is "i" -- the viewspec that indicates whether a user filter is to be applied to the statement. If this viewspec is on, the sequence generator passes control to a user filter program, which looks at the statement and decides whether it should be included in the sequence. If the statement passes the filter (i.e. the user program returns a value of true), the sequence generator sends the statement to the formatter; otherwise, it processes the next statement in the sequence and sends it to the user filter program for verification. (The particular user program chosen as a filter is determined by commands described below.)

USER FILTERS

The user filter program may be either a content analysis pattern (compiled and invoked in the manner described below) or an L10 program which may contain what are essentially content analysis patterns as well as text modification elements which may edit the NLS file automatically.
CONTENT ANALYSIS PATTERNS

Content analysis patterns describe characteristics that a statement must have to be included in the sequence being generated. For example, a content analysis pattern may stipulate that a statement must contain a particular phrase, or that it must have been written since a particular date. In general, content analysis patterns may use any of the pattern matching facilities permitted in L10 FIND statements.

Content analysis patterns cannot affect the format of a statement, nor can they initiate editing operations on a file. They can only determine whether a statement should be viewed at all.

Nevertheless, content analysis filters provide a powerful tool for user control of the portrayal of a series of statements. They are the most frequently used, and easily written, of the user programs. However, if one wishes to change the format of a statement, or to modify the file as it is displayed, he must use a user written L10 program.

USER WRITTEN L10 PROGRAMS

A user written program may be given control by the sequence generator in exactly the same fashion that a content analysis program is initiated. Writing and using such programs effectively requires a thorough knowledge of NLS (content analysis, in particular) and a modicum of exposure to L10.

Such a program may change the format of a statement being displayed and it may modify the statement itself (as well as other statements in the file).
A user written program invoked by the sequence generator has several limitations. It can manipulate only one file and it can look at statements only in the order in which they are presented by the sequence generator. In particular, it cannot back up and re-examine previous statements, nor can it skip ahead to other parts of the file. A user-written sequence generator must be provided when one needs to overcome these restrictions.

USER-WRITTEN SEQUENCE GENERATORS

A user may provide his own sequence generator to be used in lieu of the regular NLS sequence generator. (This is controlled by viewspecs 0 and P.) Such a program may call the normal NLS sequence generator, as well as content analysis filters and user-written L10 programs. It may even call other user-written sequence generators. This technique provides the most powerful means for a user to reformat (and even create) files and to affect their portrayal. However, since writing them requires a detailed knowledge of the entire NLS program, the practice is limited to experienced NLS programmers.
Section 8. INVOCATION OF USER FILTERS AND PROGRAMS

Introduction.

The user-written filters described in this document may be imposed in some cases through the NLS command "Execute Content Analyzer" and in other cases by an NLS subsystem accessed by the command "Goto Programs". The former method is easier but may be used only with simple Content Analyzer patterns. The latter method requires more of the user, furthermore, the several additional capabilities offered by general user-written programs may be invoked only through the "Goto Programs" submode.

User sequence generator programs for more complex editing among many files may be written. Additionally, programs may be written in this L10 subset to be used to generate sort keys in the NLS Sort and Merge commands. Descriptions of these more complicated types of user programs and of NLS procedures which may be accessed by such programs is deferred until a later document. In such examples, however, the user would still make use of the commands in the NLS "Goto Programs" subsystem.

These TNLS commands are used to compile, institute and execute User Programs and filters.

Compilation--

is the process by which a set of instructions in a program is translated from a form understandable by humans (e.g., the L10 language) into a form which the computer can use to execute those instructions.

Institution--

is the process by which a compiled program is linked into the NLS running system for execution.

Execution--

is the process in which the computer carries out the instructions contained in a compiled and instituted program.
This section additionally presents, in detail, examples of the use of the L10 programming language to construct user analyzer filters and reformatters. These programs were written by members of ARC who are not experienced programmers. They do not make use of any constructions not explained in this manual.

**SIMPLE CONTENT ANALYSIS PATTERNS**

The content analysis feature of NLS permits the user to specify a pattern of text content to be matched by statements in NLS files. Only those statements passed to the filter by the sequence generator satisfying the test will be sent to the formatter for display to the user. A simple content analyzer pattern is compiled by the Execute Content Analyzer command or through the Goto Programs submode, and is activated by a Viewspec parameter.

The NLS portrayal Generator, made up of the formatter, the sequence generator, and user filters, is invoked whenever the user requests a new "view" of the file, for example through the use of the TNLS "Print" command or any of the output to printer commands. Thus if one had a user content filter compiled, instituted, and invoked, one could have a printout made (using "Output quickprint", for example) containing only those statements in the file satisfying the pattern. Section 7 (8c) discusses these concepts in detail.

**Syntax of Simple Content Analysis Patterns**

A simple content analyzer pattern is made up of any number of String patterns to be matched terminated by a semi-colon.

```
$strentity ';`
```

It is thus similar to the FIND statement described in Section 6 (7c) of the L10 Primer. It is different because some of the pattern constructions, noted in that section, are neither valid nor relevant out of the context of a complete L10 user program including the constructions which manipulate text pointers.
A pattern may be written as text anywhere in an NLS file. A file may thus contain any number of patterns. However, only one pattern may be instituted (or placed as the active program or pattern) at a time although any number of content analysis patterns may be compiled.

Using commands in the Programs subsystem, one may switch back and forth between the invocation of any of them.

** Execute Content Analyzer

The TNLS command used to compile simple content analysis patterns is:

```
execute content analyzer type in? sp
   ca
   y[es]
   n[o]
```

(if sp, ca, or y[es]) LIT CA

(if n[o]) ADDR CA

In response to the prompt "type in?" the user may respond with SP, CA, or "y" indicating that the pattern will be entered directly from the keyboard. Responding by "n" indicates that the address of the pattern will be specified.

ADDR is a TNLS address specification pointing to the first character in the pattern or non-printing characters immediately preceding the pattern. If the pattern is imbedded in the text of an NLS statement the process will read characters until the first semi-colon is read.

If the semi-colon is omitted in this instance, an error will result.

Thus one may make use of parts of complex patterns by positioning the TNLS current position pointer at an appropriate place in the middle of the pattern text.

If a LIT is specified it is taken to be the text of a Content Analysis pattern. (The semi-colon may be omitted here; it will be appended by the system.)
When this command is given the pattern specified is compiled into the user program buffer, a name is assigned and put on the user program name stack, and it is instituted as a content analyzer program.

When the CA is typed the message "Compiling User Program" will be put out. If the compilation was successful, the user will be left at the TNLS command specification level. If there were any errors in the compilation a list of the places in the pattern in which the error was discovered followed by the message "(number) error(s): Type CA".

The description of the errors may be relatively cryptic. Syntax errors deal with some violation of acceptable language form. Compiler and system errors may relate to some more general (and perhaps more obscure) error in the compiler which the ordinary user cannot easily fix.

Remember that the L10 compiler does not do anything about misspelled words and misplaced punctuation marks.

Content Analysis Via Goto Programs

Simple Content Analysis patterns may also be compiled using a command of the Programs subsystem described below.

Execution and Effect

When applied to a proper pattern the "Execute Content Analyzer" command, in addition to compiling the user's pattern, institutes it as the current content analyzer filter deinstitutioning any existing content analyzer pattern program.

Most users need not be aware of this fact.

Those, however, who may compile more than one content analyzer pattern in a session may wish to switch between them.
To provide a handle on Content Analyzer patterns they are assigned program names made up of the first 5 characters of the pattern preceded by the letters "UP" (for user program), a number referring to the order of compilation, and an exclamation mark (!).

Using this name one may institute and deinstitution patterns as content analyzer filters by using a command in the programs subsystem described below. The patterns will appear under these names in the user program stack which may be examined with the Program Status command.

After compilation and institution a content analyzer pattern may be applied as a filter to any NLS file by using certain viewspecs and any command which causes the Portrayal Generator to examine the file, e.g., the TNLS print commands. Simple content analyzer programs do not modify files. Rather, they just serve as "filters" for the Portrayal Generator (see Section 7 (8c)). Relevant viewspecs are:

i-- show only statements with content which passes the filter. For example an Output Quickprint with viewspec i on would print only those statements passing the filter. If none satisfy the filter test, an "Empty" will be displayed on-line, a blank file will be printed by the Quickprint command.

j-- show all content. This is the default viewspec in NLS. The filter is not used in this case.

k-- show the first statement passing the filter then all others.

Again we emphasize that the files are not modified by simple content analysis filters. L10 user programs must be used for this purpose.

Examples of Simple Content Analysis Patterns

BEFORE (25-JAN-72 12:00);

This pattern will match those statements created or modified (whichever happened most recently) before noon on 25 January 1972.

ID = HGL OR ID = MFA;
This pattern will match all statements created or modified (whichever happened most recently) by users with the identifiers "HGL" or "MFA".

D $LD / ["CA" / "Content Analyzer"];

This pattern will match any of three types of statements: those beginning with a numerical digit followed by two characters which may be either letters or digits, and statements with either the patterns "CA" or "Content Analyzer" anywhere in the statement.

Note the use of the brackets to permit an unanchored search -- a search for a pattern anywhere in the statement. Note also the use of the slash for alternations.

/([2L (SP/TRUE) /2D) D ' - $LD]);

This pattern will match characters in the form of phone numbers anywhere in a statement. Numbers matched may have a two digit alphabetic exchange followed by an optional space (note the use of the TRUE construction to accomplish this) or a numerical exchange.

Examples include YU 4-1234, YU4-1234, and 984-1234.
PROGRAMS SUBSYSTEM

Introduction

This NLS subsystem provides several facilities for the processing of user written programs and filters. It is entered by using the NLS "Goto" (subsystem name) command. This subsystem enables the user to compile L10 user programs as well as Content Analyzer patterns, control how these are arranged internally for different uses, define how programs are used, and interrogate the status of user programs.

Programs subsystem commands

The Goto programs subsystem is entered by the NLS command:

```
g(oto) p(rograms)...
```

After the user types the above the system expects one of the following commands:

Status of User Programs

This sub-command prints out information concerning active user programs and filters which have been compiled and/or instituted. The system may be interrogated about this status with the command:

```
status of user programs) CA
```

when this command is executed the system will print:

-- the names of all the programs in the stack, including those generated for simple content analysis patterns, starting at the bottom of the stack. This stack contains the symbolic names of all compiled programs and a pointer to the corresponding compiled code. The stack is arranged in order of compilation with the most recently compiled program at the head of the stack.
INVOCATION OF USER FILTERS AND PROGRAMS

-- the remaining free space in the buffer. The buffer contains the compiled code for all the current compiled programs. New compiled code is inserted at the first free location in this buffer.

-- the current Content Analyzer program or "None"

-- the current user sequence generator program or "None"

-- the user key program or "None"

Content Analyzer

This command allows the user to specify a content analysis pattern as a content analyzer filter.

c(content analyzer type in?)

SP
CA
y[es]
n[o]

(if SP, CA, or y[es]) LIT CA

(if n[o]) ADDR CA

In response to the prompt "type in?" the user may respond with SP, CA, or "y" indicating that the pattern will be entered directly from the keyboard. Responding by "n" indicates that the address of the pattern will be specified.

ADDR must be the address of the first character or immediately preceding space of the program or pattern.

When this command is executed the pattern specified is compiled into the buffer, its name is put on the stack, and it is instituted as a content analyzer program.

The name assigned is generated in the same manner as those for patterns compiled by the "Execute Content Analyzer" command.
INVOCATION OF USER FILTERS AND PROGRAMS

This command is equivalent to the "Execute Content Analyzer" command in compilation error indications (9b3e) and execution (9b5a).

**L10 Compile**

This command compiles the program specified.

```
L10 compile at ADDR CA
```

ADDR is the address of the first statement of the program.

This command causes the program specified to be compiled into the user program buffer and its name entered into the stack. The program is not instituted.

The name of the program is the visible following the word PROGRAM or FILE in the statement indicated by ADDR.

Errors are indicated as above for the compilation of simple patterns in (9b3e).

The program may be instituted and executed by the appropriate commands.

**Institute Program**

This command enables the user to designate a program as a content analyzer, sequence generator, or key extractor.

```
institute program] PROGNAME CA [CR] NUM
```

[as] CA [content analyzer] CA
c[content analyzer] CA
k[ey extractor] CA
s[quence generator] CA

PROGNAME is the name of a program which had been previously compiled with any of the Execute Content Analyzer, Program L10, or Program Content Analyzer Commands. That is, PROGNAME must be in the stack when this command is executed.
INVOCATION OF USER FILTERS AND PROGRAMS

Instead of PROGNAME the user may specify the program to be instituted by NUM, a numeric value indicating the nth program from the bottom of the stack.  

The program on the bottom of the stack is the program compiled first.

Execute Program

This command transfers control to the specified program.

```
e/execute program/ PROGNAME CA NUM
```

PROGNAME is the name of a program which had been previously compiled. That is, PROGNAME must be in the stack when this command is executed.

Instead of PROGNAME the user may specify the program to be instituted by NUM, a numeric value indicating the nth program in the stack.

Deinstitute Program

This command deactivates the indicated program, but does not remove it from the stack and buffer. It may be reinstituted at any time.

```
de/institute program/ PROGNAME CA NUM
```

PROGNAME is the name of a program which had been previously compiled. That is, PROGNAME must be in the stack when this command is executed.

Instead of PROGNAME the user may specify the program to be instituted by NUM, a numeric value indicating the nth program in the stack.

This assumes one program will not be used for more than one purpose at one time.
Pop Stack

The Pop Stack command deletes the top (or most recent) program on the stack. The program is deinstituted, its name removed from the stack, and its space in the buffer marked as free.

pop stack/ CA

Pop Stack program command (10c2il)

Reset Stack

This command clears all programs from the user program area. All programs are deinstituted, the stack is cleared, and the buffer is marked as empty.

reset stack/ CA
Note on Returning from User Analyzer-Formatter Programs

When a user writes an analyzer-formatter filter program, the main routine must RETURN to the Portrayal Generator. The RETURN must have an argument which is checked by the sequence generator. If the value of that argument is TRUE, the statement will be passed to the formatter to be displayed; if the value is FALSE, it will not be displayed.

The user could thus use FIND statements and expressions to check for the presence of statements to be edited by the string construction elements and either display the edited statement or not, thereby saving the formatting time.

A file could thus be edited quickly without any immediate feedback to the user with the `viewspec` on. However, by turning `viewspec` off afterwards, the user could then see the completely edited file.

Examples of Analyzer-Formatter Programs

The following are examples of user analyzer-formatter programs which selectively edit statements in an NLS file on the basis of text searched for by the pattern matching capabilities. Examples of more sophisticated user programs such as sort keys and user sequence generator programs will be presented in a later supplement with a description of NLS routines easily accessed by users.

Example 1--

PROGRAM outname % removes statement names -- del= ()

DECLARE TEXT POINTER sf, daf, pae;
(outname)PROCEDURE;
  IF FIND tsf $NP '( $paf [']$) $pae THEN
  BEGIN
    ST sf + pae SE(sf);
    RETURN(TRUE);
  END;
  ELSE RETURN(FALSE);
  END.
FINISH
This program removes the text and delimiters of NLS statement names from the beginning of the statements.

Example 2--

```l10
PROGRAM changed;
(changed) PROCEDURE;
    LOCAL TEXT POINTER f, e;
    FIND * SE(f) te;
    IF FIND SINCE (25-JAN-72 12:00) THEN
        BEGIN
            ST f + "(CHANGED)", f e;
            RETURN(TRUE);
        END
        ELSE RETURN(FALSE);
    END
FINISH
```

This program checks to see if a statement was written after a certain date. If it was, the string "(CHANGED)" will be put at the front of the statement.
INDEX

A (5c4k)
ALT (5d6d5)

analyzer-formatter programs, examples of (10ch4)
AND (5cla3), (8c3alb)
argument lists (5d3b2)
arithmetic operators (5cl4a)
array variables, declaring (6b3a)
assignment statement (7ala)
assignments (5d4a)

BC (5d6d9)
BEFORE datim (8c3a2al11)

BEGIN (7cla)

BETWEEN pos pos ( strenity ) (8c3a2b2)

binding precedence (5e2a)
BLOCK construction (7cl)

body, program (4blb)

BW (5d6d10)
G. (5d6d11)
CA (5d6d12)
CASE expression (5e3a)
CASE statement (7d3a)
CCPOS (5d2j), (5d10a), (8b)
CD (5d6d13)
CH (5d7a1)
char (8c3a2a1a2)
character classes (5d7)
charclass (8c3a2a1a3)
CHR (3ble2e)
comments, def. (3c9)
compilation (10a2a)
Compile program command (10c2e1)
conditional expressions (5e3)
conditional statements (7d1)
constant, def. (3c7)
content analysis
and Goto Programs (10b1a)
-formatter programs, examples of (10q1a)
-formatter programs, returning from (10c3a)
goto programs command (10c2d1)
patterns (8c2), (10b1)
CR (5d6d6)
current character position (8b)

d (5d7a9)
declarations (6a1)
  global (6bl)
  local (6dl)
  procedure level (6a3)
  program level (6a3)
  reference (6cl)
DECLARE STRING statement (6b5a)
DECLARE TEXT POINTER statement (6b6a)
declaring
  array variables (6b3a)
  multiple variables (6b4a)
  scalar variables (6b2a)
  string variables (6b5a)
  text pointers (6b6a)
Deinstitute Program command (10c2h1)
Divide statement (7bl)

END (7cla)
ENDCASE statement (7d3a1)
ENDCHR (5d6d2)
EOL (5d6d4)

examples of analyzer-formatter programs (10c1a)

Execute Content Analyzer command (10b3a)

Execute program command (10c2g1)

eexecute, def. (3c12)

eexecution (10a2c)

eexpression, def. (3c10)

eexpressions (5e1a)

    conditional (5e3)

    FIND (8c1)

FALSE (5d6a3)

filters (9c5a)

FIND (5d2k)

FIND Expressions and Patterns (8c1)

FIND Statements (8c1)

FINISH statement (1b1c)

formatter (9c3a)

global, def. (3c5)

    declarations (6bl)

    variable (5b2)

Goto Programs subsystem (10c1a)

    and content analysis (10b1a)
commands (10c2a)

header, program (hbl1a)

heirarchy of operations (5e2a)

i viewspec (9chc), (10b5bl)
ID (3ble2a)
ID (=/=) UID (8c3a2a1a9)
identifier, def. (3c1)
IF expressions (5e3a)
IF statement (7d2a)
IN (5c3a1)
indexed variable, def. (3c4)
indexing stringnames (8b6a)
Institute Program command (10c2fl)
institution (10a2b)
interval operators (5c3a)

j viewspec (10b5b2)

k viewspec (10b5b3)
INDEX

L10

Compile command (l0c2el)
declarations (6a1)
programs, user-written (9c5a2a)
syntax (3bl)

label, def. (3c2)
LD (5d7a4)
LF (5d6d7)
literal (5d6a)
LL (5d7a7)
LLD (5d7a3)
LOCAL declaration (6d1)
local variable (5b3)
local, def. (3c6)
logical operators (8c3a1), (5c1a)

MAX (5d2h), (5d8)
MIN (5d2h), (5d8)
MOD (5c4h)
multiassignment statement (7a2a)
NLD (5d7a5)
NLS Portrayal Generator (10b1a)
NOT (5cla4), (8c3a1c)
NP (5d7al1)
NUM (3ble2b), (5d6al)
NUM argument (8c3a2a1a7)
NUM1 & NUM2 argument (8c3a2a1a8)

O viewspec (9c6a)
opertions, hierarchy of (5e2a)
operators (5c)
arithmetic (5c4a)
interval (5c3a)
logical (5cla)
relational (5c2a)
OR (5c1A2), (8c3al4)
OUT (5c3A2)

P viewspec (9c6a)
pattern matching arguments (8c3a2)
patterns (8c)
patterns,
content analysis (8c2), (9c5ala), (10bl)
syntax of content analysis (10b2a)

pointers (5d5a)

Pop Stack command (10c2il)

portrayal generator (9cl)

POS (5d21), (8c3a3b1)

posrel (5d1la)

primitives (5d1)

procedure call, as primitive (5d31)

program

  compilation (10a2a)

  compile command (10c2el)

  deinstitute command (10c2hl)

  execute command (10c2gl)

  execution (10a2c)

  institute command (10c2fl)

  institution (10a2b)

  pop stack command (10c2il)

  reset stack command (10c2jl)

  structure (4a1), (4b1)

programs,

  creating (9bl)

  examples of (10ch4a)

  returning from (10c3a)

  status command (10c2cl)
subsystem (10c1a)
subsystem commands (10c2a)
user filter (9c5a)
user-written (9c5a2a)

PT (5d7a10)

READC (5d2i), (5d9a)
REF statement (6c1)
reference declarations (6c1)
referenced variable (5b14)
relational operators (5c2a)
Reset Stack program command (10c2j1)
returning from user analyzer-formatter programs (10c3a)

SAB (5D6D8)
scalar variables, declaring (6b2a)
SE (8b2c)
sequence generator (9c1a)
sequence generator, user-written (9c6a)
SF (8b2b)
SINCE datim (8c3a2a1a10)
SP (5D6D3)
SR (3B1E2C), (8c3a2a1a1)
SR1 (3B1E2D), (5D6D1)
statement, def. (3cl1)
statements, FIND (8c1)
Status of User Programs command (10c2c1)
string entities (8c3)
string
construction (8d)
def. (3c8)
end (8b2c)
expressions (5e4a)
front (8b2b)
pattern entities (8c3)
patterns (8c)
test and manipulation (8)
variables, declaring (6b5a)
syntax (3bl)
content analysis patterns (10b2a)
program structure (4al)
terminal statement, program (4b1c)
text pointer (5b6), (8b)
comparisons (5d11)
declaring (6b6a)
Tree Meta (3a1)
TRUE (5d6a2), (8c3a3a3)
UL (5d7a6)
ULD (5d7a2)
unreferenced variable (5b5)
unreferencing (6c2)
user analyzer-formatter programs, returning from (10c3a)
user filters (9c5a)
user programs (9bl)
user programs status command (10c2cl)
user-written L10 program (9c5a2a)
user-written sequence generators (9c6a)

V (5ch1)
variables (5bl)
  def. (3c3)
  declaring multiple (6b4a)

viewspec
  i (9c4c), (10b5bl)
  j (10b5b2)
  k (10b5b3)
  o (9c6a)
  p (9c6a)
(6c2)
(strength) (8c3a2a1a4)
* stringname * (8c3a2b1)
- parameter (8c3a2a1a5)

.A (5c4k)
.V (5c4i)
.X (5c4j)
/ (8c3a1d)
< (8c3a3a1)
> (8c3a3a2)
/ (strength) (8c3a2a1a6)
† ID (8c3a3b2)
+ [NUM] ID (8c3a3b3)