< USERGUIDES, PROGRAMMERS-GUIDE.NLS;5, >, 3-Sep-79 10:34 LEN
;;;;Version with directives located in <JOURNAL,34210,>
Send requests for printed copies to <FEEDBACK>.
TABLE OF CONTENTS

INTRODUCTION

NLS provides a variety of commands for file manipulation and viewing. Editing commands allow the user to insert and change the text in a file. Viewing commands (viewspecs) allow the user to control how the system prints or displays the file. Line truncation and control of statement numbers are examples of these viewing facilities.

Occasionally one may need more sophisticated view controls than those available with the viewing features of 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, or build commands for specific applications.

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

User-written programs currently must be coded in ARC's procedure-oriented programming language, L10. NLS itself is coded in L10. L10 is a high-level language which must be compiled into machine-readable instructions. This document describes L10.

Programs which interact with users additionally use a language developed at ARC called Command Meta Language (CML), described in Part Four of this document.

This document describes three general types of programs:

--simple filters that control what is portrayed on the user's teletype or display (Parts One and Two),

--programs that may modify the statements as they decice whether

to print them (Parts Two and Three),
--those that, like commands, are explicitly given control of the job and interact with the user (Part Four).
User programs that control what material is portrayed take effect when NLS presents a sequence of statements in response to a command like Print (or Jump in DNLS).

In processing such a command, NLS looks at a sequence of statements, examining each statement to see if it satisfies the viewspecs then in force. At this point NLS may 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, NLS just goes on to the next statement.

While the program is examining the statement to decide whether or not to print it, it may modify the contents of the statement.
Such a program can do anything the user can do with NLS commands.
For more complex tasks, a user program function as a special-purpose subsystem having (in addition to the may supervisor commands) one or more commands. Once such a program is loaded, it can be used just like any of the standard subsystems. (The MESSAGE program is an example.)

This document is divided into five parts:

Part One is intended for the general user.

It is a primer on Content Analyzer Patterns, allowing the NLS user to set up simple yet powerful filters whrough which he may view and edit files. This does not involve learning the L10 language nor programming. This section can stand alone, and the general (if somewhat experienced) NLS user should find it very useful.

Part Two is intended for the beginning programmer.

It presents a hasty overview of L10 programming, with enough tools to write simple programs. This is intended as an introduction for the beginning user programmer, who we assume is reasonably familiar with NLS (its commands, subsystems, and capabilities) and has some aptitude for programming.

Part Three is a more complete presentation of L10.

It is intended to acquaint a potential L10 programmer with enough of the language and NLS environment to satisfy most requirements for automated editing programs. Many of the concepts in Part Two are repeated in Part Three so that it may stand alone as an intermediate programmer's reference guide. This is the section in which to begin looking for answers to specific questions.

Part Four presents more advanced L10 tools and an introduction to CML, allowing command syntax specification.

This should give the programmer the ability to write programs which work across files, which move through files in other than the standard sequential order, and which interact with the user. It allows the programmer to build user-attachable subsystems with commands looking very much like standard NLS facilities.

Part Five presents a number of subjects of interest to the advanced L10 progammer.

We suggest that those who are new to L10 begin by acquiring a thorough understanding of Part One. Then Part Two should be

studied one section at a time, pausing between sections to try out the concepts presented by actually writing patterns or programs that put the new ideas to experimental use. Actual experience is of at least as much value as this tutorial. Tutorial guidance should be requested from ARC through your architect. If you have problems at any point, you should get help from ARC before proceeding to the next section.

Note: For syntactical correctness, some examples include constructs not yet defined in the text; they will be discussed soon thereafter.

For examples of user programs which serve a variety of needs, examine the attachable subsystems in the <PROGRAMS> directory and their descriptions in Help. For information about commands mentioned, ask for the programming subsystem with the NLS Help command. NDM

PART ONE: Content Analyzer Patterns

Section 1: Introduction

Content analysis patterns cannot affect the format in which a statement is printed, nor can they edit a file. They can only determine whether a statement should be printed at all. They are, in a sense, a filter through which you may view the file. More complex tasks can be accomplished through programs, as described later in this document.

The Content Analyzer filter is created by typing in (or selecting from the text in a file) a string of a special form which describes those statements which will pass through the filter. This string is called the "Content Analyzer Pattern". Each statement is checked against the pattern before it is printed; only statements that are described by the pattern will be printed.

Some quick examples of Content Analyzer Patterns:

'($LD ') will show all statements whose first character is an open parenthesis, then any number of letters or digits, then a close parenthesis.

["blap"] will show all statements with the string "blap" somewhere in them.

SINCE (3-JUN-75 00:00) will show all statements edited since June 3, 1975

The next part of this section will describe the elements which make up Content Analyzer Patterns, followed by some examples. The final subject of this section is how to put them to use.

Section 2: Patterns

Elements of Content Analyzer Patterns

Content Analyzer Patterns describe certain things the system must check before printing a statement. It may check one or a series of things. Each test is called an element; the many possible elements will be described below.

The Content Analyzer searches a statement from the beginning, character by character, for described elements. As it encounters each element of the pattern, the Content Analyzer checks the statement for the occurrence of that element; if the test fails, the whole statement is failed (unless there was an "or" condition, as described later) and not printed; if the test is passed, an imaginary marker moves on to the next character in the statement, and the next test in the pattern is considered.

For example, if the next element in the Content Analyzer pattern is "LD", the imaginary marker will move over the next character and go on to test the next element of the pattern only if the next character is a letter or a digit; otherwise the whole statement fails to pass the filter.

The pattern may include any sequence of the following elements; the Content Analyzer moves the marker through the statement checking for each element of the Pattern in turn:

Literal String elements

'c -- the given character (e.g. a lower case c)

"string" -- the given string (may include non-printing characters, such as spaces)

Character class elements

CH -- any character

L -- lowercase or uppercase letter

D -- digit

UL -- uppercase letter

LL -- lowercase letter

ULD -- uppercase letter, or digit

LLD -- lowercase letter, or digit

LD -- lowercase or uppercase letter, or digit

NLD -- not a letter nor digit

PT -- any printing character (letters, digits, punctuation)

NP -- any non-printing character (e.g. spaces, control characters)

Special non-printing character elements

SP -- a space

TAB -- tab character

CR -- a carriage return

LF -- line feed character

EOL -- TENEX EOL (end of line) character

ALT -- altmode character

Special elements

ENDCHR -- beginning and end of every NLS statement; can't scan past it; not considered a character

TRUE -- is true without checking anything in statement (used with OR constructs, as described below)

ID= id -- statement created by user whose ident is given

ID# id -- statement not created by user whose ident is given

BEFORE (d-t) -- statement edited before given date and time

SINCE (d-t) -- statement edited since given date and time

E.g. BEFORE (1 OCT 1974 00:00) ;

The date and time must both appear in the parentheses. It accepts almost any reasonable date and time syntax.

Examples of valid dates:

17-APR-74	17 APRIL 74
APR-17-74	17/5/1974
APR 17 74	5/17/74
APRIL 17, 1974	

Examples of valid times:

1:12:13	1234:56
1234	1:56AM
1:56-EST	1200NOON
16:30 (i.e. 4:30 PM)	

12:00:00AM (i.e. midnight)
11:59:59AM-EST (i.e. late morning)
12:00:01AM (i.e. early morning)

Scan direction

< -- set scan direction to the left

> -- set scan direction to the right

The default, re-initialized for each new statement, is scan to the right from before the first character in the statement (beginning to end).

Modifying Elements

Several operators can modify any of the elements except the "special elements":

NUMBER -- multiple occurrences

A number preceding any element other than one of the "Special elements" means that the test will succeed only if it finds exactly that many occurrences of the element. If there aren't that many, the statement will be rejected. Even though there may be more, it will stop after that many and go on to check the next element in the pattern.

3UL means three upper case letters

$ -- range of occurrences

A dollar sign ($) preceding any element other than the "Special elements" means "any number of occurrences of". This may include zero occurrences. It is good practice to put the element itself in parentheses.

$(*-) means any number of dashes

A number in front of the dollar sign sets a lower limit.

3$(D) means three or more digits

A number after the dollar sign sets an upper limit for the search. It will stop after that number and then check for the next element in the pattern, even if it could have found more.

$3(LD) means from zero to three letters or digits

5$7(PT) means from 5 to 7 (inclusive) printing characters

[] -- floating scan

To do other than a character by character check, you may enclose an element or series of elements in square brackets []. The Content Analyzer will scan a statement until the element(s) is found. (If the element is not in square brackets, the whole statement fails if the very next character or string fails the test of the next element.) This test will reject the statement if it can't find the element anywhere in the statement. If it succeeds, it will leave the marker for the next test just after the string satisfying the contents of the square brackets.

"start" means check to see if the next five characters are: s t a r t.

["start"] means scan until it finds the string: s t a r t.

[3D] means scan until it finds three digits.

[3D *:] means scan until it finds three digits followed by a colon

- -- negation

If an element is preceded by a minus sign -, the statement will pass that test if the element does not occur.

-LD means anything other than a letter or digit, such as punctuation, invisibles, etc.

NOT -- negation

NOT will be TRUE if the element or group of elements enclosed in parentheses following the NOT is false.

NOT LD will pass if the next character is neither a letter nor a digit.

Combining Elements

You may put together any number of any of these elements to form a pattern. They may be combined in any order. Spaces within the pattern are ignored (except in literal strings) so they may be used to make reading easier for you.

e.g. 1$PT [".NLS;" 1$D] -SP

i.e. one or more printing characters, then scan for .NLS; followed by one or more digits, then check that the next character is not a space

More sophisticated patterns can by written by using the Boolean logical expression features of L10. Combinations of elements may in turn be treated as single elements, to be modified or combined using logical operators.

Generally, an expression is executed left to right. The following operations are done in the given order:

( )
/
NOT
AND
OR

( )

Parentheses (and square brackets for floating scans) may be used to group elements. It is good practice to use parenthesis liberally.

/

/ means "either or"; the bracketed element, consisting of two or more alternatives, will be true if either (any) element is true.

(3D L / 4D) means either three digits and a letter or four digits.

Since the slash is executed before NOT, NOT D / 'h will be true if the next character is NEITHER a digit nor the letter "h". It is the same as NOT (D/'h).

Sometimes you may want want the scan to pass your marker over something if it happens to be there (an optional element). "TRUE" is true without testing the statement.

If the other tests fail, the imaginary marker is not moved.

(D / TRUE) looks for a digit and passes the imaginary marker over it. If the next character is not a digit, it will just go on to the next test element in the pattern without moving the marker and without failing the test. (This test always passes.)

i.e. It is used to scan past something(s) which may or may not be there.

Since expressions are executed from left to right, it does no good to have TRUE as the first option. (If it is first, the test will immediately pass without trying to scan over any elements.)

AND

AND means both of the two separated groups of elements must be true for the statement to pass.
SINCE (3/6/73 00:00) AND ID#NDM means statements written since March 6, 1973 by someone other than NDM.

OR

OR means the test will be true if either of the separated elements is true. It does the same thing as slash, but after "AND" and "NOT" have been executed, allowing greater flexibility.
D AND LLD OR UL means the same as (D AND LLD) OR UL
D AND LLD / UL means the same as D AND (LLD / UL)
While such patterns are correct and succinct, parentheses make for much clearer patterns. Elements within parentheses are taken as a group; the group will be true only if the statement passes all the requirements of the group. It is a good idea to use parentheses whenever there might be any ambiguity.

Section 3: Examples of Content Analyzer Patterns

D 2$LD / ["CA"] / ["Content Analyzer"]
This pattern will match and pass any of three types of NLS statements: those beginning with a numerical digit followed by at least two characters which may be either letters or digits, or statements with either of the strings "CA" or "Content Analyzer" anywhere in the statement.
Note the use of the square brackets to permit a floating scan -- a search for a pattern anywhere in the statement. Note also the use of the slash for alternatives.

BEFORE (25-JAN-72 12:00)
This pattern will match those statements created or modified before noon on 25 January 1972.

(ID = HGL) OR (ID = NDM)
This pattern will match all statements created or modified by users with the identifiers "HGL" or "NDM".

[(2L (SP/TRUE) / 2D) D '- 4D]
This pattern will match characters in the form of phone numbers anywhere in a statement. Numbers matched may have an alphabetic exchange followed by an optional space (note the use of the TRUE construction to accomplish this) or a numerical exchange.
Examples include DA 6-6200, DA6-6200, and 326-6200.

[ENDCHR] < "cba"
This will pass those statements ending with "abc". It will go to the end of the statement, change the scan direction to left, and check for the characters "cba". Note that since you are scanning backwards, to find "abc" you must look for "cba". Since the "cba" is not enclosed in square brackets, it must be the very last characters in the statement.

Section 4: Using the Content Analyzer

Content Analyzer Patterns may be entered in two ways:
1) From the BASE subsystem, use the command:
Set Content (pattern) To PATTERN OK
2) From the PROGRAMS subsystem, use the command:
Compile Content (pattern) PATTERN OK
OK means "Command Accept", a control-D or, in TNLS (by default) a carriage return.

In either case:

1) Patterns may be typed in from the keyboard, or
2) they may be text in a file.
In this case, the pattern will be read from the first character addressed and continue until it finds a semicolon (;) so you must put a semicolon at the end of the pattern (in the file).

Viewspec j must be on (i.e. Content Analyzer off) when entering a pattern.

Entering a Content Analyzer Pattern does two things:
1) compiles a small user program from the characters in the pattern, and
2) takes that program and "institutes" it as the current Content Analyzer filter program, deinstituting any previous pattern.

"Instituting" a program means selecting it as the one to take effect when the Content Analyzer is turned on. You may have more than one program compiled but only one instituted.

When a pattern is deinstituted, it still exists in your program buffer space and may be instituted again at any time with the command in the PROGRAMS subsystem:

Institute Program PROGRAM-NAME (as) Content (analyzer) OK

The programs may be refered to by number instead of name. They are numbered sequentially, the first entered being number 1.

All the programs you have compiled and the one you have instituted may be listed with the command in the PROGRAMS subsystem:

Show Status (of programs buffer) OK

Programs may build up in your program buffer. To clear the program buffer, use the PROGRAMS subsystem command:

Delete All (programs in buffer) OK

We recommend that you do this before each new pattern, unless you specifically want to preserve previous patterns.

To invoke the Content Analyzer:

When viewspec i is on, the instituted Content Analyzer program (if any) will check every statement before it is printed (or displayed).

If a statement does not pass all of the requirements of the Content Analyzer program, it will not be printed.

In DNLS, if no statements from the top of the screen onward through the file pass the Content Analyzer filter, the word "Empty" will be displayed.

Note: You will not see the normal structure since one statement may pass the Content Analyzer although its source does not. Viewspec m (statement numbers on) will help you determine the position of the statement in the file.

When viewspec k is on, the instituted Content Analyzer filter will check until it finds one statement that passes the requirements of the pattern. Then, the rest of the output (branch, plex, display screen, etc.) will be printed without checking the Content Analyzer.

When viewspec j is on, no Content Analyzer searching is done. This is the default state; every statement in the output

(branch, plex, display screen, etc.) will be printed. Note that i, j, and k are mutually exclusive.

Notes on the use of Content Analyzer filters:

Some NLS commands are always affected by the current viewspecs (including i,j, or k):

Output
Jump (in DNLS)
Print (in TNLS)

Most NLS commands ignore the Content Analyzer in their editing. The following BASE subsystem commands offer the option of specifying viewspecs, or "Filters", (which may turn on the Content Analyzer) which apply only for the purpose of that one command and affect what statements the command works on (only those statements which pass the filter will be copied, moved, etc.; structure will be adjusted):

Copy
Delete
Move
Substitute

At this point, it would be wise to practice until you become proficient at Content Analyzer patterns. You might begin by trying to use some of the patterns given in the above examples, and then try writing a few patterns of your own. These patterns are both a useful NLS tool and a basic component of many L10 programs. We further recommend that you contact ARC via your architect before you begin the next part.

PART TWO: Introduction to L10 Programming

Section 1: Content Analyzer Programs

Introduction

When you specify a Content Analyzer Pattern, the PROGRAMS subsystem constructs a program which looks for the pattern in each statement and only displays the statement if the pattern matching succeeds. You can gain more control and do more things if you build the program yourself. The program will be used just like the simple pattern program and has many of the same limitations. Programs are written in NLS just like any other text file. They then can be converted to executable code by a compiler. This code resides (or is loaded) in your programs buffer space; it can be instituted as the current Content Analyzer filter program like a Content Analyzer Pattern.

Program Structure

If you specify a Content Analyzer Pattern, NLS compiles a small program that looks like this (with the word "pattern" standing for whatever you typed in):

```
PROGRAM name
   (name) PROCEDURE;
      IF FIND pattern THEN RETURN(TRUE) ELSE RETURN(FALSE);
      END.
   FINISH
```

L10 programs must begin with a header statement, the word PROGRAM (all caps) followed by the name of the first procedure to be executed (all lower-case). This name is also the name of the program. If the program is being compiled into a file (to be described at the end of this section), the word FILE should be substituted for the word PROGRAM. E.g.

```
PROGRAM first
    or
FILE deldir
```

(Note: the Content Analyzer compiler makes up a program name consisting of UP#!xxxxx, where
is a sequential number, the first pattern being number one, and
xxxxx is the first five characters of your pattern.)
E.g. UP1!$LDEP

The body of a program consists of a series of DECLARATION statements and PROCEDURES (in any order) which are blocks of instructions. In the above case, the program consisted of only one small procedure and no declarations. When the program is loaded into your programs buffer space, the declarations reserve space in the system to store information (variables). When the program is used as a Content Analyzer filter program, the first procedure is called for each statement. It may in turn call other procedures and access variables in the program or in the NLS system. E.g.

```
DECLARE x, y, z ;   (described below)
(first) PROCEDURE ;
   ...
```

The end of the program is delimited by the word "FINISH" (in all upper case). The compiler stops at that point, so any text after that in the NLS source file will be ignored. Comments may be enclosed in percent signs (%) anywhere in the program, even in the middle of L10 statements. The L10 compiler will ignore them.
Except within literal strings, variable names and special L10 words, spaces are ignored. It is good practice to use them liberally so that your program will be easy to read. Also, NLS file structure is ignored; statements will be read sequentially, regardless of their level. Structure is, however, very valuable in making the program readable, and it is good practice to use it in close correlation to the program's logical structure. For instance, the programmer usually makes each of the elements of a program (declarations, procedures, and FINISH) separate statements, below the header statement in file structure. This point will be discussed further later.
So far, we have file which looks something like:

```
PROGRAM name1
   DECLARE ... ;
   DECLARE ... ;
   (name1) PROCEDURE ;
   (name2) PROCEDURE ;
   FINISH
```

Procedure Structure

Each procedure must begin with its header statement. This header statement is a name enclosed in parentheses followed by the word PROCEDURE, and terminated by a semicolon. E.g.

```
(name) PROCEDURE ;
```

The body of the procedure may consist of Local declarations, then L10 statements. An L10 statement is any program instruction, terminated by a semicolon. The body must at some point return control to the procedure that called it. All

this will be further discussed later.
The procedure must end with the terminal statement:

```
END.
```

Example (the actual L10 statements in this example will become clear as you read on):

```
PROGRAM compare    % Content analyzer.  Displays statement if
first two visibles are the same.  %
   %reserve space for ("declare") four text pointers named
   "pt1" through "pt4"%
      DECLARE TEXT POINTER pt1, pt2, pt3, pt4;
   %reserve 100 characters of space for each of two string
   variables named "vis1" and "vis2",%
      DECLARE STRING vis1[100], vis2[100];
   (compare) PROCEDURE ;
      %if find two visibles, set pointers around first two
      visibles (strings of printing characters)%
         IF FIND $NP ^pt1 1$PT ^pt2 $NP ^pt3 1$PT ^pt4 THEN
            BEGIN
            %put visibles in strings%
               *vis1* _ pt1 pt2 ;
               *vis2* _ pt3 pt4 ;
            %compare contents of strings, return and display
            the statement if identical%
               IF *vis1* = *vis2* THEN RETURN(TRUE);
            END;
      %otherwise, return and don't display%
         RETURN (FALSE) ;
      END.
   FINISH
```

Declaration Statements

As you may have guessed from the above example, Content Analyzer programs can manipulate variables (like text pointers and strings), while patterns cannot.

Text Pointers

A text pointer points to a particular location within an NLS statement (or into a string, as described later).

The text pointer points between two characters in a statement. By putting the pointers between characters, a single pointer can be used to mark both the end of one string and the beginning of the string starting with the next character.

Text pointers are declared with the following Declaration statement:

```
DECLARE TEXT POINTER name ;
```

Strings

String variables hold text. When they are declared, the maximum number of characters is set.

To declare a string:

```
DECLARE STRING name[num] ;
```

num is the maximum number of characters allowed for the string.

E.g.

```
DECLARE STRING lstring[100];
```

declares a string named "lstring" with a maximum length of 100 characters and a current length of 0 characters (it's empty).

You can refer to the contents of a string variable by surrounding the name with asterisks. E.g.

lstring is the string stored in the variable named "lstring".

(Refering to lstring without the asterisks represents only the first computer word of the string. This is rarely needed.)

You can put the text between two text pointers in a string variable with the L10 statement:

lstring _ ptr1 ptr2 ;

where ptr1 and ptr2 are the names of previously declared and set text pointers, and lstring is a previously declared string variable.

These variables will retain their value from one statement to the next. Other types of variables and their use will be discussed in detail in Part Three, Section 3.

Body of the Procedure

RETURN Statement

No matter what it does, every procedure must return control to the procedure that called it. The statement which does this is the RETURN statement. E.g.

RETURN ;

A RETURN statement may pass values to the procedure that called it. The values must be enclosed in parentheses after the word RETURN. E.g.

RETURN (1,23,47) ;

A Content Analyzer program must return either a value of TRUE or of FALSE. If it returns the value TRUE (1), the statement will be printed; if it returns FALSE (0), the statement will not be printed. I.e.

RETURN (TRUE); will print the statement

RETURN (FALSE); will not print the statement

The RETURN statement often is at the end of a procedure, but it need not be. For example, in the middle of the procedure you may want to either RETURN or go on depending on the result of a test.

Other than the requirement of a RETURN statement, the body of the procedure is entirely a function of the purpose of the procedure. A few of the many possible statements will be described here; others will be introduced in Part Three of this document.

FIND Statement

One of the most useful statements for Content Analyzer programs is the FIND statement. The FIND statement specifies a Content Analyzer pattern to be tested against the statement, and text pointers to be manipulated and set, starting from the Current Character Position (that invisible marker refered to in Section 1). If the test succeeds, the character position is moved past the last character read. If at any point the test fails, the character position is left at the position prior to the FIND statement. The values of text pointers set in the statement prior to the failing element will remain as set; others of course will not be changed.

FIND pattern ;

The Current Character Position is initialized to BEFORE THE

FIRST CHARACTER, and the scan direction is initialized to Left to RIGHT, FOR EACH NEW STATEMENT passed to the Content Analyzer program.
Any simple Content Analyzer pattern (as describe above) is valid in a FIND statement.
In addition, the following elements can be incorporated in the pattern:

stringname
the contents of the string variable

^ptr
store current scan position into the text pointer specified by ptr, the name of a declared text pointer

_NUM ptr
back up the specified text pointer by the specified number (NUM) of characters. If NUM is not specified, 1 will be assumed. Backup is in the direction opposite to the current scan direction.

ptr
Set current character position to this position. ptr is the name of a previously set text pointer.

SF(ptr)
The Current Character Position is set to the front of the statement in which the text pointer ptr is set and scan direction is set from left to right.

SE(ptr)
The Current Character Position is set to the end of the statement in which the text pointer ptr is set and scan direction is set from right to left.

BETWEEN ptr1 ptr2 (pattern)
Search limited to between positions specified. ptr is a previously set text pointer; the two must be in the same statement or string. Current Character Position is set to first position before the pattern is tested. E.g.

```
BETWEEN pt1 pt2 (2D [.] $NP)
```

FINDs may be used as expressions as well as free-standing statements. 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 any one of the elements is false. E.g.

```
IF FIND pattern THEN ... ;
```

Complicated example:

```
IF FIND ^sf $NP *( $(LD/*-) *) [*.  * *str*] SE(sf) $NP
*. THEN RETURN(TRUE) ELSE RETURN(FALSE);
```

IF Statement

IF causes execution of a statement if a tested expression is TRUE. If it is FALSE and the optional ELSE part is present, the statement following the ELSE is executed. Control then passes to the statement immediately following the IF statement.

```
IF testexp THEN statement ;
IF testexp THEN statement1 ELSE statement2 ;
```

The statements within the IF statement can be any valid L10 statement, but are not followed by the usual semicolon; the whole IF statement is one L10 statement and is followed by a semicolon.

E.g.

```
IF FIND [5D] THEN RETURN(FALSE) ELSE RETURN(TRUE) ;
```

Programming Style: File Structure

The compiler which converts your NLS text to code ignores NLS file structure. This allows you to use structure to make your program text easier to read and understand. Logical use of structure often facilitates the actual programming task as well. Some conventions have developed at ARC in this respect, although flexibility is essential. These should seem obvious and logical to you.

All declarations and PROCEDURE statements should be one level below the PROGRAM statement.

All local declarations (not yet described) and code should be one level below the PROCEDURE statement.

It is good style, and makes for much easier programming, to list what you want to do as comment statements (in percent signs) at the level below the PROCEDURE statement. Then you can go back and fill in the code that accomplishes the task described in each comment statement. The code should go one level below the comment.

It is also worthwhile to put comments in individual statements whose purpose is not obvious.

We will later describe how to block a series of statements where one is required. These blocks should go a level below the statement of which they are a part.

File structure should follow the logical structure of the program as closely as possible. E.g.

```
IF FIND [5D]
   THEN RETURN(TRUE)
   ELSE RETURN(FALSE);
```

Using Content Analyzer Programs

Once the Content Analyzer program has been written (in an NLS file), there are two steps in using it. First, the program must be "compiled," i.e. translated into machine-readable code; the compiled code is "loaded" into a space reserved for user programs (the user programs buffer). Secondly, the loaded program must be "instituted" as the current Content Analyzer program.

There are two ways to compile and load a program:

1) You may compile a program and load it into your programs buffer all in one operation. In this case, the program header statement must have the word PROGRAM in it. When the user resets his job or logs off, the compiled code will disappear.

First, enter the Programs subsystem with the command:

```
Goto Programs OK
```

Then you may compile the program with the command:

```
Compile L10 (user program at) SOURCE OK
```

SOURCE is the NLS file address of the PROGRAM statement.

2) You may compile a program into a TENEX code file and then load it into your buffer in a separate operation. The program can then be loaded from the file into your user programs buffer at any time without recompiling. The header statement must use the word FILE instead of PROGRAM. Use the PROGRAMS subsystem command:

Compile File (at) SOURCE (using) L10 (to file) FILENAME OK
The FILENAME must be the same as the program's name. The code file is called a REL (RELocatable code) file. Whenever you wish to load the program code into the user programs buffer, use the PROGRAMS subsystem command:
Load Program (file) FILENAME OK

Once a compiled program has been loaded (by either route), it must be instituted. This is done with the PROGRAMS subsystem command:

Institute Program PROGRAM-NAME (as) Content (analyzer program) OK
The named program will be instituted as the current Content Analyzer filter, and any previously instituted program will be deinstituted (but will remain in the buffer).
Again, the programs in the buffer are numbered, the first in being number one. You may use the number instead of the program's name as a shorthand for PROGRAM-NAME.

To invoke the Content Analyzer using whatever program is currently instituted, use the viewspec i, j, or k, as described in Part One, Section 4 (3d4).

Problems

Given these few constructs, you should now be able to write a number of useful Content Analyzer programs. Try programming the following:

1) Show those statements which have a number somewhere in the first 20 characters.
2) Show those statements where the first visible in the statement is repeated somewhere in the statement.

Sample solutions:

Problem 1

```
PROGRAM number
   DECLARE TEXT POINTER ptr1, ptr2 ;
   (number) PROCEDURE ;
      FIND ^ptr1 $20CH ^ptr2 ;
      IF FIND BETWEEN ptr1 ptr2 ( [D] )
         THEN RETURN(TRUE)
         ELSE RETURN(FALSE);
      END.
   FINISH
```

Alternate Solution to Problem 1: Content Analyzer Filter

```
$20CH < [D]
```

Problem 2

```
PROGRAM vis
   DECLARE TEXT POINTER ptr1, ptr2 ;
   DECLARE STRING str[500] ;
   (vis) PROCEDURE ;
      FIND $NP ^ptr1 1$PT ^ptr2 ;
      *str* _ ptr1 ptr2 ;
      IF FIND ptr2 [NP *str* NP]
         THEN RETURN(TRUE)
         ELSE RETURN(FALSE);
      END.
   FINISH
```

Section 2: Content Analyzer Programs: Modifying Statements

Introduction

Content Analyzer programs may edit the statements as well as decide whether or not they are printed. They are very useful where a series of editing operations has to be done time and time again. This section will introduce you to these capabilities. All these constructs will be covered in detail in Part Three.

A Content Analyzer program has several limitations. It can manipulate only one file and it can look at statements only in sequential order (as they appear in the file). It cannot back up and re-examine previous statements, nor can it skip ahead to other parts of the file. It cannot interact with the user. Part Four provides the tools to overcome these limitations.

String Construction

Statements and the contents of string variables may be modified by either of the following two statements:

ST ptr _ stringlist ;
The whole statement in which the text pointer named "ptr" resides will be replaced by the string list (to be described in a minute).

ST ptr ptr _ stringlist ;
The part of the statement from the first ptr to the second ptr will be replaced by the string list.

ptr may be a previously set text pointer or SF(ptr) or SE(ptr).

The content of string variables may be replaced with the string assignment statement:

stringname _ stringlist ;

The string list (stringlist) may be any series of string designators, separated by commas. The string designators may be any of the following (other possibilities to be described later):

a string constant, e.g. "ABC" or 'w

ptr ptr
the text between two text pointers previously set in either a statement or a string

stringname
a string name in asterisks, refering to the contents of the string

E.g.:

```
ST p1 p2 _ *string* ;
   or
ST p1 _ SF(p1) p1, *string*, p2 SE(p2);
```

(Note: these have exactly the same meaning.)

Example:

```
PROGRAM delsp    % Content analyzer.  Deletes all leading
spaces from statements. %
   %reserve space for ("declare") a text pointer named "pt"%
      DECLARE TEXT POINTER pt;
   (delsp) PROCEDURE ;
      %if any leading spaces, scan past them and set pointer%
         IF FIND 1$SP ^pt THEN
            %replace statement with text from pointer to
            statement end%
               ST pt _ pt SE(pt);
      %return, don't display anything%
         RETURN (FALSE) ;
```

```
        END.
      FINISH
```

More Than One Change per Statement

Part of a text pointer is a character count. This count stays the same until the text pointer is again set (to some other position), even though the statement has been editec. If, for example, you have the statement

```
   abcdefg
```

and if you have set a pointer between the "d" and the "e", it will always point between the fourth and fifth characters in the statement. If you then delete the character "a", your pointer will be between the "e" and the "f", now the fourth and fifth characters. For this reason, you should begin a series of edits with the last one in the statement and work backwards through the statement.

Controlling Which Statements are Modified

In TNLS, the Content Analyzer program will be called for commands which construct a printout of the file (Print and Output). The program will run on every statement for which it is called (e.g. every statement in the branch during a Print Branch command) which pass all the other viewspecs. Once you have written, compiled, and instituted a program which does some editing operation, the Print command is the easiest way to run the program on a statement, branch, plex, or group.
In DNLS, the system will call the Content Analyzer program whenever the display is recreated (e.g. viewspec F and the Jump commands), and also for the Output commands. If the program returns TRUE, it will only run on enough statements to fill the screen. It is safer to have programs that edit the file return FALSE. Then when you set viewspec i, it will run on all statements from the top of the display on, and when it is done it will display the word "Empty". At that point, change to viewspec j and recreate the display with viewspec F, then all statements including the changes will be displayed.
You can control which statements are edited with level viewspecs and the branch only (g) or plex only (l) viewspecs, and by positioning the top of your window.
After having run your program on a file, you may wish to Update to permanently incorporate the changes in the file. It is wise to Update before you run the program so that, if the program does something unexpected, you can Delete Modifications and return to a good file.

Problems

Try writing the following programs:

1) Remove any invisibles from the end of each statement.
2) Make the first word a statement name surrounded by parentheses.

Sample solutions:

Problem 1

```
PROGRAM endinv
   DECLARE TEXT POINTER ptr ;
   (endinv) PROCEDURE ;
      IF FIND ^ptr SE(ptr) 1$NP ^ptr
         THEN ST ptr _ SF(ptr) ptr ;
      RETURN (FALSE) ;
      END.
```

```
            FINISH
         Problem 2
            PROGRAM makename
               DECLARE TEXT POINTER ptr1, ptr2 ;
               (makename) PROCEDURE ;
                  IF FIND $NP ^ptr1 1$LD ^ptr2
                     THEN ST ptr1 _ '(, ptr1 ptr2, '), ptr2
                     SE(ptr2);
                  RETURN(FALSE)
                  END.
               FINISH
```

PART THREE: Basic L10 Programming

Section 1: The User Program Environment

Introduction

User-written Content Analyzer programs are called in the process of creating a view of an NLS file e.g., with a Print command in TNLS, with any of the Output commands, and with the Jump command in DNLS.

The sequence generator provides statements one at a time; the Content Analyzer may then check each one. Finally, the formatter prints it or puts it on the screen.

Thus if one had a user Content Analyzer program compiled and instituted, one could have a printout made containing only those statements in the file satisfying the pattern.

Attachable subsystems are independent of this portrayal process, although they are welcome to make use of it. They consist of commands, which may utilize all the powers of NLS.

The Sequence Generator

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

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 produce pointers only to those statements passing the structural filters; the formatter will then truncate the text to only the first line before it displays or prints the statement.

Content Analyzer Filters

One of the viewspecs that the sequence generator pays attention to is "i" -- the viewspec that indicates whether a user Content Analyzer filter is to be applied to the statement. If this viewspec is on, the sequence generator passes control to a user Content Analyzer program, which looks at the statement and decides whether it should be included in the sequence. If the statement passes the Content Analyzer (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 Content Analyzer program for verification. (The particular user program chosen as a filter is determined by what program is Instituted as the current Content Analyzer

program, as described below.)

In the process of examining a statement and deciding whether or not it should be printed, the Content Analyzer program may edit the text of the statement. These edits appear in the partial copy, just as if the user had made them himself. This provides a powerful mechanism for automatic editing.

In DNLS, if you display any statements, the program will stop after filling the screen. If you are not displaying any statements, the program will run on either the whole file, a plex (viewspec l), or a branch (viewspec g). These along with level clipping viewspecs give one precise control over what statements in the file will be passed to the program.

The Portrayal Formatter

The formatter arranges text passed to it by the sequence generator in the style specified by other viewspecs. 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.

Section 2: Program Structure

An NLS user program consists of the following elements, which must be arranged in a definite manner with strict adherence to syntactic punctuation:

The header -

a statement consisting of the word PROGRAM, followed by the name of a procedure in the program. Program execution will begin with a call to the procedure with this name.

PROGRAM name

The PROGRAM statement may have a statement name in parentheses; it will be ignored.

The word FILE should be substituted for the word PROGRAM if the code is to be compiled into a file to be saved.

The FILE statement may have a statement name; if so, that name will be used as the code-file symbol. You must not follow the word FILE with a name if there is a statement name preceding FILE.

The body -

consists of declarations and procedures in any order:

1) declaration statements which specify information about the data to be processed by the procedures in the program and enter the data identifiers in the program's symbol table, terminated by a semicolon. E.g.

```
DECLARE x,y,z ;
DECLARE STRING test[500] ;
REF x, z;
```

Declaration statements will be covered in Section 3 (5c).

2) procedures which specify certain execution tasks.

Each procedure must consist of:

the procedure name 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. E.g.

(name) PROCEDURE ;
(name) PROCEDURE (param1, param2) ;
You should always include a comment in the procedure statement breifly summarizing the function of the procedure.
the body of the procedure which may consist of LOCAL, REF, and L10 statements.
LOCAL and REF declarations within a procedure must precede executable code. They will be covered in Section 3 (5c).
L10 statements will be covered in Sections 4 and 5 (5d) (5e).
A RETURN statement must be included at some point, to pass control back to the calling procedure. If it is missing, execution will run off the end of the procedure and an ILLEGAL INSTRUCTION will occur.
the statement that terminates the procedure (note the final period):
END.

The program terminal statement -
FINISH
Note: this is a signal to the compiler to stop compilation; it does not mean stop execution. Any text after that in the NLS source file will be ignored.

Notes on Program Writing Style

Except for within literal strings, variable names, and special L10 reserved words, spaces are ignored. It is good practice to use them liberally so that your program will be easy to read.

Comments may be enclosed in percent signs (%) wherever spaces are allowed. They will be ignored by the compiler. It is good practice to use the level below the procedure statement for comments, filling in the code that executes the commented function at the level below the comment. It is also wise to add comments to any individual statements whose function is not obvious, particularly calls on other procedures.

You may find it convenient to add a comment to the FILE statement including the information needed by the Compile File command. E.g.

FILE program % (L10,) to (directory,program.subsys,) %

Also, NLS file structure is ignored. Structure is, however, very valuable in making the program readable, and it is good practice to use it in close correlation to the program's logical structure.

An example of a simple L10 program is provided here. The reader should easily understand this program after having studied this document.

```
PROGRAM delsp    % Content analyzer. Deletes all leading
spaces from statements. %
   %reserve space for ("declare") a text pointer named "pt"%
      DECLARE TEXT POINTER pt;
   (delsp) PROCEDURE ;
      %if any leading spaces, scan past them and set pointer%
         IF FIND 1$SP ^pt THEN
            %replace statement holding pt with text from
```

```
                pointer to statement end%
                   ST pt _ pt SE(pt);
            %return, don't display%
               RETURN (FALSE) ;
            END.
         FINISH
```

Section 3: Declarations

Introduction

L10 declarations provide information to the compiler about the data that is to be accessed; they are not executed. Every variable used in the program must be declared somewhere in the system (either in your program or in the NLS system). There are a number of types of variables available, each with its own declaration statement; the most frequently used are discussed here. (Complete documentation is available in the L10 Reference Guide -- 7052,)

Variables

Six types of variables are described in this document: simple, constants, arrays, text pointers, strings, and referenced. Each is represented by an identifier, some unique lowercase name. Each can be declared on three levels: local, global, or external.

Local Variables

A local variable 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 statements (to be explained below). Any LOCAL declarations must precede the executable statements in a procedure.

Local variables in the different procedures may have the same name without conflict. A global variable may not be declared as a local variable and a procedure name may be used as neither. In such cases the name is considered to be multiply defined and a compilation error results.

Global Variables

Global variables are defined in the program's DECLARE statements. Variables specified in these declarations are outside any procedure and may be used by all procedures in the program.

External Variables

External variables are defined in the program's DECLARE statements or in the NLS system program.

Variables specified in these declarations may be used by all procedures anywhere in the system. Many externals are defined as part of the NLS system; user programs have complete access to these. Since other procedures may access the same variable, the user programmer must be very careful about changing their values.

Simple Variables

Simple variables represent one computer word, or 36 bits, of memory. Each bit is either on or off, allowing binary numbers to be stored in words. Each word can hold up to five ASCII 7-bit characters, a single number, or may be divided into fields and hold more than one number.

Declaring a variable allocates a word in the computer to hold the contents of the variable. The variable name

refers to the contents of that word. One may refer to the address of that computer word by preceding the variable name by a dollar sign ($).

For example, if one has declared a simple variable called "num", one may put the number three in that variable with the statement:

```
num _ 3 ;
```

One may add two to a variable with the statement:

```
num _ num + 2 ;
```

One may put the address of num into a variable called addr with the statement:

```
addr _ $num ;
```

One may refer to predefined fields in any variable by following the name of the variable with a period, then the field name. For example, the fields RH and LH are globally defined to be the right and left half (18 bits) of the word respectively; e.g.

```
num.LH _ 2 ;
num.RH _ 3 ;
```

Fields may be defined by the user with RECORD statements (described in Section 5 of Part Five). Additionally, you may refer to system-defined fields (e.g. RH). They divide words into fields by numbers of bits, so they may refer to any declared word. For example, the field "LH" refers to the left-most 18 bits in any 36-bit word.

If you assign a full word to a field of n bits within a word, the right-most n bits will be assigned to the field in the destination word; the rest of the destination word will be untouched.

If you assign a field with a word to a full word, it will be right-justified within the destination word; the remaining bits in the destination word (to the left of the assigned bits) will be set to zero.

Declaring Simple Global Variables

```
DECLARE name ;
```

"name" is the name of the variable. It must be all lower-case letters or digits, and must begin with a letter.

E.g.

```
DECLARE x1 ;
```

Optionally, the user may specify the initial value of the variable being declared. If a simple variable is not initialized at the program level, for safety it should be initialized in the first executed procedure in which it appears.

```
DECLARE name = exp ;
```

exp is the initial value of name. It may be any of the following:

- a numeric constant optionally preceded by a minus sign (-)
- a string, up to five characters, enclosed in quotation marks
- another variable name previously defined in a SET statement (described below), causing the latter's value to be assigned

Examples:

```
            DECLARE x2=5;
               %x2 contains the value 5%
            DECLARE x3="OUT";
               %x3 contains the word OUT%
            DECLARE xx=x4;
               %x4 has previously been declared in a SET
               statement%
```

Formal parameters (passed to a procedure) are allocated as local simple variables, then initialized whenever the procedure is called. Within the called procedure, they should be treated as simple variables.

Constants

You may declare a (simple) variable to be a constant value with the statement:

```
SET name1=exp ;
```

where names and expressions are as described above for initializing simple variables.

Constants take no memory. They may be refered to just like simple variables, except the name must be preceded by a dollar sign ($). They may not be changed by the program. E.g.

after the declaration:

```
SET var = 4 ;
```

the assignment:

```
num _ $var ;
```

will assign the value 4 to the variable num.

Arrays

Multi-word (one-dimensional) array variables may be declared; computer words within them may be accessed by indexing the variable name. The index follows the variable name, and is enclosed in square brackets []. The first word of the array need not be indexed. The index of the first word is zero, so if we have declared a ten element array named "blah":

```
blah  is the first word of the array
blah[1]  is the second word of the array
blah[9]  is the last word of the array
```

Declaring Global Array Variables

```
DECLARE name[num] ;
```

num is the number of elements in the array if the array is not being initialized. It must, of course, be an integer.

E.g.

```
DECLARE sam[10];
```

declares an array named "sam" containing 10 elements.

Optionally, the user may specify the initial value of each element of the array. If array values are not initialized at the program level, for safety they should be initialized in the first executed procedure in which the array is used.

```
DECLARE name = (num1, num2, ... ) ;
```

num is the initial value of each element of the array. The number of constants implicitly defines the number of elements in the array. They may be any of the constants allowed for simple variables.

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 numbs=(1,2,3);
   declares an array named numbs containing 3
   elements which are initialized such that:
      numbs = 1
      numbs[1] = 2
      numbs[2] = 3
DECLARE motley=(10,$blah);
   declares an  array named motley containing 2
   elements which are initialized such that:
      motley = 10
      motley[1] = $blah = the address of the variable
      "blah"
```

Text Pointers

A text pointer is an L10 feature used in string manipulation constructions. It is a two-word entity which provides information for pointing to particular locations within text, whether in string variables or in NLS statements.

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.

A text pointer consists of two words: a string identifier and a character count. Assume you have declared a text pointer named "pt."

pt refers to the first word of the text pointer. The first word, called an "stid," contains three system-defined fields:

stfile -- the file number (if an NLS statement)
stastr -- a bit indicating string, not an NLS statement
stpsid -- the psid of the statement; every statement has a unique number (psid) attached to it.
The stid is the basic handle on a statement in L10. It is often used alone. Since it is a single-word value, it may be stored in a simple variable and passed easily between procedures, and is used by many routines to specify a statement or string.

If an stid is used without being properly set, the run-time error message "fst entry nonexistant" may result.

pt[1] refers to the second word of the text pointer. The second word contains a character count, with the first position being 1 (before the first character).

For example, one might have the following series of assignment statements which fill the three fields of the first word and the second word with data, with pt being the name of a declared text pointer:

```
pt.stfile _ fileno;
   %fileno is a simple variable with a number in it%
pt.stastr _ FALSE;
   %a statement, not a string%
pt.stpsid _ origin;
   %all origin statements have the psid = 2; origin is a
   global variable with the value 2 in it%
```

```
        pt[1] _ 1;
            %the word one after pt (i.e. the character count)
            gets 1, the beginning of the statement%
      It is important that stid*s be initialized properly to
      avoid errors.  Text pointers may be most easily initialized
      by setting them in a FIND statement (see Section 6).
   Declaring Text Pointers
      DECLARE TEXT POINTER pt ;
      The names p1, p2, p3, p4, and p5 are globally declared and
      reserved for system use.
Strings
   String variables are a series of words holding text.  When
   they are declared, the maximum number of characters is set.
   The first word contains the two globally defined fields:
      M -- the maximum number of characters the string can hold
      L -- the actual number of characters currently in the
      string
   The next series of words (as many as are required by the
   maximum string size) hold the actual characters, five per
   word, in ASCII 7-bit code.
      *str* refers to the contents of the string variable "str".
      str refers to the first word of the string variable "str";
      typically this is only useful in combination with the two
      fields "M" and "L":
         str.M  refers to the maximum declared length of the
         string variable "str" (an integer).
         str.L  refers to the current length of the string stored
         in the string variable "str" (an integer).
   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.
         To declare a string:
            DECLARE STRING name[num] ;
            num is the maximum number of characters allowed for
            the string
            Since the maximum statement length is 2000
            characters, you should not need to declare a string
            greater than 2000 characters long.
         E.g.
            DECLARE STRING lstring[100];
            declares a string named "lstring" with a maximum
            length of 100 characters and a current length of 0
            characters
         To declare and initialize a string:
            DECLARE STRING name="Any string of text" ;
            The length of the literal string defines the maximum
            length of the string variable.
         E.g.
            DECLARE STRING message="RED ALERT";
            declares the string message, with an actual and
            maximum length of 9 characters and contains the text
            "RED ALERT"
REF: Referenced Variables
   Reference Declarations
      After a simple variable has been declared, the REF
```

statement can define it to represent some other variable. A referenced variable holds the address of another declared variable of any type. Whenever the referenced variable is mentioned, L10 will operate on the other variable instead, as if it were declared in that procedure and named at that point.

This is useful when you wish a procedure to know about a multi-word variable. In procedure calls, you are only allowed to pass single-word parameters. If you wish a called procedure to use or operate on a text pointer, array, or string, you may pass the address of that multi-word variable. Then, in the called procedure, you must REF the formal parameter receiving that address. From then on in the called procedure, when you refer to the REFed parameter, you are actually operating on the multi-word variable declared in some other procedure to which the local REFed variable points, i.e. on the variable at the address contained in the REFed parameter.

Example:

If the simple variable "loc" in the current procedure has been REFed and contains the address of the string "str" local to the calling procedure, then operations on loc actually operate on the string in str:

```
*mes* _ *loc*;
   %mes gets the string in str%
*loc* _ "corpuscle";
   %str gets the string "corpuscle"%
```

Similarly, you cannot return multi-word variables from a called procedure. If you wish a procedure to return a string, you must declare the string as a local in the CALLING procedure, pass its address to a REFed variable in the called procedure. Then the called procedure can modify the string as if it were local (and return nothing). The modifications will be made in the actual string variable.

Unreferencing REFed Variables

One may refer to the actual contents (an address) of a REFed variable (i.e. "unref" it) by preceding the referenced variable name with an ampersand (&). If, for example, an address was passed to a REFed variable, and you wish now to pass that address on to another procedure, you can "unref" it, i.e. access the actual content (the address of some variable).

E.g. if x has been REFed and holds the address of y:

```
z _ x ;
   %z gets the CONTENTS of y%
z _ &x ;
   %z gets the ADDRESS of y%
```

This construct might be used, for example, if one procedure has been passed the address of a string, operates on it, then wishes to pass (the address of) that string on to another procedure that it calls.

This can be a tricky concept; it may be worthwhile to review this section carefully.

REFing Simple Variables

Once a simple variable has been declared (as a global, local, or parameter), it may be REFed with the L10

declaration statement:
REF var ;
It will be a reference from then on in that procedure, and you must always use the ampersand to refer to its actual contents: the address of the variable it references.
Note that the REF statement does not allocate storage; it just sets an attribute of an existing variable.
If you wish to use a variable that is not REFed as if it were REFed, enclose it in square brackets []. E.g. assume the simple variable "astr" holds the address of a string variable but was NOT REFed:
[astr] refers to the contents of the string variable whose address is in astr.

Note on Programming Style
You should always REF locals and parameters which hold the address of something to be accessed (even if that variable is only used to pass the address on to another procedure).

Declaring Many Variables in One Statement

One may avoid putting several individual declarations of variables in a series by putting variables of similar type, initialized or not, in a list in one statement following a single DECLARE, separated by commas and terminated by the usual semicolon. Array and simple varibles may be put together in one statement.

Examples:

```
DECLARE x, y[10], z = (1, 2, -5);
DECLARE TEXT POINTER tp, sf, pt1, pt2 ;
DECLARE STRING lstring[100], message="RED ALERT" ;
```

Declaring Locals

Program level declarations (DECLARE and REF) and procedures may appear in any order. However, procedure level declarations (LOCAL and REF inside a procedure) must appear before any executable statements in the procedure. The different types of variables may be declared in any order, but a variable must be declared before it can be REFed.

Whenever possible, LOCALs should be used instead of globals. It makes for a cleaner program if you pass parameters among procedures rather than depend on global variables to transmit information.

With one exception, a local variable declaration statement is just the same as a global with the word "LOCAL" substituted for the word "DECLARE". The one exception is that LOCAL declarations can not initialize the variables.

Examples:

```
LOCAL var, flag, level[12] ;
LOCAL TEXT POINTER tp, pt, sf ;
LOCAL STRING test[100], out[2000] ;
```

When a procedure is called by another procedure, the calling procedure may pass one-word parameters. The procedure receives these values in simple local variables declared in the PROCEDURE statement's parameter list. For example, two locals will automatically be declared and set to the passed values whenever the procedure "procname" is called:

```
(procname) PROCEDURE (var1, var2) ;
```

var1 and var2 must not be declared again in a LOCAL statement. They may, however, be REFed by a REF statement,

as discussed above, and used throughout the procedure.
The statement which calls procname may look like:
procname (locvar, 2) ;
var1 will be initialized to the value of the variable "locvar" and var2 will get the value 2.

Declaring Externals

Externals are declared just like globals, with one exception. The word DECLARE must be followed by the word EXTERNAL. E.g.

```
SET EXTERNAL one=1, two=2 ;
DECLARE EXTERNAL a, b[10], c=5 ;
DECLARE EXTERNAL TEXT POINTER exptr1, exptr2 ;
DECLARE EXTERNAL STRING exstr[100] ;
```

REF specifications may not be external to the program.

Accessing Registers

The user may access machine registers (the same length as other words, i.e. 36 bits) by naming them with the declaration:

```
REGISTER name = regnum ;
   or
REGISTER name1=regnum1, name2=regnum2 ;
```

The declared names will then represent the registers to which they are attached. You may then access or assign values to their content. On TENEX, the user programmer may use the first seven registers, registers 0 through 6. (Registers 7 through 15 are reserved for system use.) E.g.

```
REGISTER r0=0, r1=1, r2=2, r3=3, r4=4, r5=5, r6=6 ;
```

The names used in the above example are used most often by convention.

Registers must be used very carefully! They are typically used when calling TENEX JSYS (see Section 4). Many L10 constructs and procedures use the registers; you should assign their content to a variable immediately after the JSYS call if you wish to save it.

Section 4: Statements

Introduction

This section will describe some of the types of statements with which one can build a procedure. The term "expression" (often abbreviated to "exp") will be used in this section, and will be explained in detail in Section 5 (5e).

Assignment

In the assignment statement, the expression on the right side of the "_" is evaluated and stored in the variable on the left side of the statement.

```
var _ exp ;
```

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

One may make a series of assignments in one statement by enclosing the list of variables and the list of expressions in parentheses. The order of evaluation of the expressions is left to right. The expressions are evaluated and pressed onto a stack; after all are evaluated they are popped from the stack and stored in the variables.

```
(var1, var2, ...) _ (exp1, exp2, ...) ;
```

Naturally, the number of expressions must equal the number of variables.

Example:

```
        (a, b) _ (c+d, a-b)
        The expression c+d is evaluated and stacked, the
        expression a-b is evaluated and stacked, the value of
        a-b is popped from the stack and stored into b, and
        finally, the value of c+d is popped and stored into a.
        It is equivalent to:
           temp1 _ c+d ;
           temp2 _ a-b ;
           b _ temp2 ;
           a _ temp1 ;
   One may assign a single value to a series of variables by
   stringing the assignments together:
      var1 _ var2 _ var3 _ exp ;
      The assignment will be made from right to left.  var1,
      var2, and var3 will all be given the value of the
      expression.
      Example:
         a _ b _ 0;
         Both a and b will be given the value zero.  This type of
         statement can be useful in initializing a series of
         variables at the beginning of a procedure.
BUMP Statement
   The BUMP statement will add one to a variable:
      BUMP var ;
      This is equivalent to:
         var _ var + 1 ;
   BUMP DOWN will subtract one from a variable:
      BUMP DOWN var ;
      This is equivalent to:
         var _ var - 1 ;
   You may BUMP more than one variable in a single statement:
      BUMP var1, var2, var3,... ;
         or
      BUMP DOWN var1, var2, var3,... ;
IF Statement
   This form causes execution of a statement if a tested
   expression is TRUE.  If the expression is FALSE and the
   optional ELSE part is present, the statement following the
   ELSE is executed.  Control then passes to the statement
   immediately following the IF statement.
      IF testexp THEN statement ;
      IF testexp THEN statement1 ELSE statement2 ;
   The statements within the IF statement can be any statement,
   but are not followed by the usual semicolon; the whole IF
   statement is treated like one statement and followed by the
   semicolon.
   E.g.
      IF y=z THEN y_y+1 ELSE y_z ;
   In some cases, complex nested IFs may be simpler if rewritten
   as a CASE statement.
CASE Statement
   This form is similar to the IF statement except that it causes
   one of a series of statements to be executed depending on the
   result of a series of tests.
      CASE testexp OF
         relop exp : statement ;
```

```
      relop exp : statement ;
      relop exp : statement ;
            .
            .
      ENDCASE statement ;
```

where relop = any relational or interval operator (>=, <=, IN, etc.) see Section 5 (5e3c) and (5e3d).

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. Each expression is evaluated and compared according to the relational operator to the CASE expression. If the relationship is TRUE, the statement is executed. If the relationship is FALSE, the next expression and relational operator will be tried. If none of the relations is satisfied, the statement following the word "ENDCASE" will be executed. Control then passes to the statement following the CASE statement

Note that the relop and expressions are followed by a colon, and the statements are terminated with the usual semicolon. The word ENDCASE is not followed by a colon. In ENDCASE, the statement may be left out -- this is the equivalent of having a NULL statement there; nothing will happen.

Example:

```
CASE c OF
   = a:  %executed if c = a%
      x _ y;
   > b:  %executed if c > b%
      (x, y) _ (x+y, x-y);
   ENDCASE  %executed otherwise%
      y _ x;
CASE char OF
   = D:  %if char = the code for a digit%
      char _ '1;
   = UL:  %if char = the code for an upper-case letter%
      char _ '0;
   ENDCASE;  %otherwise nothing%
```

Several relations may be listed at the start of a single case; they should be separated by commas. The statement will be executed if any of the relations is satisfied.

```
CASE testexp OF
   relop exp: statement ;
   relop exp, relop exp: statement ;
   relop exp, relop exp, relop exp: statement ;
         .
         .
   ENDCASE statement ;
```

Example: .

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

```
                y _ x;
```

As a point of style, the conditions of the CASE statement should be put one level below the CASE statement in the source (text) file. The statements (if they are more than one line) may be put one level below the condition.

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 L10 statement

Example:

```
LOOP IF a>=b THEN EXIT LOOP ELSE a _ a+1 ;
```

(It is assumed that a and b have been initialized before entering the loop.)

The EXIT construction is described below. It is extremely important to carefully provide for exiting a loop.

WHILE...DO

This statement causes a statement 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 EXIT LOOP (described below).

```
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. Then control will pass to the next statement.

For example, if you want to fill out a string with spaces through the 20th character position, you could:

```
WHILE str.L < 20 DO *str* _ *str*, SP;   %what's already
there, then a space%
Remember that the first word of every string variable
has two globally defined fields:
   L -- actual length of contents of string variable
   M -- maximum length of string variable
```

The WHILE construct is equivalent to:

```
LOOP
   IF NOT exp THEN EXIT LOOP
   ELSE statement ;
```

Statement

UNTIL...DO Statement

This statement is similar to the WHILE...DO statement except that the statement following the DO is executed until exp is TRUE. As long as exp has a logical value of FALSE the statement will be executed repeatedly.

```
UNTIL exp DO statement ;
```

Example:

```
UNTIL a>b DO a _ a+1 ;
```

The UNTIL construct is equivalent to:

```
LOOP
   IF exp THEN EXIT LOOP ELSE statement ;
```

DO...UNTIL/DO...WHILE Statement

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

```
DO statement UNTIL exp;
```

```
      DO statement WHILE exp;
```

Thus the specified statement is always executed at least once (the first time, before the test is made). For example, this DO...UNTIL:

```
   DO array[var] _ 0 UNTIL (var := var - 1) = 0 ;
```

and this DO...WHILE:

```
   DO array[var] _ 0 WHILE (var := var - 1) > 0 ;
```

are both equivalent to:

```
   LOOP
      BEGIN
      array[var] _ 0 ;
      IF (var := var - 1) = 0 THEN EXIT LOOP ;
      END;
```

FOR...DO Statement

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

```
   FOR var UP UNTIL relop exp DO statement;
      (UP will be assumed if left out.)
   FOR var DOWN UNTIL relop exp DO statement;
   where
      var =   the variable whose value is incremented or
              decremented each time the FOR statement is
              executed
      relop = any relational operator (described in 5e3c)
      exp =   when combined with relop, determines whether
              or not another iteration of the FOR statement
              will be performed.  It is recomputed on each
      iteration.
```

E.g. `FOR i UP UNTIL > 7 DO a _ a + t[i] ;`

Optionally, the user may initialize the variable and may increment it by other than the default of one.

```
   FOR var _ exp1 UP exp2 UNTIL relop exp3 DO statement;
   FOR var _ exp1 DOWN exp2 UNTIL relop exp3 DO statement;
   where
      exp1 = an optional initial value for var.  If exp1 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.
   Note that exp2 and exp3 are recomputed on each iteration.
   Example:
      FOR k _ n UP k/2 UNTIL > m*3 DO x[k] _ k;
      is equivalent to
         k _ n;
         LOOP
            BEGIN
            IF k >m*3 THEN EXIT LOOP;
            x[k] _ k;
            k _ k + k/2;
            END;
```

BEGIN...END Statement

The BEGIN...END construction enables the user to group several statements into one syntactic statement entity. A BEGIN...END construction of any length is valid where one statement is

required.
BEGIN statement ; statement ; ... END ;
Example:

```
IF a >= b*c THEN
   BEGIN
   a_b;
   c_d+5;
   END  %no semicolon here because an L10
   statement here wouldn't have one; see 5d4%
ELSE
   BEGIN
   a_c;
   b_d+2;
   c_b*d*7;
   END;  %this semicolon terminates the entire IF
   statement%
```

Note the use of NLS file structure to clarify the logic and separate the blocks. Blocks should always be put one level below the statement of which they are a part.

EXIT Statement

The EXIT statement transfers control (forward) out of CASE or iterative statements. A CASE statement can be left with an EXIT CASE statement. All of the iterative statements (LOOP, WHILE, UNTIL, DO, FOR) can be exited by the EXIT LOOP statement. EXIT and EXIT LOOP have the same meaning.

EXIT LOOP num or EXIT num
EXIT CASE num

where num is an optional integer. The optional number (num) specifies the number of lexical levels of CASE or iterative statements respectively that are to be exited (e.g. if loops are nested within loops). If a number is not given then 1 is assumed.

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;
             ........
```

REPEAT Statement

The REPEAT statement transfers control (backward) to the front of CASE or iterative statements. The optional number has the same meaning as in the EXIT statement. REPEAT and REPEAT CASE have the same meaning.

```
REPEAT LOOP num
REPEAT CASE num (exp)   or   REPEAT num (exp)
```

If an expression is given in parentheses 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.

Examples:

```
CASE exp1 OF
   =something:
      BEGIN
      ........
      IF test1 THEN REPEAT;
         %REPEAT with a reevaluated exp1%
      ........
      IF test2 THEN REPEAT(exp2);
         %REPEAT with exp2%
      ........
      END;
   ........
   ENDCASE ;
LOOP
   BEGIN
   ........
   IF test THEN REPEAT LOOP;
      %REPEAT LOOP will go to the top of the LOOP%
   ........
   END;
```

DIVIDE Statement

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

```
DIV exp1 / exp2 , quotient , remainder ;
```

Quotient and remainder are variable names in which the

respective values will be saved after the division.

E.g.

```
DIV a / b, a, r ;
a will be set to a/b to the greatest integer with r
getting the remainder
```

Floating point calculations are described in Part Five, Section 4.

PROCEDURE CALL Statement

Procedure calls direct program control to the procedure specified. A procedure call occurs when the name of the procedure is followed by parentheses. If the procedure requires that arguments be passed, they should be included in the parentheses, separated by commas.

```
procname (exp, exp, ...) ;
where procname = the name of a procedure
   exp = any valid L10 expression (explained in Section 5).
   The set of expressions separated by commas is the
   argument list for the procedure.
```

The argument list consists of a number of expressions separated by commas. The number of arguments should equal the number of formal parameters for the procedure. The argument expressions are evaluated in order from left to right. Each expression (parameter) must evaluate to a one-word value. The values will be assigned to the formal parameters of the called procedure.

To pass an array, text pointer, string, or any multi-word parameter, the programmer may pass the address of the first word of the variable, then REF the receiving local in the called procedure.

For example, one may pass an stid directly, but to pass a text pointer, you must pass the address of the text pointer and REF the receiving parameter. Remember that a dollar sign ($) preceding a variable represents the address of that variable.

The procedure may return one or more values. The first value is returned as the value of the procedure call. Therefore, if only one value is returned, one might say:

```
a _ proc (b) ;
```

In this context, the procedure call is an expression.

If more than one value is returned by the called procedure, one must specify a list of variables in which to store them. 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. The first RETURN value is still taken only as the value of the procedure call.

```
var _ procname (exp, exp, ... : var, var, ...) ;
```

Example:

If procedure "proc" ends with the statement

```
RETURN (a,b,c)
```

then the statement

```
q _ proc(:r,s);
```

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

A procedure call may just exist as a statement alone without returning a value. Not all procedures require parameters, but the parentheses are mandatory in order to distinguish a procedure call from other constructs.

E.g. lda();

If a block of instructions are used repeatedly, or are duplicated in different sections of a program, it is often wise to make them a separate procedure and simply call the procedure when appropriate.

It is considered good style to "modularize" the functions of your program as much as possible, where each procedure represents a function which will be performed no matter which procedure called it. This implies very limited use of global variables and careful definition of the procedure interface.

Procedures should not be made too long, nor have complex nested loops. Often breaking the code into a number of shorter procedures will make the program clearer and easier to debug.

A procedure may recursively call itself. Each call will have its own unique set of local variables. This may be useful if a procedure is built to handle a general case as well as a specific case or number of cases. The general case may call that same procedure for the specific case after some manipulations.

A great many procedures are part of the NLS system and are available to your programs. A list of them is available in the file <NLS,XPROCS,> or <NLS,SYSGD,>. SYSGD lists links to the source code, so that you can examine the procedure in detail to see just what it expects as arguments and what it returns.

RETURN Statement

This statement causes a procedure to return control to the procedure which called it. Optionally, it may pass the calling procedure an arbitrary number of results. The order of evaluation of results is from left to right.

```
RETURN ;
RETURN (exp, exp, ...) ;
```

E.g.

```
RETURN (TRUE, a+b) ;
RETURN ( getnmf(stid) ) ;
```

GOTO Statement

Any statement may be labeled; one puts the desired label (a string of lower case letters and digits) in parentheses and followed by a colon at the beginning of a statement.

```
(label): statement ;
```

E.g.

```
(there): a _ b + c ;
```

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

```
GOTO label ;
```

E.g.

```
GOTO there ;
```

GOTO statements make reading and debugging your program difficult and are not considered good style; they can usually be eliminated by use of procedure calls and the iterative

statements.

NULL Statement

The NULL statement may be used as a convenience to the programmer. It does nothing.

```
NULL ;
```

Example:

```
CASE exp OF
    =0, =1: NULL;
    ENDCASE y_1;
```

JSYS Call and Assembly Language Statement

The use of these capabilities should be limited to system programmers. Assembly language code makes user programs difficult to understand and to maintain as the executive underlying NLS changes over time. L10 procedures are available to accomplish most of the tasks one might want to do with a JSYS. System programmers should refer to the TENEX JSYS manual for a description of the available JSYS's. Assembly language statements may be included in the L10 code by preceding the statement with an exclamation-point (!). The instruction must be upper-case; the arguments must be lower-case. E.g.

```
!PUSH s,jfn ;
```

A TENEX JSYS may be invoked with a statement similar to the procedure call statement; the name of the JSYS must be lower-case, preceded by an exclamation-point:

```
!jsysname (reg1, reg2,...) ;
```

E.g. `!gjinf() ;`

The arguments in the parentheses are evaluated and loaded into the registers before the JSYS is invoked. The first argument will be put in register one, the second in register two, etc. Up to eight arguments may be given.

Like a procedure call, multiple results may be received. They will be taken in order from the registers. (See <13510,3c> for a description of user JSYS calls.

Some JSYS return to the assembly-language line of code (not the L10 statement) one beyond the normal return location. With such JSYS, you may use the SKIP construct to test if it has done so:

```
IF SKIP !jsys(arg1,...) THEN ... ;
```

In using SKIP, you may not receive multiple results directly, but must read the registers into globals (see 5c12).

Section 5: Expressions

Introduction

This section will describe the composition of the expressions, which are an integral part of many of the statements described in Section 4.

Primitives

Primitives are the basic units which are used as the operands of L10 expressions. There are many types of elements that can be used as L10 primitives; each type returns a value which is used in the evaluation of an expression.

Each of the following is a valid primitive:

a constant (see below)

any valid variable name, refering to the contents (of the first word, if not indexed) of that variable

the contents of a string variable, refered to as *var*

a dollar sign ($) followed by a variable name, refering to the address of the variable
a procedure call which returns at least one value
the first (leftmost) value returned is the value of the procedure call; other values may be stored in other variables as described in Section 4.
an assignment (see below)
classes of characters; described in Section 1 of Part One
MIN (exp, exp, ...) the minimum of the expressions
MAX (exp, exp, ...) the maximum of the expressions
TRUE has the value 1
FALSE has the value 0
VALUE (astring) given the address of a string containing a decimal number, has the value of the number
VALUE (astring, num) given the address of a string containing a number and the base of that number, has the value of the number (allows other than base-ten numbers)
READC (see below)
CCPOS (see below)
FIND
used to test text patterns and load text pointers for use in string construction (see Section 6); returns the value TRUE or FALSE depending on whether or not all the string tests within it succeed.
POS
POS textpointer1 relop textpointer2
may be used to compare two text pointers. If the POS construction is not used, only the first words of the pointers (the stid's) will be compared. If a pointer is before another, it is considered less than the other pointer.
E.g.
POS pt1 = pt2
POS first >= last

Constants

A constant may be either a number or a literal constant. 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:

```
64    =  100B  =  1B2
144B  =  100   =  1D2
```

Literals may be used as constants as they are represented internally by numeric values. The following are valid literal constants:

-any single character preceded by an apostrophe
e.g. 'a represents the code for 141B.
-the following synonyms for commonly used characters:
ENDCHR -- endcharacter as returned by READC
SP -- space
ALT -- Tenex's version of altmode or escape (=33B)
CR -- carriage return

LF -- line feed
EOL -- Tenex EOL character
TAB -- tab
BC -- backspace character
BW -- backspace word
C. -- center dot
CA -- Command Accept
CD -- Command Delete

Assignments

An assignment can be used as a value in an expression. The form a _ b has the effect of storing b into a and has the value of b as the value of the assignment.

Another form of the assignment statement is:

a := b

This will store b into a, but have the old value of a as the value of the assignment when used as a primitive in an expression.

For example,

b _ (a := b) ;

The value of b will be put in a. The assignment will get the old value of a, which is then put in b. This transposes the values of a and b. (The parentheses are not really necessary.)

READC - ENDCHR

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. CCPOS and FIND are explained in detail in Section 6 of this document.

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

For 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).

CCPOS

When used as a primitive, CCPOS has as its value the index of the character to the right of the current character position. 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). CCPOS is more commonly used as a statement to set the current character position for use in text pattern matching. This is discussed in detail in Section 6. CCPOS may be useful as an index to sequentially process the first n characters of a string (assumed to have at least n characters).

Example:

```
            CCPOS SF(*str*);
               %CCPOS now has the index value of one, the front
            of the string%
            UNTIL CCPOS > n DO process(READC);
               %READC reads the next character and increments
            CCPOS%
```

Operators

Primitives may be combined with operators to form expressions. Four types of operators will be described here: arithmetic, relational, interval, and logical.

Arithmetic Operators

+ (in front of a number) -- positive value
- (in front of a number) -- negative value
+ -- addition
- -- subtraction
* -- multiplication
/ -- integer division (remainder not saved)
MOD -- a MOD b gives the remainder of a / b
.V -- (OR) a .V b => bit pattern which has 1's where either a or b contains 1, 0 elsewhere
.X -- (XOR) a .X b => bit pattern which has 1's where either a holds 1 and b contains 0, or a contains 0 and b contains 1, 0 elsewhere
.A -- (AND) a .A b => bit pattern which has 1's where both a and b contain 1, 0 elsewhere

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.

Operator	Meaning	Example	
=	equal to	4+1 = 3+2	(TRUE, =1)
#	not equal to	6#8	(TRUE, =1)
<	less than	6<8	(TRUE, =1)
<=	less than or equal to	8<=6	(FALSE, =0)
>	greater than	3>8	(FALSE, =0)
>=	greater than or equal to	8>=6	(TRUE, =1)
NOT <other-relational-operator>		6 NOT > 8	(TRUE, =1)

Interval Operators

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

IN (primitive, primitive) IN [primitive, primitive]

The value is tested to see whether or not it lies within 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 IN [1,100)
```

is the same as

```
      (x >=1) AND (x < 100)
```

Logical 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.

OR

```
a OR b  = TRUE  if a = TRUE  or if  b = TRUE
        = FALSE if a = FALSE and if b = FALSE
```

AND

```
a AND b = TRUE  if a = TRUE  and if b = TRUE
        = FALSE if a = FALSE or if  b = FALSE
```

NOT

```
NOT a   = TRUE  if a = FALSE
        = FALSE if a = TRUE
```

Expressions

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.

Examples of assigning an expression to a variable:

```
var _ 0;
var _ var + 2 ;
var _ POS ptr1 >= ptr2 ;
var _ (a > b) OR (a IN [c, d]) ;
```

Liberal use of parentheses is highly recommended.

Special L10 expressions are:

- the FIND expression which is used for string manipulation, and
- 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 (see below).

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 heirarchy are performed from left to right in an expression. Operations in parentheses are performed before operations not in parentheses.

The order of execution of operators (from first to last) 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

The two conditional constructs (IF and CASE) can be used as expressions as well as statements. As expressions, they must return a value.

IF Expressions

IF testexp THEN exp1 ELSE exp2

testexp is tested for its logical value. If testexp is TRUE then exp1 will be evaluated. If it is FALSE, then exp2 is evaluated.

Therefore, the result of this entire expression is EITHER the result of exp1 or exp2.

Example:

```
y _ IF x IN[1,3] THEN x ELSE 4;
   %if x = 1, 2, or 3, then y_x; otherwise y_4%
```

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.

```
CASE testexp OF
   relop exp : exp ;
   relop exp : exp ;
   relop exp : exp ;
       .
       .
   ENDCASE exp
```

where relop = any relational or interval operator (>=, <, =, IN, etc. See above (5e3c) and (5e4d)

In the above, the testexp is evaluated and used with the operator relops and their respective exps to test for a value of TRUE or FALSE. If TRUE in any instance, the companion expression to the right of the colon is executed and taken to be the value of the whole expression. A value of FALSE for all tests causes the next relop in the CASE expression to be tested against the testexp. If all relops are FALSE, the ENDCASE expression is taken to be the value of the whole expression.

Note that ENDCASE cannot be null; it must have a value. As with the CASE statement, any number of cases may be specified, and each case may include more than one relop and expression, separated by commas.

Example:

```
y _ CASE x OF
   <3:      x+1;
   =3, =4:  x+2;
   =5:      x;
   ENDCASE x*2;
```

Value of X	Value of y
2	3
3	5
4	6
5	5
6	12

String Expressions

L10 also provides several expression forms which are used for string manipulation and evaluation. These are

discussed in Section 6 of this document. When using string manipulation statement forms as expressions, parentheses may be necessary to prevent ambiguities.

Section 6: String Test and Manipulation

Introduction

This section describes statements which allow 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 (or statement), 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.

Current Character Position (CCPOS)

The Current Character Position is similar to the TNLS CM (Control 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. In Content Analyzer programs, it is initialized to the BEGINNING OF EACH NEW STATEMENT. For each new statement, the scan direction is initialized to LEFT TO RIGHT. It is moved through the statement or through strings by FIND expressions. It may be set to a particular position in a statement or string by the L10 statement:

```
CCPOS pos ;
```

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.
If a text pointer is given after CCPOS, then the character position is set to that location. A text pointer points between two characters in a string.
e.g. CCPOS pt1 ;

String Front -- left of the first character
SF(stspec)
When SF is specified, CCPOS will be set before the first character of the statement or string variable specified by stspec.
stspec is a string specification that may be expressed as
- an stid (e.g. the first computer word of a previously declared text pointer), or
- a previously declared string name enclosed in asterisks.

Examples:

```
CCPOS SF(pt1) ;
   %pt1 is a text pointer%
CCPOS SF(stid) ;
   %stid is an stid%
CCPOS SF(*str*) ;
   %str is a string%
```

String End -- right of the last character
SE(stspec)
When SE is specified scanning will take place from right to left, and CCPOS will be set after the last character of the statement or string variable specified by stspec.

A string (*stringname*) is given after CCPOS. The position

is moved to the beginning of that string.
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.
E.g.
```
CCPOS *str*[3] ;
```
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.
E.g.
```
CCPOS *str* ;
   means the same as
CCPOS SF(*str*);
```
Setting the current character position with the CCPOS statement also sets the scan direction to forward (left-to-right), except if the SE construct is used.

FIND Statement

The FIND statement specifies a string pattern to be tested against a statement or string variable, 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 left at the position prior to the FIND statement. The values of text pointers set in the statement prior to the failing element will remain as set; others of course will not be changed.
```
FIND pattern ;
```
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 any one of the elements is false.
E.g.
```
IF FIND pattern THEN ... ;
```
It is good practice to use FIND as an expression with the appropriate error conditions if the FIND fails. If the FIND fails, text pointers may not be set as expected.

FIND Patterns

A string pattern may be any valid combination of the following logical operators, testing arguments, and other non-testing parameters (note the identity with Content Analyzer Patterns):
Pattern Matching Arguments--
(each of these can be TRUE or FALSE)
string constant, e.g. "ABC"
or any character, preceded by an apostrophy
It should be noted that if the scan direction is set right-to-left the string constant pattern should be reversed. In the above example, one would have to search for "CBA".
Any of the system defined mnemonics, as described in the last section (5e2c), such as "SP" or "CR", are also valid.

character class
look for a character of a specific class; if found, = TRUE, otherwise FALSE.
Character classes:
CH -- any character
L -- lowercase or uppercase letter
UL -- uppercase letter
LL -- lowercase letter
D -- digit
LD -- lowercase or uppercase letter or digit
NLD -- not a letter or digit
ULD -- uppercase letter or digit
LLD -- lowercase letter or 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.

(elements)
look for an occurrence of the pattern specified by the elements. If found, = TRUE, otherwise FALSE. Elements may be any pattern; the parentheses serve to group the elements so as to be treated as a single element in any of the following elements.

-element
TRUE only if the string constant or character class element following the dash does not occur.

NOT element
TRUE only if the element or group of elements following the NOT does not occur.

[elements]
TRUE if the pattern specified by the elements can be found anywhere in the remainder of the string. elements may be any pattern; the squarebrackets also group the elements so as to be treated as a single element. It first searches from current pcsition. If the search failed, then the current position is incremented by one and the pattern is tried again. 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 element
find (exactly) the specified number of occurrences of the element.
E.g.
3(LD) means three letters or digits

NUM1 $ NUM2 element
Tests for a range of occurrences of the element specified. If the element 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.
         Examples:
            2$4(UL) -- from two to four upper-case letters
            $10(SP) -- up to ten spaces
            1$(*.) -- one or more periods
      ID = user-ident
      ID # user-ident
         if the string being tested is the text of an NLS
         statement then ident of the user who created or last
         edited the statement is tested by this construction;
         if CCPOS is in a string, you will get the error
         "string treated as statement"
      FT var
         TRUE if the variable holds a value of TRUE
         (non-zero).
      SINCE datim
         if the string being tested is the text of an NLS
         statement, this test is TRUE if the statement was
         created or modified 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 or modified before the date and time (datim,
         see below) specified.
         Acceptable dates and times follow the forms permitted
         by the TENEX system's IDTIM JSYS described in detail
         in the TENEX JSYS manual.  It accepts "most any
         reasonable date and time syntax."
            Examples of valid dates:
               17-APR-70
               APR-17-70
               APR 17 70
               17 APRIL 70
               17/5/1970
               5/17/70
               APRIL 17, 1970
            Examples of valid times (zero assumed if time left
            out):
               1:12:13
               1234
               1234:56
               1:56AM
               1:56-EST
               1200NOON
               16:30  (4:30 PM)
               12:00:00AM   (midnight)
               11:59:59AM-EST   (late morning)
               12:00:01AM   (early morning)
         Examples:
            BEFORE (MAR 19, 73 16:49)
            SINCE (25-JUL-73 2130:00)
   These may not appear in Content Analysis patterns, but are
   valid elements in FIND statements in any program:
      *stringname*
```

the contents of the string variable

BETWEEN pos pos (element)
Search limited to between positions specified. pos is a previously set text pointer; the two must be in the same statement or string. Scan character position is set to first position before the pattern is tested (This is not an unanchored scan unless square brackets are used within the parentheses.). E.g.

```
BETWEEN pt1 pt2 (2D [.] $NP)
```

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. The character position will be reset to its position before encountering the group before a new group is tested. Any text pointers set within a test pattern before it fails will retain their new values. (See examples below.)

/
AND
OR
These logical concatenators bind in the order in which they are listed. I.e.

```
a / b AND c
```

means the same as

```
(a / b) AND c
```

Other Elements--
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 OR when a value of TRUE is desired even if all tests fail.

ENDCHR
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 mneumonics are provided and may be referenced by the identifier "ENDCHR".

These may not appear in simple Content Analysis Patterns, but may in FIND statements:

pos
pos is a previously set text pointer, or an SE(pos) or SF(pos) construction. 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.
E.g.
FIND x; %sets CCPOS to position of previously set text pointer x%

^ 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.

FS var
FR var
FS will set the variable to TRUE (1). FR will reset the variable to FALSE (0).

String Construction

One may modify an NLS statement or a string with the statement:

ST pos _ stringlist ;
The whole statement or string in which pos resides will be replaced by the string list.

ST pos pos _ stringlist ;
The part of the statement or string from the first pos to the second pos will be replaced by the string list. "pos" may be a previously set text pointer or the SF(pos)/SE(pos) construction.

There are two additional ways of modifying the contents of a string variable:

ST *stringname*[exp TO exp] _ stringlist ;
means the same as
stringname[exp TO exp] _ stringlist ;
The string from the first position to the second position will be replaced by the string list. The square-bracketed range is entirely optional; if it is left off, the whole string will be replaced.
Note that the "ST" is optional when assigning a stringlist to the contents of a string variable. The statement then resembles any simple assignment statement. I.e.
stringname _ stringlist ;

The string list (stringlist) may be any series of string designators, separated by commas. The string designators may be any of the following:

the word NULL
represents a zero length (empty) string

string constant, e.g. "ABC" or 'w

part of any string or statement, denoted either by
two text pointers previously set in either a statement or a string
pos pos
a string name in asterisks, refering to the whole string
stringname
a string name in asterisks followed by an index, refering to a character in the string

```
            *stringname*[exp]
               (The index of the first character is one.)
         a string name in asterisks followed by two indices,
         refering to a substring of the string
            *stringname*[exp TO exp]
               A construction of the form *str*[i TO j] refers to
               the substring starting with the ith character in
               the string up and including the jth character.
               Examples:
                  *str*[7 TO 10] is the four character substring
                  starting with the 7th character of str.
                  *str*[i TO str.L] is the string str without the
                  first i-1 characters.  (i is a declared
                  variable.)
      + substring
         substring capitalized
      - substring
         substring in lower case
      exp
         value of a general L10 expression taken as a character;
         i.e., the character with the ASCII code value (see chart
         at end of document) equivalent to the value of the
         expression
      STRING (exp1, exp2);
         gives a string which represents the value of the
         expression exp1 as a signed decimal number.  If the
         second expression is present, a number of that base is
         produced instead of a decimal number.
         E.g.
            STRING (3*2) is the same as the string "6"
               or
            STRING (14,8) is the same as the string "16"
   Examples:
      ST p1 p2 _ *string*;
         does the same as
      ST p1 _ SF(p1) p1, *string*, p2 SE(p2);
         assuming p1 and p2 have been set somewhere in the same
         statement.  The latter reads "replace the statement
         holding p1 with the text from the beginning of the
         statement to p1, the contents of string, then the text
         from p2 to the end of the statement."
      *st*[low TO high] _ "string";
         does the same as
      *st* _ *st*[1 TO low-1], "string", *st*[high+1 TC st.L];
         assuming low and high are declared simple variables.
Example:
   Let a "word" be defined as an arbitrary number of letters and
   digits.  The text pointer "t" is set before or after some
   character in the word.  The two statements in this example
   delete the word which holds 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 ptr1 and ptr2 are used to delimit the left
   and right respectively of the string to be deleted.
   IF (FIND t < $LD ^ptr1 > $LD (SP ^ptr2 / ^ptr2 ptr1 < (SP
```

```
^ptr1 / TRUE)) ) THEN
   ST ptr1 ptr2 _ NULL;
```

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

More Than One Change per Statement

The second word of a text pointer, the character count, stays the same until the text pointer is again set to some other position (as does the first word), even though the statement has been edited. If, for example, you have the statement

```
abcdefg
  /\
```

and if you have set a pointer between the "d" and the "e", it will always point between the fourth and fifth characters in the statement; the second word of the text pointer holds the number 5. If you then delete the character "a", your pointer will be between the "e" and the "f".

```
bcdefg
  /\
```

For this reason, you probably want to do a series of edits beginning with the last one in the statement and working backwards.

Text Pointer Comparisons

This may be used to compare two text pointers.

```
POS pt1 =  pt2;
           #
           >
           <
           >=
           <=
```

pt1 and pt2 are text pointers.

NOT may precede any of the relational operators. 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.

A pointer closer to the front of the statement is "less than" a pointer closer to the end.

Section 7: Invocation of User Filters

Introduction

The Content Analyzer filters described in this document may be imposed through the NLS PROGRAMS subsystem.

User-attachable subsystems may be written for more complex tasks. This type of user program and NLS procedures which may be accessed by them will be discussed in Part Four. With such a program, however, the user will still make use of the commands in the NLS PROGRAMS subsystem.

This section describes NLS commands which 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 the L10 language written in an NLS source file into object code, which the computer can use to execute those instructions.

Loading--
: is the process which copies the compiled instructions

into the user-programs buffer.

Institution--
is the process by which a compiled and loaded Content Analyzer program is designated as the current Content Analyzer filter.

This section additionally presents examples of the use of the L10 programming language. They do not make use of any constructions not explained so far in this manual.

Programs Subsystem

Introduction

The PROGRAMS subsystem provides several facilities for the processing of user written programs and filters. It is entered by using the NLS command:

Goto Programs OK

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 to see the status of user programs.

PROGRAMS subsystem commands

After entering the PROGRAMS subsystem, you may use one of the following commands:

Show Status of programs buffer

This command prints out information concerning active user programs and filters which have been loaded and/or instituted:

Show Status (of programs buffer) OK

When this command is executed the system will print:

-- the names of all the programs in the user programs buffer, including those generated for simple Content Analysis patterns, starting with the first program loaded.

-- the remaining free space in the buffer. The buffer contains the compiled code for all the current compiled programs.

-- the current Content Analyzer Program or "None"

-- the current user Sequence Generator program or "None"

-- the user Sort Key program or "None"

Compile

L10 Program

This command compiles the program specified.

Compile L10 (user program at) ADDRESS OK

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

This command causes the program specified to be compiled and loaded into the user program buffer in a single operation. The program is not instituted.

The name of the program is the visible following the word PROGRAM. ADDRESS points to the PROGRAM statement.

The program may be instituted by the appropriate commands.

File

The user program buffer is cleared whenever the user resets or logs out of the system. If you have a long

program which will be used periodically, you may wish to save the compiled code in a TENEX file. It can then be retrieved with the Load Program command. The command to compile the code into a TENEX file is:

Compile File (at) ADDRESS (using) L10 OK (to file) FILENAME OK

The FILENAME must be the same as the program name. The program will then be compiled and stored in the TENEX file of the given name (with the extension REL, unless otherwise specified). The user may then load it at any time.

Before doing this, the programmer must replace the word PROGRAM at the head of the program with the word FILE.

Content Analyzer Pattern

This command allows the user to specify a Content Analyzer pattern as a Content Analyzer filter.

Compile Content (analyzer filter) ADDRESS OK

The pattern must begin with the first visible after the ADDRESS, or at that point you may type it in. It will read the pattern up to a semicolon, so be sure to insert a semicolon where you want it to stop. When this command is executed, the pattern specified is compiled into the buffer, AND it is automatically instituted as the Content Analyzer filter.

Procedure

This command compiles a single procedure.

Compile Procedure (at) ADDRESS OK

ADDRESS is the address of the PROCEDURE statement. This command causes the procedure specified to be compiled and loaded into the user program buffer in a single operation.

If a procedure of the same name has already been loaded (in the user programs buffer or in the NLS system), the old procedure will be replaced. I.e. any calls to that procedure name will invoke the newly compiled procedure.

Error Message during Compilation

"SYNTAX ERROR" messages include the type of error, the location of the line of assembly code that caused trouble, and a few characters of the NLS source code. The last of these characters is the one which caused the error. In some cases this may be misleading, when a previous error (e.g. a missing quote or percent sign) caused trouble later in the compilation.

"ext & local" -- a symbol was used as both an external or global and a local variable in the file. If a variable is not declared in the program, the compiler assumes it is a system EXTERNAL. If it is later used as a LOCAL, an error will result.

"field too large" -- a field may not be defined as more than 36 bits.

"sides not equal" -- in a multiple assignment statement, the sides must have the same number of

values, e.g. (a,b,c) _ (x,y,z);
"not REF or POINTER" -- an ampersand (&) was used on a variable not REFed or declared as a POINTER (not described in this document).
"8 args max" -- you may not pass more than eight arguments in a JSYS call.

"SYSTEM ERROR" messages also include the type of error, the location of the line of assembly code that caused trouble, and a few characters of the NLS source code.

"EOF READ" -- the compiler hit the end of the NLS file before it read a FINISH statement. (This may happen if you don't have viewspecs set to all lines, all levels.)
"HASH TABLE FULL" -- you have used too many symbols in the file. Each file is limited to approximately 2000 symbols.
"BACKUP TOO FAR" -- a symbol or a literal string (text within quotes) has too many characters in it. They are limited to 148 characters.
"SYMBOL TOO LONG" -- as above, a symbol has too many characters in it.
"INPUT TOO LONG" -- as above, a literal string has too many characters in it.
"S.S. FULL" -- as above, a symbol has too many characters in it.
"I/O ERROR" -- a number has too many digits in it.
"LIT TABLE FULL" -- the file has too many literal strings and numbers.

"PUSHDOWN OVERFLOW" means that one of the stacks that the compiler uses overflowed. Look for an L10 statement containing too many parentheses or particularly complex constructions. You may have to break some statements into multiple statements.

"Boolean as operand" -- you used an expression as a parameter or in a RETURN statement. This is NOT an error, but only a warning of unusual (though in many cases good) programming practice.

If you include the L10 statement
```
NOMESS ;
```
at the beginning of the file, at the same level as global declarations (i.e. not within a procedure), this warning will not be printed. Errors will be printed as usual.

When the compilation is finished, it will list the number of errors and wait for a Command Accept to continue. You should then search for the error in the NLS source code file, correct it, and recompile before attempting to use the program.

Errors involving undefined variables will be reported when you attempt to load the program. Of course any code using these variables will cause execution errors.

If you include the L10 statement
```
LIST ;
```
anywhere in the code, all the undefined symbols at

that point in the compilation will be printed. The Compile Procedure command will generate undefined variable errors legitimately if the procedure refers to global variables.

If the addition of your program to the user programs buffer requires more than the maximum space allotted for user programs (either in number of pages or number of symbols), you will get a "format error" upon loading. (If you have any other programs loaded, use the "Delete All" command prior to loading.)

NDDT (described in Part Five, Section 2) will help you trace run-time errors to errors in the NLS source code.

Load Program

A pre-compiled program existing as a REL file may be loaded into the program buffer with the command:

Load Program FILENAME OK

If the FILENAME is specified without specifying an extension name, this command will search the connected directory, then the <PROGRAMS> directory, for the following extensions:

REL -- it will simply load the REL file

CA -- it will load the program and institute it as the current content analyzer program

SK -- it will load the program and institute it as the current sort key extractor program

SG -- it will load the program and institute it as the current sequence generator program

SUBSYS -- it will load the program and then look for a file of the same name with extension CML; if both are successfully loaded, they will be treated as a single program

CML -- it will load the program and then try to attach it as a subsystem

PROC-REP -- it will load the program and then try to replace an existing procedure of the same name as the TENEX code file by the first procedure in loaded program

Sort key extractor and sequence generator programs are more complex and are generally limited to experienced L10 programmers.

FILENAME is the name of the TENEX code file, not the name of the program.

If any variables are undefined, they will be reported upon loading. The program should not be used until those variables are declared somewhere.

Delete

All

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

Delete All (programs in buffer) OK

The user programs buffer shares memory with data pages for files which the user has open, therefore increasing the size of the user programs buffer

decreases the amount of space available for file data with a possible slowdown in response for that user. The buffer size is increased automatically as needed. This command also resets the buffer size to the original 8 pages (saving system storage space).

Last

This command deletes the most recently loaded program in the buffer. The program is deinstituted if instituted and its space in the buffer marked as free.

Delete Last (program in buffer) OK

Run Program

This command transfers control to the specified program. This type of program is used very little, having been substantially replaced by user-attachable subsystems, as described in Part Four.

Run Program PROGNAME OK
Run Program NUMBER OK

PROGNAME is the name of a program which had been previously compiled. That is, PROGNAME must be in the buffer when this command is executed.
Instead of PROGNAME, the user may specify the program to be run by its number. This first program loaded into the buffer is number one.

Institute Program

This command enables the user to designate a program in the buffer as the current Content Analyzer, Sequence Generator, or Sort Key extractor program.

Institute Program PROGNAME OK (as) type OK
where type is one of the following:
Content (analyzer)
Sort (key extractor)
Sequence (generator)
If no type is specified, Content analyzer will be assumed.

Instead of PROGNAME the user may specify the program to be instituted by number. The first program loaded into the buffer is number one.

If a program has already been instituted in that capacity, it will be deinstituted (but not removed from the buffer).

Deinstitute Program

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

Deinstitute type OK
where type is one of the following:
Content (analyzer)
Sort (key extractor)
Sequence (generator)

Assemble File

Files written in Tree-Meta can be assembled directly from the NLS source file with the Assemble File command. This aspect of NLS programming will not be described in this document.

Examples of User Programs

The following are examples of user programs which selectively edit statements in an NLS file on the basis of text matched against the pattern. For examples of L10 programming problems, you may find out how the standard NLS commands work by tracing them through, beginning with <NLS, SYNTAX, 2>. A table of contents to all the global NLS routines is available to the user in <NLS, SYSGD, 1>.

Example 1 -- Content Analyzer program

```
PROGRAM outname % removes the text and delimiters () of NLS
statement names in parentheses from the beginning of each
statement%
   DECLARE TEXT POINTER sf;
   (outname)PROCEDURE;
      IF FIND '( [']] ^sf THEN %found and set pointer after
      name%
         BEGIN
         %replace stmnt by everything after pointer%
            ST sf _ sf SE(sf);
         %display statement%
            RETURN(TRUE);
         END
      %otherwise don't display statement%
         ELSE RETURN(FALSE);
      END.
   FINISH
```

Example 2 -- Content Analyzer program

```
PROGRAM changed   %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.%
   (changed) PROCEDURE ;
      LOCAL TEXT POINTER pt ;
      %remember, CCPOS is initialized to the beginning of
      each new statement%
      IF FIND ^pt SINCE (25-JAN-72 12:00) THEN
         %the substring of zero length is replaced with
         "[CHANGED]"%
            ST pt pt _ "[CHANGED]";
      RETURN(FALSE) ;
      END.
   FINISH
```

PART FOUR: Interactive L10 Programming

Section 1: Introduction

For many programming applications, it is sufficient to accept statements one at a time from the sequence generator and assume as an initial character position the beginning of the statement (a Content Analyzer program as described above). For more complex applications, you may have to write programs which skip around files, between files, and interact with the user. These are not called by the sequence generator but "Attached" and then used like standard NLS subsystems, holding one or more commands. All the capabilities described above are available to such programs.

There are two parts to every user-attachable subsystem:

1) the L10 execution routines which do the file manipulations, and

2) the command syntax, specified in a language called Command Meta Language (CML), describing the user interface of each command in the user attachable subsystem.

These two parts are two separate programs, compiled separately into two REL files. The two programs are loaded in unison and together comprise the subsystem.

Like L10, source programs for the CML compiler are free form NLS files. Comments may be used wherever a blank is permitted and the structure of the source file is ignored by the compiler. CML source programs are compiled into REL files with the Compile File command in the PROGRAMS subsystem. CML is the compiler name for the CML compiler.

The REL file name of the CML code should have the extension "cml". The REL file name of the corresponding L10 execution procedures should have the same first name as the CML code file, and should have the extension "subsys." If these conventions are followed, the Load Program command in the PROGRAMS subsystem will automatically load both parts of the user subsystem and attach it, making it available for use. The user's subsystem may then be invoked by using the Goto or Execute commands.

The CML program describes the command words, noise words, selection requests, etc. that make up an NLS command. The CML code interacts with the user when he enters the subsystem and as he specifies commands. In the process of interacting with the user, the CML code may call one or a number of L10 execution procedures which "do the work."

CML automatically provides prompting, questionmark, and <CTRL-S> facilities. The CML syntax specification applies to both TNLS and DNLS (unless restricted by the programmer to one or the other), and will conform to all user options with respect to prompting and to recognition and completion modes.

The next section will describe CML, and how to design the user interface. Section 3 explains the requirements of the L10 procedures which CML calls. The remainder to Part Four discusses additional L10 capabilities useful in the context of attachable subsystems.

Section 2: Command Meta Language (CML)

Introduction

This section describes the Command Meta Language (CML). CML allows the specification of the user interface to commands. The CML program (the grammar) may call L10 procedures of a certain type (described in the next section). The programs written in CML are similar in structure to L10 programs. Typically, a CML and an L10 program are used in unison as a user attachable subsystem. A more technical presentation of CML may be found in <20438,>.

Program Structure

The basic CML program structure is much like that of L10 programs. The program begins with a "FILE" statement (as does an L10 program) of the form:

FILE name

where name is the name of the program code (in lowercase letters and numbers, beginning with a letter); it must be a unique symbol, different from the FILE name of the L10 code file.

The program ends with the statement (like L10):

```
FINISH
```

Within the program, one may have a series (in any order) of declarations, rules, and subsystems.

As in L10, all variables used in the program must be declared somewhere in the system. Other values and attributes must also be declared in CML.

Rules are defined sequences of the CML elements described below. Rule names can be placed anywhere in a CML command specification. When a rule is called within a command, it is almost as if the CML elements represented by that rule were inserted at that point in the command. This allows the definition of general interactions that may be of use in a number of commands or points in a command.

Each program usually represents one or more subsystems. A subsystem may include one or more commands. Each command is a rule itself. It may optionally include rules to be performed upon entering or leaving the subsystem. (One enters a subsystem with the Goto or Execute commands, and leaves with the Quit command.) A subsystem may also include general rules defined throughout the subsystem.

Each of these parts of the CML program will be described independently. The CML elements which make up rules will also be described.

Subsystems

A CML program holds declarations, general rules which apply throughout the program, and subsystems (usually only one).

The Subsystem begins with a statement of the form:

```
SUBSYSTEM name KEYWORD "NAME"
```

where name is the internal name of the subsystem (primarily for debugging purposes) and NAME is the name which the user must specify (in a Goto or Execute command) to access commands in the subsystem.

These two names may be the same but they must be unique, different from the FILE names of the CML and L10 files.

A subsystem ends with the statement:

```
END.
```

Within the subsystem, you may have any number of rules.

A rule as described below will be known throughout the subsystem, but not outside the subsystem.

A rule preceded by the word "COMMAND" will be available as a command in the subsystem. It should begin with a command word element. E.g.:

```
COMMAND zshow = "SHOW"!L2!
    ent _ ("EXAMPLE"/"SAMPLE")
    CONFIRM
    proc (ent) ;
```

A rule preceded by the word "INITIALIZATION" will be executed whenever the subsystem is entered (either with a Goto or an Execute command from another subsystem). E.g.:

```
INITIALIZATION example =
    proc1 (ent)
    proc2 (ent) ;
```

A rule preceded by the word "TERMINATION" will be executed whenever the subsystem is left (with a Goto or Quit command from this subsystem).

A rule preceded by the word "RENTRY" will be executed whenever the subsystem is reentered (either with a Quit command from another subsystem, having left this one with a Goto, or upon completing an Execute of a command in another subsystem from this subsystem).

Preceding a rule with the above modifiers does not prevent calling that rule from within another rule.

Rules

A CML rule is a defined series of elements, each of which represents one piece of the interaction with the user or system action. The elements will be described below. The name of a rule (defined to be the given series of CML elements) may be used in other rules. When the name of a rule appears in another rule, the CML code which it represents will be executed at that point.

A rule takes the form:

name = element1 element2 element3 ... element ;

where "name" is any unique name (lowercase letters and numbers, beginning with a letter).

Alternative elements (where the user has a choice) are indicated by a slash (/) in the expression. Parentheses should be used to group elements, particularly when alternative logic and nesting of alternatives is involved. E.g.

name = (element1 / element2 element3) element4 ;

Note that, by use of parentheses, an alternative may include more than one element.

Elements grouped in square brackets are options, and the user must type the option character <CTRL-u> to access them. E.g.

name = element1 [element2 element3] element4 ;

E.g.

```
zinsert = "INSERT" ent_("WORD"/"CHARACTER") <"at">
dest_DSEL(ent) xins(dest);
```

A number of elements may be included in a single rule. (If you exceed the maximum, you will get a "stack overflow" error at run-time.) Elements are NOT separated by any delimiter character (except by spaces or the source file structure). The entire rule is terminated by a semicolon.

The return value of elements may be assigned to CML variables (single-word as in L10), using a left-arrow (_) in the form:

variable _ element

The variable must have been declared, as described below. A variable must be initialized by such an assignment before its content is passed to any routine. It must be initialized in the rule which passes it to a routine (not just in other rules called from the given rule, even though other rules may subsequently set it to another value). (If you fail to do so, you will get the run-time error "reference to undefined interpreter variable.")

Names on the left side of an assignment are assumed to be variables; other names in CML rules are assumed to be CML rules.

Declarations

Declarations are used to associate names with their CML function. A number of types of names may be used in CML

programs.

Variables

Whenever a procedure is called from CML, CML creates a ten-word record. The address of the record is passed to the procedure, which may put information in any of the ten words. The procedure usually returns the address of its record.

A CML variable is a cell which holds the address of a CML record. By this mechanism, up to ten words of information may be handled with a single parameter by passing the address of the first word of the record. A variable may be declared with the statement:

```
DECLARE VARIABLE name ;
   or
DECLARE name ;
```

where "name" is any unique name (lowercase letters and numbers, beginning with a letter).

You may declare any number of variables in a single statement, i.e.:

```
DECLARE VARIABLE name1, name2,... ;
   or
DECLARE name1, name2,... ;
```

Many CML variables have been declared for use anywhere in the system, and may be used freely in user attachable subsystems (without being declared by the user programmer). Some commonly used variable names are:

ent	namfil	level	param
dent	dest	filtre	param2
sent	source	vs	param3
port	fromwhom	literal	param4

External Variables

As in L10, external variables are variables which are made available to any procedure anywhere in the NLS system. (Simple variables are only known in the file in which they are declared.) One or more may be declared with a statement of the form:

```
DECLARE EXTERNAL name1, name2,... ;
```

Parsefunctions

An L10 function which processes input and supplies a prompt string is called a "parsefunction." The name of the procedure must be declared as a parsefunction for CML to request a prompt string whenever the procedure is called.

```
DECLARE PARSEFUNCTION name1, name2,... ;
```

More detailed information about the nature of parsefunctions will be offered below.

Command Words

A command word is a word specified as part of a command (e.g. "Insert" or "Word" in the Insert Word command); it is specified in accordance with each user's recognition scheme (often recognized after the first character). A declaration may assign a value to a command word, to be passed to an L10 procedure which needs to know which command word was chosen by the user.

```
DECLARE COMMAND WORD "WORD1"=100, "WORD2"=101,... ;
```

The value must be a positive decimal integer, less than 511. (This limit may have to be changed to 255 in

future versions of NLS.)
More than one command word may have the same value (unless of course the L10 procedure must distinguish the user's choice between the two).

A command word that has not been declared may be included in the syntax; it will have no value though. Only those command words which are assigned a value and then passed to an L10 procedure must be declared. Many command words have been declared for use in the NLS system. It is considered good practice to use command words already known to users when possible, and to use the same values for those words as declared in NLS. Section 5 offers a set of declarations, including all the system defined command words; it can be copied as the foundation for a CML program.

You may not use command words identical to the names of the L10 or CML files, to the name of the subsystem, nor to any variable names.

CML Elements

The CML elements described here are the building blocks of rules, which describe interactions with the user.

Command Word Recognition

The appearance of a command word element in a rule means that the user must specify that (or an alternative command word) at that point in the command specification.

In the CML description, each command word is represented by its full text. The algorithm used to match a user's typed input against any list of alternative command words is known as "recognition." Each individual's command word recognition mode will determine what characters the user must type to specify the command word. This is handled automatically by the command interpreter.

As the user specifies a command, the command words (and noise words described below) are echoed in a line at the top of the DNLS screen, or printed in TNLS. This is called the "command feedback line."

Command word elements must be uppercase words enclosed in double-quotes (""); e.g.

"INSERT"

Command words optionally may be followed by one or more qualifiers which modify the recognition process, separated by spaces and enclosed in exclamation points. The qualifiers are:

NOTT -- not available in TNLS
NOTD -- not available in DNLS
L2 -- second level (some recognition modes differentiate first from second level command words, e.g. second level are preceded by a space)
number -- explicit value for command word; supercedes any value assigned by a DECLARE COMMAND WORD

For example:

"SET"!L2!
"PRINT"!NOTD!
"EXAMPLEWORD"!L2 104!

The address of records holding declared command word values

may be assigned to CML variables so that the user's choice can be passed to subsequent routines, e.g.

```
ent _ "CHARACTER"
   or
ent _ ("CHARACTER" / "WORD")
   then
xprocedure (ent)
```

Remember that, like all other CML assignments, the variable receives the address of a record which holds the information. When the content of this variable (the address of the record) is passed to a procedure, the procedure must REF its receiving variable to access the contents of the record, the value.

This value will be assigned as above even if the command word is followed by ather CML elements; e.g.

```
ent _ ("CHARACTER" param_FALSE / "WORD" <"at">
param_LSEL(#"WORD") )
```

ent will get the value of the command word CHARACTER or the value of the command word WORD. The appropriate actions will happen after the user chooses the command word.

You may wish to pass this value without forcing the user to type the command word. This address may be assigned by preceding the command word by a pound-sign (#).

```
ent _ #"CHARACTER"
```

will assign the address of the declared command word value without forcing the user to type the command word

Selection Recognition

Selections are input from users pointing to places in files or typing in strings of text. The three types of selection routines available in CML, with their respective command prompts, are:

```
DSEL -- destination selection
   B/A
SSEL -- source selection
   B/A/[T]
LSEL -- literal selection
   B/T/[A]
```

where B = bug (not available in TNLS), A = Dynamic Address Element (any series of NLS addressing elements), and T = typein from keyboard.

Each of these predefined selection routines prompts the user and receives the input.

The selection routines must be passed the address of a record holding the value of a noun command word (character, word, statement, plex, etc.). The command word enclosed in double-quotes and preceded by a pound-sign (#) is equivalent to the address of a record holding the declared value of that command word, e.g.:

```
DSEL(#"CHARACTER")
```

Or you may have assigned the address of the value of a previously selected command word to a CML variable, then pass the content of the variable, e.g.:

```
ent _ "CHARACTER"
DSEL(ent)
```

CML will prompt the user for the appropriate input. If

more than one selection is necessary (e.g. to specify both ends of a group or string of text), they will prompt for both automatically. They will delimit the appropriate entity automatically (e.g. both ends of a word will be found from a single selection).
The routine will return the address of a CML record holding two text pointers in the first four words, delimiting the beginning and end of the entity selected.

for string entities within statements
words 1-2: txt ptr before first character of string
words 3-4: txt ptr after last character of string
for types "STATEMENT" and "BRANCH"
words 1-2: txt ptr before first character of statement
words 3-4: txt ptr after last character of statement
for types "GROUP" and "PLEX"
words 1-2: txt ptr before first character of first statement
words 3-4: txt ptr before first character of last statement
for type "WINDOW"
word 1: address of display area
word 2: x and y screen coordinates

One usually assigns the returned address of this record to a CML variable, e.g.:

```
dest _ DSEL(#"STATEMENT")
```

Other Recognizers

Other prespecified input routines are available, each prompting for and receiving a type of input from the user:

VIEWSPECS -- takes no argument and returns the address of a CML record holding:
word 1: updated viewspec word 1
word 2: updated viewspec word 2
words 3-7: used for collecting characters from user

LEVADJ -- takes no argument and returns the address of a CML record holding:
word 1: level adjust count
(up = +1, same = 0, down = -1, up two levels = +2, etc.)
words 2-7: used for collecting characters from user

CONFIRM -- waits for user to type confirmation character (a Command Accept, Insert, or Repeat character); it takes no argument and returns the address of a CML record holding the confirmation code in word 1.
These values are rarely used, since subsequent functions are handled automatically by the command parser. For reference, they are:
1 = Command Accept
2 = Insert
3 = Repeat

DUMMY -- does nothing but always TRUE; may be used to allow elements to be skiped, e.g.:

```
( "OPTION" somprocedure() / DUMMY ) CONFIRM
```

allows the user to specify "Option" before the CONFIRM, or skip it and just type a CONFIRM.

CML Constants

TRUE -- holds the address of a CML record whose first word has the value TRUE (i.e. 1)
FALSE -- holds the address of a CML record whose first word has the value FALSE (i.e. 0)

L10 Procedure Calls

L10 procedures may be called at any point in the rule by including the name of some routine followed by its parameter list enclosed in parentheses. (The next section describes the special requirements of L10 procedures called from CML.) E.g.

```
procedurename (param1, param2,...)
```

Parameters may include CML variables (whose content is passed), the CML elements TRUE, FALSE or NULL, or the # construct (see "Selection Recognition") representing the address of a command word value.

Helpful Procedures in building CML logic:

isdnls() -- returns TRUE if DNLS, else FALSE
istnls() -- returns TRUE if TNLS, else FALSE
true() -- returns TRUE
false() -- returns FALSE
abort() -- abort command as if user typed a Command Delete

Parsefunctions

Procedures which are declared as PARSEFUNCTIONs examine the information being typed by the user during command specification (characters going into the input buffer). CML places additional requirements on L10 procedures declared as parsefunctions, as described in the next section. They may be called from CML like any other L10 procedure. The following parsefunctions are available as part of the running system; they of course must be declared as parsefunctions in any program which uses them as such:

answ() -- if the next character in the input buffer is a CONFIRM, option character, or the letter "y", it reads the character (out of the input buffer) and returns TRUE; else it reads the next character and returns FALSE

answer() -- reads next character; like answ, but returns the address of a CML record whose first word holds either the value TRUE (1) or the value FALSE(0)
lookansw() -- if next character is a CONFIRM, option character, or the letter "y", returns TRUE and leaves next character in buffer; else returns FALSE and reads character
mylookansw() -- if next character is a CONFIRM, option character, or the letter "y", returns TRUE; else returns FALSE; leaves next character in buffer
readconfirm() -- if next character a CONFIRM character, reads and returns TRUE; else leaves character in buffer and returns FALSE
lookconfirm() -- if next character is a CONFIRM, returns TRUE; else returns FALSE; leaves next character in buffer

readbug() -- if next character a Command Accept character, reads and returns TRUE; else leaves character in buffer and returns FALSE
lookbug() -- if next character is a Command Accept, returns TRUE; else returns FALSE; leaves next character in buffer
notca() -- if next character NOT a Command Accept character, reads and returns TRUE; else leaves Command Accept character in buffer and returns FALSE
readoption() -- if next character an option character, reads and returns TRUE; else leaves character in buffer and returns FALSE
readrepeat() -- if next character a repeat character, reads and returns TRUE; else leaves character in buffer and returns FALSE
lookrpt() -- if next character is a REPEAT, returns TRUE; else returns FALSE; leaves next character in buffer
sp() -- if next character a space, reads and returns TRUE; else leaves character in buffer and returns FALSE
lookback() -- if next character is a back-arrow (_), returns TRUE; else returns FALSE; leaves next character in buffer
looknum() -- if next character is a digit, returns TRUE; else returns FALSE; leaves next character in buffer

Parsefunctions may appear as alternatives to recognizers. However, they must precede any non-failing recognizers in the list of alternatives. E.g.:

```
( lookconfirm() / "APPEND" / "FILE" ) CONFIRM
-- this example either will accept a CONFIRM or will
   accept a specification of the command word APPEND or
   FILE followed by a CONFIRM.
```

Feedback

Noise words between command words are very helpful to the user learning a new command. Any string of text may be added to the command feedback line by enclosing the text in parentheses and within angle-brackets in a rule. E.g.

```
<"Text of noise words">
```

The last noise word string on the command feedback line (in DNLS) may be replaced with a new string by placing three dots before the first double-quote, e.g.:

```
<..."new noise words">
```

The last noise word string can be erased (in DNLS) with the procedure call:

```
clearname()
```

The entire command feedback line can be cleared (in DNLS) with the CML element:

```
CLEAR
```

A few characters of the noise word will follow the command word in the system's response to a questionmark if:

1) the noise word immediately follows the command word, and
2) the command word is not being assigned to a variable (it may however be part of a list of alternatives being assigned).

E.g. the noise words in the CML below will show in the

systems response to a questionmark:

```
ent _ ("FILE" <"name"> / "STATEMENT" <"at"> )
```

Loops

A looping facility permits repetition of a different rule until an exit condition is met. The rule is evaluated and then the expression following the UNTIL keyword is evaluated. If the expression returns TRUE, then the loop is exited and the next element of the rule is evaluated. If the expression returns FALSE, then the named rule is invoked once again.

```
PERFORM rulename UNTIL ( exp )
```

where rulename is the name of the rule to be repeatedly executed and exp is an expression of CML elements which evaluates to TRUE or FALSE.

E.g.

```
PERFORM rulename UNTIL ( <"Finished?"> answ() )
```

Nested loops (loops within rules called by a PERFORM element) are not currently allowed. Backspacing through executed loops requires special treatment not described here.

Sample CML Program

The following sample program should help illustrate the use of the CML language for describing NLS commands. For more exhaustive examples, look at the CML specification for the standard NLS commands, in <NLS,SYNTAX,>. An example of a problem treatment can often be found by thinking of an NLS command which is similar.

```
FILE sampleprogram   % <CML,> to <sample.rel,> %
   DECLARE what, whom, where ;
   DECLARE COMMAND WORD
      "GLUE" = 1,
      "PASTE" = 2,
      "CRAYONS" = 3,
      "PENS" = 4,
      "PENCILS" = 5 ;
   SUBSYSTEM sample KEYWORD "SAMPLE"
      objects =
            "GLUE"
         / "PASTE"
         / writingthings ;
      writingthings =
            "CRAYONS"
         / "PENS"
         / "PENCILS"!L2! ;
      COMMAND zuse = "USE"
         what _ writingthings
         CLEAR
         <"to draw a pretty"> whom _
            ( "PICTURE" <"of Aunt Mary">
            / "SKETCH" <"of your dog">
            )
         CONFIRM
         % call execution routine process the USE command %
            xuse( what, whom) ;
      COMMAND ztake =  "TAKE"
         what _ objects
```

```
            <"out of your">
               where _ ("EARS"!1! / "NOSE"!2! / "MOUTH"!3!)
               <"PLEASE!!">
            CONFIRM
            xtake (what, where) ;
         END.
      FINISH
```

Given this sample CML, the user might specify the command:

```
"Use Pens
 (to draw pretty) Sketch (of your dog) <OK>"
"Take Crayons (out of your) Mouth (PLEASE!!) <OK>"
```

The execution routines called from CML typically have names beginning with the letter "x".

Section 3: L10 Execution Procedures

The CML program interacts with the user and gathers information; it subsequently calls one or more L10 procedures. The procedure CML calls must meet certain requirements, described in this section. Because of these requirements, typically the execution routine is written as an interface to a number of other L10 procedures performing the actual functions. This way the function routines can be written independent of which command or procedure calls them. This section will describe the requirements of procedures called from CML. The next section offers additional L10 capabilities in this environment.

CML can be in one of four states as it parses a command based on the syntax described in your CML program (known as the "parsemode"):

1) parsing: recognition state where input text is compared with grammatical constructs in CML program

2) backup: the user has typed a backspace, or a procedure call has returned FALSE; CML backs up through previously specified elements of the CML code, calling each in backup mode, to before the last CML alternative (not necessarily equivalent to user input element; maybe through the entire command, aborting the command)

3) cleanup: the user has typed a Command Delete, or the command has been completed (including any execution procedure calls); CML backs up through all previously specified elements of the CML code; each procedure is again called, this time in "cleanup" mode

4) parsehelp: (used only with parsefunctions) before calling a parsefunction in "parsing" mode, the procedure is called in "parsehelp" mode to solicit a user prompt string.

5) parseqmark: (used only with parsefunctions) when the user types a questionmark, the procedure is called in "parseqmark" mode to solicit a questionmark string.

When CML calls a procedure, it automatically passes two extra implicit parameters before the parameters the programmmer specifies:

The first parameter is the address of a CML record reserved for use by that procedure. The record is initially empty (or filled with garbage). The execution procedure may fill the ten words of the record by receiving the address in a REFed parameter variable and then indexing into the array.

CML considers the procedure to have returned TRUE if it returns the address of the CML record; otherwise the return

is considered FALSE. When a procedure returns FALSE, CML will back up, calling that and previous procedures in "backup" mode, until another branch in the command syntax logic is found or until the entire command has been aborted.

The second parameter is a value (not an address of a record) representing the parse mode. Whenever CML encounters a procedure call in the syntax (in any mode) it calls the procedure, passing it the value of the parsemode.

Typically, the execution routine should only perform its primary function in the parsemode "parsing". In "backup" and "cleanup", it may reset any globals or state information it may have affected while in the parsemode "parsing." The names of the modes (see above) are globals to which you may compare the value received in the second parameter. An execution routine typically consists of a large CASE statement, e.g.

```
CASE parsemode OF
   = parsing:
      BEGIN
        •
        •
      END;
   = backup, = cleanup:
      BEGIN
        •
        •
      END;
   ENDCASE ;
```

Calls on procedures declared as parsefunctions pass a third implicit parameter, the address of a string in which to put the prompt. They are called in the parsemode "parsehelp" for the string before being called in the parsemode "parsing", or in parsemode "parseqmark" when the user types a questionmark. CML passes the parameters specified in the call after the two or three system supplied parameters. Remember that these parameters will always be the address of a record holding the information, so the receiving variable must be REFed. The format of the record itself is determined by the routine that filled it.

For example, if the CML procedure call looked as follows:

```
xprocedure (param1, param2)
```

then the L10 execution procedure would receive parameters as follows:

```
(xprocedure) PROCEDURE (result, parsemode, parameter1,
parameter2) ;
```

All parameters except the parsemode should be REFed in the execution procedure.

Section 4: Additional L10 Capabilities

Introduction

The attachable subsystems have access to the full capabilities of the NLS environment. This section will describe some capabilities not discussed in the context of Content Analyzer programs. Further capabilities will be discussed in Part Five.

Moving Around Within NLS Files

Generally, at least one simple variable or a text pointer will have to be declared to hold the statement identifier (stid) of the current statement. (The first word of a text pointer is an stid.) Assume the simple variable with the name "stid" has been declared for the purpose of the following discussion.
In the NLS file system, two basic pointers are kept with each statement: to the substatement and to the successor.

If there is no substatement, the substatement-pointer will point to the statement itself.

The procedure getsub returns the stid of the substatement. To do something to the substatement if there is one:

```
IF (stid := getsub(stid)) # stid THEN something..;
```

stid is given the value of the substatement-pointer, then the old value of stid is compared to the new. If they are the same, then there is no substructure. If they are different, you have the stid of the substatement and can operate on it.

If there is no successor (at the tail of a plex), the successor-pointer will point to the statement UP from the statement (i.e. the statement to which the current statement is a sub).

The procedure getsuc returns the stid of the successor (or up).

To move to the successor:

```
stid _ getsuc(stid);
```

Given these two basic procedures, a number of other procedures have been written and are part of the NLS system. All of the following procedures take an stid as their only parameter, and do nothing but return a value, usually a stid. If the end of the file is encountered, these procedures return the global value "endfil".

getup(stid) -- returns the stid of the up
getprd(stid) -- returns stid of the predecessor
getnxt(stid) -- returns stid of next statement or endfil
getbck(stid) -- returns the stid of the back or endfil
gethed(stid) -- returns stid of the head of the plex
getail(stid) -- returns stid of the tail of the plex
getend(stid) -- returns the stid of the end of the tail of the plex
getftl(stid) -- returns TRUE if stid is tail of plex, else FALSE
getlev(stid) -- returns level of statement

Once you have the stid of a statement, you may operate on it as in Content Analyzer programs. E.g.

```
FIND SF(stid) $NP ^ptr...
```

Another common operation is to access the statement (file) in which the CM (or bug) was at the time of the last Command Accept (or other command terminator). This is stored in the system, and can be accessed with the following procedure call:

```
stid _ lccsp() ;
```

Then, if you wish to set the stpsid to the origin of that file, you could say:

```
stid.stpsid _ origin ; %origin is a global with the
stpsid of the origin statement in it%
```

The following procedures may also assist you in moving around

files:
caddexp(aptr1,aptr2,da,startptr) -- given the addresses of two text pointers surrounding an NLS address expression, the address of a display area, and the address of a text pointer representing the starting position: caddexp will evaluate the address expression with respect to the starting position, and update the start pointer to the new location.
This procedure will follow file returns, links, etc., opening files as necessary. Remember to close any open files when you are done with them (see 6d4 below).
The procedure lda() returns the address of the display area which held the bug at the time of the last Command Accept; it may be used as the third parameter of caddexp. E.g.
caddexp($ptr1, $ptr2, lda(), $sptr) ;
namingrp(stid1,stid2,astring,levels) -- given two stids representing a group, the address of a string holding the name, and a number representing levels of depth below the stids': returns stid of the statement with the given statement name in the group specified by the stids. Only searches through given number of levels below stid level. (If the stids are the same, will search the branch.)
lookup(ptr,string,type) -- given the address of a text pointer, the address of a string, and a type, will do a variety of searches (in the process destroys string and changes pointer). type may be one of the following:
nametyp -- non-sequential search for statement of name given in string; returns stid and sets pointer to stid or else returns endfil in both places
nxtname -- like name, also a non-sequential search, but starts from place in file ring to which ptr points
seqname -- starting with the statement following the one refered to by the ptr, does a sequential search of the file for the given name; returns stid or endfil in pointer
contnt -- does a sequential search of the file, beginning with the character following the pointer, for a statement with the content of the string; returns stid or endfil in pointer
contls -- same as contnt, but looks only in statement holding pointer
wordtyp -- same as contnt, but looks for word given in string
sid -- pass an SID instead of the address of the string; searches for statement with that SID and returns in pointer and as procedure value the stid or endfil

Calling NLS Commands
A program may execute any of the standard NLS commands by calling the same procedure that the system execution routines call for each command. These procedures are called the "core" procedures. They are listed in <NLS,XPROCS,> and in <NLS,SYSGD,>. Their names begin with the letter "c", generally followed by three initials of each command word, e.g. Insert Statement could be executed by calling the procedure "cinssta".

Usually the required arguments can be discovered by knowing the command and by looking at XPROCS and/or SYSGD. For example, the formal parameters to the procedure "cinssta" are (stid,rlevcnt,tp1,tp2). As one might guess from the command syntax, the procedure wants a target stid, the value of level adjustment (up = +1, same = 0, down = -1, etc), and the address of two text pointers surrounding the string of text to be inserted.

Much can be learned by looking at the code of the core procedure. You can see what procedures it in turn calls to discover how the command is actually performed. But most importantly, you can find out what the procedure returns. The RETURN statement for "cinssta" look like:

```
RETURN(stid);
```

from which it can be inferred that the procedure returns the stid of the newly created statement.

When you are not sure what the arguments mean, a good way to find out is to see where the command parser picks up the information. You can follow through the parsing of a command by beginning with <NLS,SYNTAX,>, the actual NLS CML code. Tracing a command from <NLS,SYNTAX,> is also valuable in finding out how the system performs an operation which you would like your program to do. For example, if you wish to parse a link and open the given file, you might learn how to do it by following the Jump to Link command through.

Opening Files

When you ask the user for an address or bug, you don't have to open the file; you have a handle on it with the stid the user gives you. There may be times, however, when you wish your program to open a file not specified by the user. There is a procedure which does this:

```
open (jfn, astring);
```

You should pass zero as the jfn, and the address of a string containing the name of the file to astring. This procedure will return the file number. If the file is not already open, it will open it. It will also fill out the string with the complete file name if you do not specify the directory or version number.

If the file does not exist, open calls the procecure "err", which generates a signal of the value "errsig." Signals are discussed in Part Five.

The usual sequence of steps to open a file is as follows:

```
%"stid" has been declared as a simple variable or text
pointer%
stid _ orgstid; %orgstid is a global with all zeros except
in the stpsid field, where it has the stpsid of the origin
statement (the same for every file)%
*str* _ "<dirname>filename.nls"; %str is of course a
declared string variable%
stid.stfile _ open (0,$str);
```

Note that the procedure "open" requires a TENEX file name.

The procedure "lnbfls" converts links to TENEX file names:

```
lnbfls (linkstr, linkparseblock, filenamestr)
```

Pass the address of the string holding the link as the first parameter, zero for the second parameter (used if link already parsed), and the address of a string to

receive the filename as the third parameter.
The procedure returns the host number in case the link includes a site name. This value might be compared to the following globals:

lhostn -- the number of the local host
utilhost -- the number of Office-1
archost -- the number of the ARC machine (BBN-TENEX-B)

For example, you might use the procedure as follows:

```
CASE lnbfls(&linkstr,0,$filename) OF
   = lhostn: NULL;
   ENDCASE err(notyet) ;
```

At the end of your program, you should close any files that you have opened. Use the procedure:

```
close (filnum);
```

E.g.

```
close (stid.stfile);
```

Displaying Messages

The following procedures may be of use in displaying messages. In all cases, the appropriate actions will occur in TNLS as well as DNLS, although these descriptions are oriented to DNLS.

dismes(type, astring) -- teletype window
where type is one of the following:
0 -- clear teletype window (no address need be passed)
1 -- add text in string whose address is passed as a new line in the teletype window
2 -- add text in string whose address is passed as a new line in the teletype window for about 3 seconds, then clear window
n -- any number >=1000 represents the number of milliseconds the message is to be displayed before the teletype window is cleared.
In TNLS, type = 1, 2, and >=1000 all simply print the string starting on a new line.

fbctl(type, astring) -- literal display window
where type is one of the following:
typenulllit -- begin empty literal display (replacing file window), no string address passed
fbaddlit -- add string whose address is passed to current literal display
addcalit -- add "Type <CA> to continue." to current literal display, then wait for <CA> or <CD>, then restore file window
typelit -- start literal display with string, then wait for user input, then restore file window
fbendlit -- add string to current literal display, then wait for user input, then restore file window
typecalit -- start literal display with string, add "Type <CA> to continue.", then wait for <CA> or <CD>, then restore file window
The literal display replaces the file window on the screen, or is simply printed in TNLS. For example, it is used by the Show File Status command.

dn(astring) -- name display
add string whose address is passed to command feedback

line, enclosed in quotes

Setting Up for Display Refreshing

The command parser calls the procedure "cmdfinish" after completing and cleaning up every command. If certain parameters are set properly, "cmdfinish" will automatically update the user's screen (primarily of concern in DNLS). You may also move a different statement to the top of the window (i.e. jump) before updating the screen.

To refresh the screen after editing a file:

The procedure "dpset" sets up parameters for refreshing the screen after a command. If "dpset" is properly used, the screen will automatically be refreshed after the command. One should look for the most efficient way to make the proper changes.

The procedure "dpset" must be called BEFORE any changes are made in the file. This is so that the display reformatter will have something with which to compare when looking to see what has been changed.

The procedure call should look as follows:

```
dpset ( type, stid1, stid2, stopstid ) ;
```

There are a number of globals which may be passed for "type":

dsprfmt -- rewrite the content of one or two statements

- stid1 -- the stid of the statement that has been changed
- stid2 -- the stid of another statement that has been changed, or "endfil"
- stopstid -- ignored, pass it "endfil"

dspstrc -- if file restructuring occured beginning at at one or two places; doesn't rewrite content of statements; will add new statements in a structure

- stid1 -- the stid of the statement where a structural change begins
- stid2 -- the stid of where another structural change begins, or "endfil"
- stopstid -- the stid of the statement after which it can stop changing the screen (whether change began with stid1 or stid2); the procedure "dpstp" may be of service here; if you cannot figure out where it should stop, pass it "endfil" (go till end of window)

dsprfst -- rewrites content of one or two statements, then looks for structural changes thereafter

- stid1 -- the stid of the statement where a set of changes begins
- stid2 -- the stid of where another set of changes begins, or "endfil"
- stopstid -- the stid of the statement after which it can stop changing the screen (whether change began with stid1 or stid2); the procedure "dpstp" may be of service here; if you cannot figure out where it should stop, pass it "endfil" (go till end of window)

dspjpf -- jump command in one window only, no editing

- stid1 -- the stid of the statement to be at the

top of the screen; see below for other parameters which must be set
stid2 --"endfil"
stopstid -- "endfil"

dspyes -- completely refresh all windows holding one or either of two files specified
stid1 -- the stid of a statement in the file where changes will be made
stid2 -- the stid of a statement in the file where another set of changes will be made, or "endfil"
stopstid -- "endfil"

dspno -- do no display refreshing
stid1 -- "endfil"
stid2 -- "endfil"
stopstid -- "endfil"

dspallf -- refresh the entire screen
stid1 -- "endfil"
stid2 -- "endfil"
stopstid -- "endfil"

The procedure "dpstp", when passed an stid, returns the stid of the next statement in the file at the same or a higher level. This can be used as the stopstid in "dpset" if structural changes are occuring such that you don't know a priori what the last statement changed will be.

To change the position of a window (jump):

The global "cspupdate" should be set to the address of the display area descriptor for the window you want changed.

In TNLS, it is always the address contained in the global "tda".

If you wish to change the view in the window which held the bug at the time of the last CONFIRM, you may use the statement:

```
cspupdate _ lda();
```

This also works for TNLS.

Once cspupdate is set, any of the globals described below will replace the appropriate field in the display area descriptor upon completion of the command.

The global "curmkr" is a text pointer pointing tc the statement at the top of a window in DNLS, or the CM in TNLS.

The first word of "curmkr" should be set to the stid of the statement you want at the top of the window (in TNLS the statement which you want to hold the CM).

The second word of "curmkr", i.e. curmkr[1], should hold the character position for the CM. (In DNLS it is usually 1.)

The global "cspvs" is a two word array which should hold two viewspec words for the new view.

The global stdvsp is a two work array holding the NLS standard viewspecs (i.e. the ones in effect when you first enter NLS).

The current viewspec words may be gotten from the display area descriptor. If you have REFed a variable called "da", for example, you may assign the address of the display area which held the cursor at the time of the last command Accept with the statement:

```
        &da _ lda() ; %return address of display area
        descriptor%
    You may then refer to fields within the display area
    descriptor.
        davspec -- holds the first viewspec word
        davspc2 -- holds the second viewspec word
    You may change individual fields within viewspec words.
    The following fields apply to viewspec words:
        vslev -- lowest level to be displayed
        vsrlev -- if set to TRUE, the level of the current
        statement will be added to vslev
        vslevd -- if set to TRUE and vsrlev is TRUE, the
        current level will be subtracted from rather than
        added to vslev
        vstrnc -- number of lines of each statement to be
        displayed
        vscapf -- if TRUE, content analyzer on (viewspec i);
        takes precedence over vscakf
        vscakf -- if TRUE, content analyzer on until one
        statement passes (viewspec i)
        vsusqf -- if TRUE, user sequence generator on
        (viewspec O)
        vsbrof -- if TRUE, branch only on (viewspec g); takes
        precedence over vsplxf
        vsplxf -- if TRUE, plex only on (viewspec l)
        vsblkf -- if TRUE, blank lines on (viewspec y)
        vsindf -- if TRUE, indenting on (viewspec A; on by
        default)
        vsrind -- if TRUE, indenting relative to first
        statement in display (viewspec Q)
        vsnamf -- if TRUE, statement names on (viewspec C; on
        by default)
        vsstnf -- if TRUE, statement numbers or SIDs on
        (viewspec m)
        vsstnr -- if TRUE, statement numbers/SIDs put on
        right (viewspec G)
        vssidf -- if TRUE, SIDs replace statement numbers
        (viewspec I)
        vsidtf -- if TRUE, statement signatures on (viewspec
        K)
        vsfrzf -- if TRUE, frozen statements on (viewspec o)
        vspagf -- if TRUE, pagination on in TNLS (viewspec E;
        on by default)
        vsdaft -- if TRUE, don't defer display recreation in
        DNLS (viewspec u; on by default)
If you wish, you may set the variable "cspcacod" to the
address of a user content analyzer procedure, and/or the
variable "cspusqcod" to the address of a user sequence
generator procedure; they will be instituted before the
window is updated.
    The following fields in the display area descriptor may
    be useful:
        dacacode -- holds address of currently instituted
        Content Analyzer procedure
        dausqcod -- holds address of currently instituted
        user Sequence Generator procedure
```

If you have a REFed variable called "da", will not edit the file, and do not wish to change the viewspecs, you might use the following sequence of commands:

```
%address of last display area%
   &da _ cspupdate _ lda();
%stid of stmnt to be put at top of window%
   curmkr _ stid ;
   curmkr[1] _ 1 ;
%two current viewspec words%
   cspvs _ da.davspec;
   cspvs[1] _ da.davspc2;
%turn on Content Analyzer%
   cspvs.vscapf _ TRUE;
%institute the procedure "filterproc" as Content
Analyzer%
   cspcacod _ $filterproc;
%set up for display recreation%
   dpset (dspjpf, curmkr, endfil, endfil);
```

If you have edited the file, use the type "dspyes" instead of "dspjpf" in your call on "dpset".

Other Useful Procedures

astruc(astring) -- given the address of a string, sets the string to upper case.

fechno(stid,astring) -- given an stid, appends the statement number string to the string variable whose address is passed.

getsid(stid) -- given an stid, returns value of SID (don't forget to add zero to front if converting to a string)

fechsig(stid,astring) -- given an stid, appends the statement signature to the string variable whose address is passed.

getdat(astring) -- given the address of a string, appends date and time to string.

grptst(stid1,stid2) -- checks that two stid's specify a legal group; returns them ordered or else an "illegal group" signal is generated.

plxset(stid) -- given an stid, returns the stid of the head and of the tail of the plex of which the passed stid is a member; e.g. first _ plxset(stid : last) ;

resetf(fileno) -- given the file number of and open file, deletes all contents of the file leaving only origin statement, resets date and ident in origin statement (leaves file locked)

filnam(filno,astring) -- given the file number, appends the file name (in link format surrounded by angle-brackets <>) to string whose address is passed

pause(milliseconds) -- waits the given number of milliseconds, then returns

settimer(milliseconds,aproc,param1,param2,param3,param4) -- calls procedure whose address is passed, passing up to four parameters to that procedure, after given number of milliseconds; other code will be executed in the mean time

Globals of Interest:

initsr -- is the login ident of the person currently using the program.

inptrf -- is incremented every time the user types a <CTRL-o>; this can be used as a user program interrupt mechanism; i.e. you can set it to 0 at the beginning of the program and then

check it at the start of each loop of your program to see if the user has typed a <CTRL-o>, i.e. wishes to abort the command.

inpstp -- is incremented every time the user types a <CTRL-s>.

Section 5: Creating and Using Attachable Subsystems

In summary, the programmer must write two programs to build a user attachable subsystem: the CML and the L10 support procedures. Each of these programs is compiled separately (by their respective compilers) into separate REL files. The Load Program command (in the PROGRAMS subystem) will load both at once if the extension on the filename holding the CML code is "cml" and the extension on the L10 code file is "subsys". Once loaded, the user may use commands in the subsystem as he does commands in any of the standard subsystems.

You may find it convenient to begin writing a program by copying the following skelton (plex) from this NLS file <USERGUIDES,L10-GUIDE,6e2a>. It can then be modified to fit the needs of your program. (The comments in the FILE statements allow you to quickly bug the information required by the Compile File command. All the CML declarations that are used in the NLS system are included only to contribute to consistent use of command words and values. The CML rules have been left blank; they must be filled in or removed. All file, procedure, subsystem, and rule names are only exemplary. The last three parameters in the L10 procedure are only exemplary.)

```
FILE cname % (CML.SAV,) TO (cname.cml,) %
   % DECLARATIONS %
      DECLARE PARSEFUNCTION
         answ,          % reads answer construct %
         answer,        % for questions - returns  0/1 %
         sp,            % reads next char, TRUE if space %
         readconfirm,   % reads next char if ca %
         readbug,       % reads next char if BUG %
         readoption,    % TRUE if next char is optchar %
         readrepeat,    % TRUE if next char is repeat %
         lookansw,      % TRUE if next char is Y/CA %
         lookconfirm,   % TRUE if next char is CA/REPEAT/INSERT
         %
         lookbug,       % TRUE if next char is BUG %
         looknum,       % TRUE if next char is a number %
         clearname,     % clears the name area %
         notca;         % reads next char, TRUE if not CA char %

      DECLARE COMMAND WORD
         "BRANCH" = 1 ,
         "GROUP" = 2 ,
         "PLEX" = 3 ,
         "STATEMENT" = 4 ,
         "CHARACTER" = 5 ,
         "CONTROLCHAR" = 6 ,
         "INVISIBLE" = 7 ,
         "LINK" = 8 ,
         "DIRECTORY" = 9 ,
         "PASSWORD" = 10 ,
         "NUMBER" = 11 ,
         "TEXT" = 12 ,
```

```
"VISIBLE" = 13 ,
"WORD" = 14 ,
"FILE" = 15 ,
"NEWFILELINK" = 16 ,
"OLDFILELINK" = 17 ,
"NAME" = 18 ,
"IDENT" = 19 ,
"IDENTLIST" = 20 ,
"EDGE" = 21 ,
"MARKER" = 22 ,
"NLS" = 23 ,
"ITEM" = 24 ,
"ITEMNOVS" = 25 ,
"SUCCESSOR" = 26 ,
"PREDECESSOR" = 27 ,
"UP" = 28 ,
"DOWN" = 29 ,
"HEAD" = 30 ,
"TAIL" = 31 ,
"END" = 32 ,
"BACK" = 33 ,
"NEXT" = 34 ,
"ORIGIN" = 35 ,
"FILERETURN" = 36 ,
"RETURN" = 37 ,
"FILENAME" = 38 ,
"FIRSTNAME" = 39 ,
"NEXTNAME" = 40 ,
"EXTNAME" = 41 ,
"FIRSTCONTENT" = 42 ,
"NEXTCONTENT" = 43 ,
"FIRSTWORD" = 44 ,
"NEXTWORD" = 45 ,
"DETACHED" = 46 ,
"TTY" = 47 ,
"AUTO" = 48 ,
"CONTINUE" = 49 ,
"ON" = 50 ,
"RECOVER" = 51 ,
"SLINKER" = 52 ,
"UPDATE" = 53 ,
"CLEAR" = 54 ,
"IDENTS" = 55 ,
"FILES" = 56 ,
"DELETE" = 57 ,
"DEFERRED" = 58 ,
"IMMEDIATE" = 59 ,
"NOT" = 60 ,
"PREVENT" = 61 ,
"RESET" = 62 ,
"ARCHIVE" = 63 ,
"SEQUENTIAL" = 64 ,
"TWO" = 65 ,
"JUSTIFIED" = 66 ,
"ASSEMBLER" = 67 ,
"BOTH" = 68 ,
```

```
        "UNDELETE" = 69 ,
        "FOR" = 70 ,
        "STATUS" = 71 ,
        "TAPE" = 72 ,
        "ACCOUNT" = 73 ,
        "NO" = 74 ,
        "VERSIONS" = 75 ,
        "EXTENSION" = 76 ,
        "DATE" = 77 ,
        "CREATION" = 78 ,
        "LAST" = 79 ,
        "FIRST" = 80 ,
        "READ" = 81 ,
        "WRITE" = 82 ,
        "DUMP" = 83 ,
        "EVERYTHING" = 84 ,
        "LENGTH" = 85 ,
        "MISCELLANEOUS" = 86 ,
        "ACCESSES" = 87 ,
        "PROTECT" = 88 ,
        "SIZE" = 89 ,
        "TIME" = 90 ,
        "VERBOSE" = 91 ,
        "SORT" = 92 ,
        "BYTESIZE" = 93 ,
        "ARCHIVED" = 94 ,
        "ALL" = 95 ,
        "MODIFICATIONS" = 96 ,
        "UPPER" = 97 ,
        "LOWER" = 98 ,
        "MODE" = 99 ,
        "SENDMAIL" = 100 ,
        "BUSY" = 101 ,
        "QUICKPRINT" = 102 ,
        "JOURNAL" = 103 ,
        "PRINTER" = 104 ,
        "COM" = 105 ,
        "TERMINAL" = 106 ,
        "REMOTE" = 107 ,
        "REST" = 108 ,
        "CASE" = 109 ,
        "CONTENT" = 110 ,
        "TEMPORARY" = 111 ,
        "VIEWSPECS" = 112 ,
        "EXTERNAL" = 113 ,
        "TO" = 114 ,
        "PRIVATE" = 115 ,
        "PUBLIC" = 116 ,
        "TENEX" = 117 ,
        "ALLOW" = 118 ,
        "EXECUTE" = 119 ,
        "APPEND" = 120 ,
        "LIST" = 121 ,
        "SET" = 122 ,
        "SELF" = 123 ,
        "FORBID" = 124 ,
```

```
"DISK" = 125 ,
"DEFAULT" = 126 ,
"OLD" = 127 ,
"NEW" = 128 ,
"COMPACT" = 129 ,
"RENAME" = 130 ,
"ADD" = 131 ,
"SUBTRACT" = 132 ,
"MULTIPLY" = 133 ,
"DIVIDE" = 134 ,
"RIGHT" = 135 ,
"LEFT" = 136 ,
"ACTION" = 137 ,
"AUTHORS" = 138 ,
"COMMENT" = 139 ,
"EXPEDITE" = 140 ,
"HARDCOPY" = 141 ,
"INFORMATION" = 142 ,
"INSERT" = 143 ,
"KEYWORDS" = 144 ,
"OBSOLETES" = 145 ,
"RFC" = 146 ,
"SUBCOLLECTIONS" = 147 ,
"TITLE" = 148 ,
"UNRECORDED" = 149 ,
"L10" = 150 ,
"PROCEDURE" = 151 ,
"SEQGENERATOR" = 152 ,
"BUFFER" = 153 ,
"NDDT" = 154 ,
"PARSERULE" = 155 ,
"CA" = 156 ,
"CD" = 157 ,
"RPT" = 158 ,
"BC" = 159 ,
"BW" = 160 ,
"BS" = 161 ,
"LITESC" = 162 ,
"IGNORE" = 163 ,
"SC" = 164 ,
"SW" = 165 ,
"TAB" = 166 ,
"IMLAC" = 167 ,
"TI" = 168 ,
"NVT" = 169 ,
"EXECUPORT" = 170 ,
"MENU" = 171 ,
"DNLS" = 172 ,
"TNLS" = 173 ,
"COMMAND" = 174 ,
"RULE" = 175 ,
"SUBSYSTEM" = 176 ,
"DISPLAY" = 177 ,
"FROZEN" = 178 ,
"HLPCOM" = 179 ,
"PROGRAM" = 180 ,
```

```
            "TERSE" = 181 ,
            "INDENTING" = 182 ,
            "UNIVERSAL" = 183 ,
            "ENTRY" = 184 ,
            "INCLUDE" = 185 ,
            "BOTTOM" = 186 ,
            "PAGE" = 187 ,
            "OFF" = 188 ,
            "FULL" = 189 ,
            "PARTIAL" = 190 ,
            "ANTICIPATORY" = 191 ,
            "DEMAND" = 192 ,
            "FIXED" = 193 ,
            "CONTROL" = 194 ,
            "CURRENTCONTEXT" = 195 ,
            "FEEDBACK" = 196 ,
            "HERALD" = 197 ,
            "PRINTOPTIONS" = 198 ,
            "PROMPT" = 199 ,
            "RECOGNITION" = 200 ,
            "STARTUP" = 201 ,
            "LEVELADJUST" = 202 ,
            "REVERSE" = 203 ,
            "TEST" = 204 ,
            "TASKER" = 205 ,
            "LINEPROCESSOR" = 206 ,
            "CENTER" = 207 ,
            "CNTLQ" = 208 ;
      % COMMON RULES %
         % ENTITY DEFINITIONS %
            editentity = textent / structure;
         % TEXT ENTITY DEFINITIONS %
            textent = text1 / "TEXT" / "LINK";
            text1 = "CHARACTER" / "WORD" / "VISIBLE" /
            "INVISIBLE" / "NUMBER";
         % STRUCTURE ENTITY DEFINITIONS %
            structure = "STATEMENT" / notstatement;
            notstatement = "GROUP" / "BRANCH" / "PLEX" ;
       SUBSYSTEM name KEYWORD "NAME"
         INITIALIZATION  fname1 =
            ;
         COMMAND  fname2 = "COMMANDWORD"
            ;
         TERMINATION  fname3 =
            ;
         END.
      FINISH
   FILE lname % (L10.SAV,) TO (lname.subsys,) %
      % globals %
      (xname) PROCEDURE % execution procedure %
         %Formal Parameters%
            (result,     %result record%
            parsemode,   %parsing, backup, cleanup%
            param1,      %your first parameter...%
            param2,      %of course you may have...%
            param3);     %0 to 7 of your own parameters%
```

```
            %Locals%
               REF result, param1, param2, param3;
            CASE parsemode OF
               = parsing:
                  BEGIN
                  END;
               = backup, = cleanup:
                  BEGIN
                  END;
               ENDCASE;
            RETURN(&result);
            END.

         FINISH
```

PART FIVE: Advanced Programming Topics

Section 1: Error Handling -- SIGNALs

Introduction

When an NLS system procedure fails to perform properly, it may generate an error signal. Every signal has a value. When a signal is generated, control is passed back to the last signal trap in effect. If no explicit program control statement (e.g. RETURN, GOTO) is given in that signal trap, a new signal will be generated. If the error is not dealt with, the signal will eventually bubble all the way back to the execution routine. The execution routine should always trap a signal. You may trap signals and regain control by setting up the response in advance.

Trapping Signals

To trap error signals of any error value:

```
ON SIGNAL ELSE statement ;
```

E.g.

```
ON SIGNAL ELSE
   BEGIN
   dismes(2,$string);
   RETURN;
   END;
```

It is a good idea to set up a signal response before calling any NLS system procedures.

Once the signal response is set, it remains in effect through the end of the procedure or until it is changed, and will be executed whenever a signal is received by that procedure. Any subsequent ON SIGNAL statements will at that point change the signal response (i.e. only one signal response can be in effect at any point during procedure execution).

Only signals generated by procedures below (e.g. called by) your procedure will be trapped by your procedure's signal trap. It will not trap signals generated in the same procedure.

The signal response may be any (block of) L10 statement(s). It will be executed, then

- if you have an explicit program control statement (RETURN, GOTO, EXIT LOOP), control will be passed accordingly and the signal will stop there, or
- if the signal trap includes no explicit program control statement, another signal of the same value will be generated, and control will pass upward through the stack

of procedures called until it encounters another signal trap.

A RETURN will return control to the procedure which called the one which intercepted the signal (not the one which generated it).

Thus, if you wish to resume control in the current procedure, the signal trap will have to end with a GOTO statement pointing to an appropriately labeled statement. This is one of the few places where a GOTO is really necessary.

If the signal trap applies to a loop, an EXIT LOOP or REPEAT LOOP is a valid signal program control statement.

Trapping Signals in Execution Routines

If a signal bubbles up through the execution routine to the command parser (in a command in an attachable subsystem), the results may be unpredictable. Execution routines should always include signal traps.

A RETURN(FALSE) will shift CML into backup mode. It will back up to before the last set of CML alternatives (not necessarily equivalent to the last user input element), and then shift back into parsing mode. (This may imply backing all the way through the command, i.e. aborting the command.)

The procedure "abortsubsystem" may be useful in this context. It will shift the command parser into backup mode and abort the current command. Then it will execute a Quit (from the current subsystem) and return the user to the previously used subsystem. It should be passed the address of an error string to be displayed. E.g.

```
ON SIGNAL ELSE abortsubsystem($"Error in xprocedure..." ) ;
```

or

```
ON SIGNAL ELSE abortsubsystem(sysmsg) ;
```

(see "Specific Signals")

Cancelling Signal Traps

After program execution sets up a signal response, the following statement will cancel it so that thereafter a signal will just bubble on up:

```
ON SIGNAL ELSE NULL ;
```

or just

```
ON SIGNAL ELSE ;
```

It may be subsequently reset by execution of another ON SIGNAL statement.

Specific Signals

When a signal is generated, the NLS system global variable "sysgnl" is given a specific value (the value of the signal). Each value represents a certain type of error. Also the system global variable "sysmsg" is given the address of a string which holds an error message.

The above constructions react to any signal, no matter what its value may be. The ON SIGNAL statement can be used much like a CASE statement (comparing cases to the global sysgnl) if you wish to trap specific signals:

```
ON SIGNAL
   =constant: statement;
   =constant: statement;
          •••
   ELSE statement;
```

E.g.

```
ON SIGNAL
   =ofilerr:  %open file error%
      BEGIN
      IF sysmsg THEN dismes(2,sysmsg);
      RETURN;
      END;
   ELSE  %any other error signal%
      BEGIN
      dismes(2,$"Error");
      RETURN;
      END;
```

The current signal constants can be found in <NLS,BCONST,>. The common reason for using this specific signal treatment is when you call a procedure which you know will generate a certain signal value under certain conditions. In such a case, you can learn the signal constant of concern from the SIGNAL statement which generates it.

Generating Signals

You may generate a SIGNAL in a procedure by the statement:

```
SIGNAL (value, astring) ;
```

where value is the value of the signal (perhaps a system global) and astring is the address of a string holding the error message. If the second parameter is omitted, it will be assumed to be zero and no message will be printed. The first parameter is mandatory; every signal must have a value.

Examples:

```
SIGNAL (ofilerr, $"Couldn't open your file.") ;
SIGNAL (2) ;
```

Another way to generate a SIGNAL is by calling the procedure err(errno)

It will generate a SIGNAL of the value "errsig" (a system global) and will set up a message depending on the value you pass for errno. errno may be any of the following:

```
1 -- "File copy fails";
2 -- "Open scratch fails";
3 -- "Cannot load program";
4 -- "I/O Error";
5 -- "Exceed capacity";
6 -- "Bad file block";
7 -- "Not implemented"
```

If you pass it the address of a string as the error number, it will signal using that address for sysmsg, and that string will be printed.

By allowing err to generate all the signals, you will find it easy to freeze execution upon an error condition while debugging using NDDT, as described in the next section (by setting a breakpoint at err).

Be careful not to call err and then trap its SIGNAL in that same procedure. You might say:

```
ON SIGNAL
   =errsig: NULL;
   ELSE ...
```

Section 2: NDDT Debugging

Introduction

Debugging is the process of finding the errors in a program. Once the problem is located, you may correct it in the source

code (NLS file) and recompile.
NLS includes a debugging tool called NDDT, for "NLS Dynamic Debugging Technique." NDDT allows you to examine the state of your program during or after running it (i.e. using the command or filter). This section describes the capabilities of NDDT.

Accessing NDDT

To make NDDT available from NLS, you must execute the command in the PROGRAMS subsystem:

Set Nddt (control-h) OK

This adds the program NDDT to your user programs buffer. Thereafter, whenever you type a <CTRL-h>, NLS will immediately be interrupted (be it in a waiting or running state) and you will enter NDDT. NDDT will respond with its command hearald, a right angle-bracket (>), indicating that NDDT is ready to accept a command.

NDDT commands are specified by typing the first character of the command word.

You may continue with NLS (from the point where it was interrupted) with the NDDT command:

Continue OK

You may continue NLS from a specific instruction address with the NDDT command:

Goto ADDRESS OK

NDDT Address Expressions

Everything stored in the machine (instructions and variables) has an address, its location within the computer's memory. An address is an octal (base-eight) number.
The name of a procedure or of a named L10 statement may be used instead of a number. It represents the octal location of the named statement or of the first instruction of the procedure.
Addresses (symbols or numbers) may be combined, to evaluate to some location. An expression concatenated with the following operators will be evaluated from left to right (no hierarchical ordering) to a single value:

<SP> same as +
-
*
/

Thus, a symbol may be followed by a space (or plus-sign) and then an octal number. The number is added to the location represented by the symbol.

Single-Word Variables

Often, programmers wish to examine or modify the contents of a single word at a given location. The NDDT Show command prints the contents of the word at that address.

Show Location ADDRESS OK

where address is an address expression as defined above or one of the following:

^ -- preceding entity
<LF> -- next entity
Next -- next entity
<TAB> -- entity whose address is the content of current location

NDDT maintains some address as your current location, and the

Show command sets this location to the one it examines. If you do not specify an address in a show command, the current location is assumed.
NDDT can print the contents in three ways: as a symbol followed by a number (to be added to the symbol location), as a single number, or as text. The default printout mode is symbolic. The printout mode may optionally be changed in a Show command. The new printout mode remains in effect until subsequently changed.

Show Location ADDRESS <CTRL-b> PRINTMODE OK
where PRINTMODE is one of the following:

Numeric
Symbolic
Text

A fast way to do the same thing is provided with the Value command:

Value of ADDRESS OK
or
Value of ADDRESS <CTRL-b> PRINTMODE OK

You may print and then replace the value in a word with the Show command:

Show Location ADDRESS _ EXP OK
or
Show Location ADDRESS <CTRL-b> PRINTMODE _ EXP OK
where EXP is an expression whose value will replace the old value of the given location. In addition to the address expressions discussed above, you may use the form:

value1,,value2
where "value1" is a standard expression which will be put in the left half of the word, and "value2" is an expression which will be put in the right half.

String Variables

The contents of a string variable may be examined and modified as well as simple variables, using the command:

Show String ADDRESS OK

Strings are always printed in text printout mode.
You may print and then replace the string with the Show command:

Show String ADDRESS _ STR OK
where STR is a literal string which you type in.

Records

To work with L10 records, you must first set the NDDT record pointer to the first word of an L10 record definition, with the command:

Record pointer set to: SYMBOL OK
where SYMBOL is the name of some L10 record. Note that it may be necessary to use the MARK command (described below) to make local records known to the NDDT system.

This is equivalent to the command:

Show Location RP _ SYMBOL OK

You may then examine all the fields of any record with the command:

Show Record ADDRESS OK
or
Show Record ADDRESS <CTRL-b> PRINTMODE OK

You may examine and optionally change a single field within a

record with the Show Location command, substituting ADDRESS.FIELD for ADDRESS.

You may replace each field in a record with the command:

Show Record ADDRESS _

The name of each field is then printed and a new value may be typed in, terminated by a Command Accept. Typing only a Command Accept will advance to the next field of the record without modifying the last field.

Built in NDDT symbols

A number of symbols are built in to NDDT and may be used in address expressions. When they are used, PRINTMODE will be ignored, since the printout mode is predefined for each of these symbols.

Built in Locations, Registers

A1 -- register A1
A2 -- register A2
A3 -- register A3
A4 -- register A4
R1 -- register R1
R2 -- register R2
R3 -- register R3
R4 -- register R4

Built in Locations, Frame

When a procedure is called, a "frame" is added to the stack. It includes a word (holding the return location of that procedure in the right half) followed by all the parameters, then all the locals. Some predefined symbols allow you access the current or any previous frames and the information in them.

M -- contains address of current frame
MARK -- contains address of previous frame
RET -- return location in current frame
RP -- address of record definition of last field used
S -- contains address of top of stack (last LOCAL word, or whatever)
SIG -- current frame signal location

Built in Records

BASE -- first frame in procedure stack
FRAME -- current frame description
F -- same as FRAME
LOCALS -- current frame LOCALS
L -- same as LOCALS
RECP -- description of current record
R -- same as RECP
PARMS -- current frame parameters
P -- same as PARMS
TOP -- description of top frame in procedure stack

Control Switches

EC -- Current symbol escape character (;)
RNAMES -- If FALSE suppresses printing of record field names
SF -- If FALSE disables these NDDT built in symbols

Special character commands

The special character commands are provided for commonly used functions. All but = are essentially subcommands of the SHOW command and are processed exactly as if they had been preceded

by the command word Show.

= -- Show current location in numeric typout without changing the current printing mode

_ -- Assign a value to current location

^ -- Show previous location

LF -- Show next location

TAB -- Show location addressed by current location

Traces and Breakpoints

If you set a "trace" at a location, the system will print that address every time that instruction is executed. Execution will not be interrupted. You may set a trace with the command:

Trace location ADDRESS OK

If you set a breakpoint at a location, a <CTRL-h> will automatically be executed just before the given instruction (causing you to interrupt execution and enter NDDT). This allows you to interrupt execution of your program at a given point and examine and change the state of the system. A breakpoint may be set with the command:

Breakpoint Set ADDRESS OK

Each trace and breakpoint is assigned a number, beginning with zero, when it is set. You may cancel a trace or breakpoint using this number or using the address to which it is set:

Breakpoint Clear NUMBER OK

or

Breakpoint Clear ADDRESS OK

You may cancel all traces and breakpoints that you have set with the command:

Breakpoint Clear All OK

You may list a trace or breakpoint of a given number and the location to which it is set with the command:

Breakpoint Print NUMBER OK

You may list all traces and breakpoints, their numbers, and their locations with the command:

Breakpoint Print OK

A breakpoint may replace a previous trace or breakpoint (new address, same number) with the command:

Breakpoint Set ADDRESS <CTRL-b> Replaces breakpoint NUMBER OK

A breakpoint may be set so that it only interrupts if a comparison between location and a given constant is true, with the following command:

Breakpoint Set ADDRESS <CTRL-b> Test ADDRESS RELOP CONSTANT OK

where ADDRESS is the location of the word to be compared,
RELOP is one of thfe following: = # < > <= >=
CONSTANT is an expression with a value.

A breakpoint may be set so that it only interrupts if a procedure is called and returns true, with the following command:

Breakpoint Set ADDRESS <CTRL-b> Call PROCEDURENAME OK

L10 Procedures

You may call an L10 procedure from NDDT with the command:

Procedure Call PROCEDURENAME OK

If the procedure requires parameters, you must list them in parentheses, separated by commas, after the name of the

procedure:

Procedure Call PROCEDURENAME (param1, param2, ...) OK

One string, enclosed in quotes, may be included in the parameter list, e.g.:

Procedure Call PROCEDURENAME ("literal", param2, ...) OK

The return value(s) of a procedure call will be typed out. NDDT allows you to replace an existing procedure with a new procedure. Whenever the old procedure is called anywhere in the system, the new procedure will be called instead. The new procedure will be passed the same parameters as were passed to the old. This replacement can be done with the command:

Procedure Replace OLDNAME OK NEWNAME OK

The name of the procedure which was replaced is saved so that it may be restored. The replacement may be cancelled with the command:

Procedure Back up to OLDNAME OK

Symbols

The system maintains a table of symbol names and the addresses which they represent. When a user program is loaded, its symbols are added to the symbol table. Thus, (in addition to system globals) the table is composed of blocks, one for each program.

Each block is refered to by the (unique) name of the program. (This is why the CML and SUBSYS parts of a user attachable subsystem must have different names in the FILE statement.) The list of blocks (programs) is called the "mark stack." Locals as well as globals are recognized by NDDT for only those user programs in the mark stack.

You may list the names of the blocks currently in the mark stack with the command:

Mark symbol table: Print contents of stack OK

A block may be deleted from the mark stack (the symbols remain in the symbol table, but they are not recognized by NDDT) with the command:

Mark symbol table: Clear block PROGRAMNAME OK

A block may be reinstated to the mark stack with the command:

Mark symbol table: Set at PROGRAMNAME OK

A new (empty) block may be added to the mark stack with the command:

Mark symbol table: Set at NEWBLOCKNAME OK

If there is at least one block in the mark stack, a new symbol representing some address may be created with the command:

Define New SYMBOLNAME OK ADDRESS OK

Symbols defined with this command have a global scope, and may be used to satisfy external references in L10 user programs subsequently compiled.

Any symbol within a block listed in the mark stack may be redefined to represent a different address with the command:

Define Old SYMBOLNAME OK ADDRESS OK

If you wish to replace an existing routine by a new version of the same routine, some method of distinguishing between new and old occurrences of the same symbol is required. Any symbol preceded by a semicolon (;) refers to the old occurrence of the symbol. (The semicolon has the effect of disabling the symbol table marking mechanism for the given symbol, causing it to be identified in the "old" section of

the symbol table.)

For example, suppose an existing routine named TEST is to be replaced by a new version of the same routine which you have just compiled (hence is in the mark stack). The NDDT Procedure Replace command can be used as follows:

```
Procedure Replace ;TEST OK TEST OK
```

Scanning for Content

You may search a set of words for a specific content with the command:

```
Find content: CONTENT OK masked by: OK lower address:
STARTADDRESS OK upper address: ENDADDRESS OK OK
```

The content of every word in the specified range will be compared to CONTENT. CONTENT may be of the form of an address or a PDP10 machine instruction. The address and content of each word which matches will be printed. (Note that the "masked by" field was ignored.)

If you wish only to compare certain bits in each word to corresponding bits in CONTENT, you may specify a mask. A mask is a number (of the address form). Only those bit positions in which the mask has a one will be compared. (If the mask is not specified, all ones will be assumed and the entire word will be compared.)

```
Find content: CONTENT OK masked by: MASK OK lower address:
STARTADDRESS OK upper address: ENDADDRESS OK OK
```

MASK may also be of either the ADDRESS form or the PDP10 instruction form.

Section 3: Writing CML Parsefunctions

Parsefunctions

Functions which are declared with the PARSEFUNCTION attribute in CML are assumed to be L10 procedures which are designed to be parsing functions. They are used to examine the user's input. They are called in "parsehelp" mode before being called in "parsing" mode. When so called, they are passed the address of a string as a third implicit argument. The parsefunction routine should fill that string with the appropriate prompt characters which tell what the parsing function is looking for.

When the user is faced with alternatives which include a parsefunction, the parsefunction will be called in parsemode "parseqmark" for the string to include in the questionmark display. This string must be no greater than 24 characters.

Sample Interpreter Parsefunction Routine

Assume that in some command we want the typein of a number to appear as an alternative to some set of keywords. We can accomplish this by defining a parsefunction (call it looknum) which looks at the next input character and succeeds if the next character is a digit and fails otherwise. If we write this function as the first alternative in some command, then control will pass from the interpreter to the parsefunction before it passes to the keyword interpreter.

Suppose our command looks like:

```
COMMAND sample =  "INSERT"
    ( looknum() <"number">  ent _ #"NUMBER"
    /  ent _ ("TEXT"/"LINK") )
    % entity now contains an entity type ( NUMBER, TEXT, or
```

```
        LINK ). We now use the LSEL function to get a selection
        of this type %
           source _ LSEL(ent)
        CONFIRM
        xinsert (ent, source) ;
```

The parsefunction looknum which is called by the interpreter both when prompting the user and also during the actual parse of the command.

```
        (looknum) PROCEDURE  % looks at the next input character,
        if it is a digit, then return TRUE, else return FALSE %
           % FORMAL ARGUMENTS %
              (result,  % address of the result record %
              parsemode,  % parsing mode of the interpreter %
              string);  % address of prompting string %
           REF result, string;
           CASE parsemode OF
              = parsing:
                 CASE lookc() OF  %value of next character in input
                 buffer%
                    IN ['0, '9]:  NULL ;
                    ENDCASE  RETURN(FALSE) ;
              = parsehelp:  %supply string for prompt%
                 *string* _ "NUM:" ;
              = parseqmark:  %supply string for questionmark%
                 *string* _ "Number" ;
              ENDCASE;
           RETURN (&result);
           END.
```

Section 4: Calculator Capabilities

Introduction

L10 arithmetic can only work with integers. The CALCULATOR subsystem holds a numbers of procedures which the user programmer may call to do double-precision floating point arithmetic. Floating point numbers are stored in two-word arrays, which the user programmer must declare. All CALCULATOR routines work with these two word arrays.

Converting String to Double-Precision Floating Point

A number in a string variable may be converted to a floating point array with the procedure:

```
nfloat (astring, aword1, aword2)
```

where astring is the address of a string holding the number,

aword1 is the address of the first word of the array, and

aword2 is the address of the second word of the array.

The number in the string may hold a decimal point, and may be preceded by a minus-sign (-). Other characters (e.g. a dollar sign) may precede the first character of the number (a digit, minus sign, or decimal); they will be ignored.

Converting Floating Point to String

The two word array may be converted back to a string with the procedure:

```
qfloutp (avar, astring, format)
```

where

```
        avar is the address of the (first word of the) array
        holding the floating point number, and
        astring is the address of a string variable in which the
        text of the number is to be placed;
        the third parameter is ignored, so just pass zero.
    The format of the string is dictated by the global variable
    "dfoutm."  The following fields apply to this global [default
    values are in square brackets]:
      fld1 -- characters to the left of the decimal [10]
      fld2 -- characters to the right of the decimal [2]
      fld3 -- characters in exponent field [0]
      round -- number of significant digits to round to [12]
      round must be less than or equal to fld1 + fld2 fld1 + fld2
      must be less than or equal to 12
      oflo -- go to exponent notation if left-of-decimal too big
      [0]
      exsign -- if a positive exponent, use first character of
      exponent field for: [0]
                0 -- first digit of exponent
                1 -- "+"
                2 -- a space
      exp2 -- prefix on exponent: [0]
                0 -- no exponent
                1 -- "E"
                2 -- "D"
                3 -- "*10^"
      dpt -- print decimal point switch (0=Off, 1=On) [1]
      dig -- print at least one digit to left of decimal (0 if
      necessary) (0=Off, 1=On) [1]
      just -- justify number within space of three fields: [1]
                0 -- right justify by adding spaces to left
                     you must also set the
                     global "calflg" to TRUE
                1 -- right justify by adding "0"s
                2 -- right justify by adding "*"s
                3 -- left justify by adding spaces to right
                     you must also set the
                     global "calflg" to FALSE
      sign -- if a positive number, use first character of field
      1 for: [0]
                0 -- first digit of number
                1 -- a space
                2 -- "+"
    Additionally, if the global "cacflg" is TRUE, the number will
    be formatted with commas.
  Calculations with Foating Point
    The following procedures do floating point calculations on the
    two-word arrays described above.  All of the following
    procedures require as parameters the address of the (first
    word of the) arrays.
      qcadd(a,b) -- a _ a + b
      qcsub(a,b) -- a _ a - b
      qcmult(a,b) -- a _ a * b
      qcdiv(a,b) -- a _ a / b
      qcdivw(a,b,c) -- c _ a / b
      qcneg(a) -- a _ -a
```

Section 5: Fields and Records

Introduction

A set of bits within a word can be used without affecting the rest of the word. (On the PDP-10, words are 36-bits long.) A contiguous set of bits within a word is called a field. Fields allow more efficient use of storage.

Once a field is defined, you may apply it to any word (variable). It will refer to the defined set of bits in that word (e.g. the field "RH" refers to the right-most 18 bits of whatever word it modifies).

You may assign a number to or from a field by following the variable name with a period (.), then the name of the field:

```
var.field
```

E.g. stid.stpsid _ origin ;

Many fields are defined in the NLS system, and may be used by user programmers. Some have been mentioned in preceding sections; others may be found in the NLS source code.

Declaring Records

Records are always defined globally. Record definitions are, like global declarations, put outside of procedures within L10 files.

A record definition defines a series of fields, with the length (number of bits) specified for each field:

```
RECORD field1[length], field2[length], ... ;
```

The fields are allocated from right to left within the word.

E.g. the record definition:

```
RECORD right[18], left[17] ;
```

would define two fields. The field "right" refers to the right-most 18 bits of the word. The field "left" refers to the next 17 bits to the left of the field "right." (The left-most bit is not used in this example.)

A RECORD definition may specify any number of fields. If a field is defined to be too large to fit in the remaining bits of the current word, it is automatically defined to represent the first field in the next word. I.e. this and subsequent fields are defined from the right of the next word. This can extend through any number of words.

E.g. the RECORD definition:

```
RECORD field1[18], field2[10], field3[18], field4[36] ;
```

would define the fields as follows:

```
field1  --  right half of word
field2  --  right-most 10 bits in left half of word
field3  --  right half of next word
field4  --  entire third word (i.e. word[2])
```

Of course when using fields that refer to subsequent words, you must be sure that you are operating on arrays of the appropriate size.

Declaring Fields

Although you can declare single fields as described here, the practice is limited. (It is useful in manipulating byte pointers.) User programmers should use RECORD definitions instead.

A single field may be defined with the declaration:

```
DECLARE FIELD name = [address, size : position] ;
```

where

address is the address of the word to which the field

refers,
size is the number of bits in the field, and
position is the number of bits left to the right of the field.

In an assignment, the address of the word referenced is kept in a register, named "rp." It may be used as an index by placing it in parentheses. Thus a FIELD declaration refering to the right half of a word is:

```
DECLARE FIELD right=[(rp), 18:0] ;
```

The left half of the next word could be defined:

```
DECLARE FIELD left=[1(rp), 18:18] ;
```

The address is held in the right half of a byte pointer. You may declare a field with zero as the address, then assign the field definition plus an address to set up a byte pointer:

```
DECLARE FIELD right=[0, 18:0] ;
   then
bytepointer _ right + $variable ;
```

A FIELD declaration may be external as well as global:

```
DECLARE EXTERNAL FIELD name = [address, size : position] ;
```

Section 6: Stacks and Rings

Declaring Stacks and Rings

Stacks and rings are allocated series of words of storage. A stack or ring is defined to hold a given number of records; each record may be a single or a defined number of words. You may "push" records onto the stack or ring and then "pop" them off, as described here.

A stack may be declared (at the global level) with the L10 declaration:

```
DECLARE STACK stackname[size] ;
```

where size is the number of one-word records in the stack.

You may work with records of more than one word with the stack declaration:

```
DECLARE STACK stackname[size,recsize] ;
```

where recsize is the number of words in each record. All records in a stack must be the same size.

Like other declarations, any number of stacks may be declared with the same statement:

```
DECLARE STACK stackname[size], stackname[size,recsize],
...;
```

Stacks may be declared as external to the program:

```
DECLARE EXTERNAL STACK stackname[size,recsize], ...;
```

Ring declarations are identical, with the word "RING" substituted for "STACK." E.g.:

```
DECLARE RING ringname[size], ringname[size,recsize], ... ;
DECLARE EXTERNAL RING ringname[size,recsize], ...;
```

Initializing Stacks and Rings

Before it is used, a stack or ring must be initialized (i.e. cleaned up), with the L10 statement:

```
RESET stackname ;
   or
RESET ringname ;
```

The storage can then be considered empty. The RESET statement can be used whenever you wish to clean up the stack or ring.

Using Stacks and Rings

You may add a record to the top of the stack or ring with the L10 statement:

PUSH address ON stackname ;
where address is the address of the first word (perhaps the single word) of the record to be added to the stack.
-If you try to add more elements than the stack can hold, a SIGNAL will be generated.
-If you try to add more elements than the ring can hold, records will be replaced, starting from the bottom (the first record pushed on).

You may remove a record from the stack or ring, and optionally assign it to a record variable (a simple variable or array of the appropriate size) with the L10 statement:

POP stackname ;
or
POP stackname TO address ;
where address is the address of the first word (perhaps the single word) of the record to receive the record from the stack.
-If you try to remove more elements than the stack currently holds, a SIGNAL will be generated.
-If you try to remove more elements than the ring currently holds, records will be reread, starting from the top. This should be avoided. If you did not previously fill the ring, this top record will hold garbage.

You may read the first word of the record at the top of the stack or ring (without affecting the stack or ring) as an expression by enclosing the name in square-brackets:

[stackname]
The second word (the one below that one the stack) may be read as [stackname - 1], and so on.

E.g.

```
var _ [stackname] ;
```

To use stacks and rings, one usually must keep track of how many records are currently on the storage. Thus, you probably will need to maintain a count in a simple variable in parallel to use of the stack or ring.

Section 7: Using the Sequence Generator

Introduction

The Sequence Generator is used by a number of NLS commands which require a series of statements from an NLS file. A procedure may open a sequence holding a number of statements; the Sequence Generator then passes those statements back, one at a time, every time it is called.

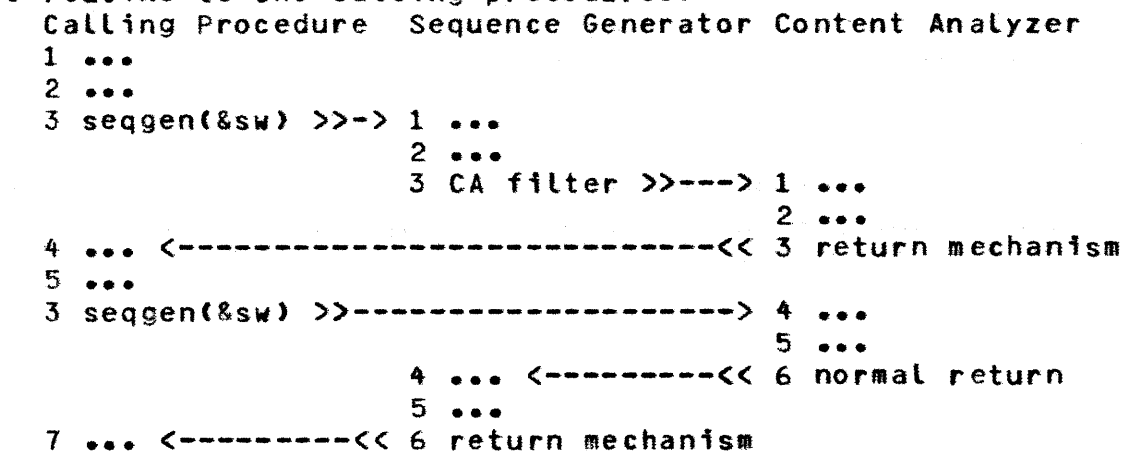
The Sequence Generator considers viewspecs in choosing which statements to return, e.g. level truncation. If viewspec i or k is on, it may call a Content Analyzer program before returning the statement. This allows a great deal of flexibility in working with a series of statements.

Co-Routine Effect

Once the Sequence Generator decides to return a statement (or string), it calls a mechanism which returns control to the procedure that called the Sequence Generator. Thus control will return directly to that calling procedure, even from other procedures the Sequence Generator has called, i.e. even if the return mechanism was called from a procedure called by the Sequence Generator.

When the Sequence Generator is called the next time, it passes

control to the instruction after the one which called the return mechanism. I.e. it continues right where it left off. Thus, the Sequence Generator may call a Content Analyzer program which may return control directly to the procedure which called the Sequence Generator. The next time the Sequence Generator is called, execution will begin in the middle of that Content Analyzer program (which may later return through the normal RETURN statement to the Sequence Generator). (Thus, the Sequence Generator is behaving like a co-routine to the calling procedure.)



Sequence Work Area

When a Content Analyzer program is called by the Sequence Generator, one parameter is passed, the address of an array called the "sequence work area." This array, although ignored by most Content Analyzer programs, holds a great deal of useful information. If the Content Analyzer procedure receives this address as a parameter, and then REFs it, it may refer to the following fields in the sequence work area (see <NLS,BRECORDS,seqr> for entire record declaration):

swstid -- stid of current statement or string in sequence
swcstid -- stid of current real STATEMENT in sequence (even if swstid points to a string)
swlbstid -- stid of statement heading last branch in sequence
swclvl -- level of current statement in sequence
swslvl -- level of first statement in sequence
swvspec -- first word of viewspecs for sequence
swvsp2 -- second word of viewspecs for sequence
swusqcod -- address of user Sequence Generator procedure for sequence
swcacode -- address of Content Analyzer procedure for sequence
swkflg -- FALSE when sequence is opened, TRUE once something has been returned by sequence

Displaying Strings

You may call the return mechanism from Content Analyzer programs while causing the Sequence Generator to inject a string in the sequence. Under the normal circumstance, where the sequence is being used to put up a display or print a file or to do filtered editing, this allows you to inject a string into the output. Thus you may receive a statement, reformat it into a string (without editing the statement itself), and

then display the string.
The following procedure injects a string in the sequence, then returns to the procedure that called the Sequence Generator:

send (sw, astring) ;

where sw is the address of the sequence work area, and astring is the address of the string. (Remember, if you REFed the parameter holding the address of the sequence work area, use the ampersand (&) construct when passing it to send.)

Note that the co-routine effect will cause execution to pick up right where it left off when the Sequence Generator is called for the next statement. Thus, execution will begin just after the send. If you then RETURN a value of TRUE, the statement itself will ALSO be displayed. Most applications of send will RETURN(FALSE) immediately after the call on send.
An example of a Content Analyzer program using send() to show only the first line of each statement:

```
(firstline) PROCEDURE (sw) ; %content analyzer filter to
display only first lines%
   LOCAL TEXT POINTER ptr ;
   REF sw ;
      %to hold address of sequence work area%
   %set pointer at end of first line%
      CASE READC OF
         = ENDCHR: FIND ^ptr ;
         = EOL:    FIND ^ptr _ptr ;
         ENDCASE   REPEAT CASE;
   %put first line in global string%
      *dspstr* _ SF(ptr) ptr ;
   %inject string into sequence%
      send (&sw, $dspstr) ;
   %so statement won't also be displayed%
      RETURN (FALSE) ;
   END.
```

Using Sequences

You may open and use your own sequences in attachable subsystems. This may be useful when you wish to process a series of statements, perhaps only those passing certain requirements (e.g. level or a Content Analyzer filter).
To open a sequence, you should have declared and REFed a variable to hold the address of the sequence work area that will be reserved for your sequence. The procedure which opens the sequence returns this address.

```
&sw _ openseq(stid1, stid2, vspec1, vspec2, seqproc,
caproc);
```

where

stid1 and stid2 are two stids deliniating a group in an NLS file that will be the source of the statements in the sequence. They may be the same (for a branch). The Sequence Generator ignores the branch only and plex only viewspecs.

To get stid2, the procedure "seqend" may be useful. Given stid1 and the two viewspec words, it checks the branch-only and plex-only viewspecs and returns the appropriate stid for stid2. E.g.:

```
&sw _ openseq (stid1, seqend(stid1,vspec1,vspec2),
```

```
            vspec1, vspec2, seqproc, caproc);
```
vspec1 and vspec2 are two words holding the viewspecs for the sequence. There a a number of predefined fields which allow you to set bits within these words. (See Part Four, Section 4.) Of particular interest to the Sequence Generator are the level truncation (not the line truncation) and the Content Analyzer viewspecs.
seqproc is the address of the Sequence Generator routine to be used. If you pass zero, the NLS standard Sequence Generator will be used. (User Sequence Generators are not described here.)
caproc is the address of a Content Analyzer procedure to be used if needed by the sequence (as specified in the viewspecs). If none is needed, you may pass zero.
Passing the address of a sequence is in effect instituting that procedure for that sequence. The address of the currently instituted procedure may be gotten from the display area descriptor, as described in Part Four, Section 4.

A call on the procedure "seqgen" will increment the fields in the sequence work area to the next statement (or string) in the sequence; it will return the first statement in the sequence the first time it is called. You must pass it the address of a sequence work area, e.g.:

```
seqgen (&sw) ;
```

seqgen returns the new swstid field of the sequence, or endfil if there are no more statements in the sequence. You may then refer to the fields in the sequence work area for information about that statement, e.g.:

```
sw.swstid -- stid of current item in sequence
sw.swclvl -- level of current item in sequence
```

When you are done with a sequence, you must close it by calling the procedure "closeseq" with the adddress of the sequence work area; e.g.:

```
closeseq(&sw) ;
```

A typical use of the Sequence Generator might be as follows:

```
% set up sequence %
   % set up viewspecs %
      %get adress of display area descriptor; da is REFed
      simple variable%
         &da _ lda() ;
      %get current viewspecs; vspec is LOCAL two-word
      array%
         vspec _ da.davspec ;
         vspec[1] _ da.davspc2 ;
      %turn on Content Analyzer for this sequence%
         vspec.vscapf _ TRUE ;
   %openseq with "proc" as Content Analyzer filter, returns
   the address of sequence work area; sw is REFed simple
   variable%
      &sw _ openseq(sourcestid, sourcestid, vspec,
      vspec[1], da.dausqcod, $proc);
   ON SIGNAL ELSE closeseq(&sw) ;
% loop through sequence %
   %reset control-o flag%
      inptrf _ 0 ;
```

```
            LOOP
               BEGIN
               IF inptrf THEN %user typed a control-o%
                  BEGIN
                  dismes (1, $"User terminated process") ;
                  EXIT LOOP ;
                  END;
               %increment to next statement in branch you are
               processing which passed filter "proc"; or else exit%
                  IF seqgen(&sw) = endfil THEN EXIT LOOP ;
               %call some procedure to process current stid (could
               as well have been any block of code)%
                  process(sw.swstid) ;
               END;
         % close sequence %
            ON SIGNAL ELSE ;
            closeseq (&sw) ;
```

Section 8: Conditional Compiling

You may delimit blocks of code within procedures that will only be compiled if a constant is TRUE or FALSE. If the code is not compiled, of course it will not be part of the code file and will not be executed.

First a constant must be defined with the SET construct (at the beginning of the file) as either zero (FALSE) or non-zero (TRUE).

Then, code delimited by the string:

```
%+name%
```

where name is the SET constant

will only be compiled if the constant is SET to a TRUE value.

Similarly, code delimited by the string:

```
%-name%
```

will only be compiled if the constant is set to zero (FALSE).

For example,

if the following statement appears at the beginning of the program:

```
SET test=0;
```

then a procedure in the program might include code delimited by this construct, e.g.:

```
L10 statement ; %normal code, always compiled%
   .
   .
L10 statement ; %normal code, always compiled%
%-test%
   L10 statement ; %this statement WILL be compiled%
      .
      .
   L10 statement ; %this statement WILL be compiled%
   %-test%
%+test%
   L10 statement ; %this statement will NOT be compiled%
      .
      .
   L10 statement ; %this statement will NOT be compiled%
   %+test%
```

```
        L10 statement ; %normal code, always compiled%
              .
              .
              .
```

ASCII 7-BIT CHARACTER CODES

Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII
^A	001	!	041	A	101	a	141
^B	002	"	042	B	102	b	142
^C	003	#	043	C	103	c	143
^D	004	$	044	D	104	d	144
^E	005	%	045	E	105	e	145
^F	006	&	046	F	106	f	146
Bell	007	'	047	G	107	g	147
BS	010	(	050	H	110	h	150
Tab	011	)	051	I	111	i	151
LF	012	*	052	J	112	j	152
VT	013	+	053	K	113	k	153
FormFeed	014	,	054	L	114	l	154
CR	015	-	055	M	115	m	155
^N	016	.	056	N	116	n	156
^O	017	/	057	O	117	o	157
^P	020	0	060	P	120	p	160
^Q	021	1	061	Q	121	q	161
^R	022	2	062	R	122	r	162
^S	023	3	063	S	123	s	163
^T	024	4	064	T	124	t	164
^U	025	5	065	U	125	u	165
^V	026	6	066	V	126	v	166
^W	027	7	067	W	127	w	167
^X	030	8	070	X	130	x	170
^Y	031	9	071	Y	131	y	171
^Z	032	:	072	Z	132	z	172
ESC	033	;	073	[	133		
		<	074	\	134		
		=	075	]	135		
		>	076	^	136		
		?	077	_	137	DEL	177
SP	040	@	100				

TITLE PAGE

NLS Programmers' Guide
Content Analyzer
L10 Language
Command Meta Language
NDDT

Augmentation Research Center Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025