KIRK 10-SEP-73 12:43 18967

PSO Group Allocation and my new hours.

Since comiing back from vacation, I have noticed PSO allocation has been lowered from two to one in the mornings. The reason I notice this, is because I have started working 8-5 (for several reasons) and have had difficulty getting online. If the PSO allocation has been cut back in the mornings because it was assumed that I would be working nights, I wish to point out that that is no longer the case. Since you are the assistant director assigned to me and also a decision maker concerning group allocations, I thought I should point out the change in my schedule. 1

18967 Distribution
James C. Norton,

KIRK 10-SEP-73 12:43 18967

PSO Group Allocation and my new hours.

(J18967) 10-SEP-73 12:43; Title: Author(s): Kirk E. Kelley/KIRK;
Distribution: /JCN; Sub-Collections: SRI-ARC; Clerk: KIRK;

Jeff, when youget this, let me know. Mike

18968 Distribution
Jeffrey A. Krend,

(J18968) 10-SEP-73 12:47; Title: Author(s): M. R. Leavitt/MRL;
Distribution: /JAK; Sub-Collections: NIC; Clerk: MRL;

L10 USERS' GUIDE

SRI-ARC

12 SEP 73

Augmentation Research Center

STANFORD RESEARCH INSTITUTE
MENLO PARK, CALIFORNIA 94025

Obsoletes (9246,) and (17698,). Current version available through (userguides,arclocator,2:teb).

Table of Contents

INTRODUCTION.....2

PART ONE: Beginning L10 Programming.....3

Section 1. Content Analyzer Patterns.....3A
Introduction.....3A1
Patterns.....3A2
Examples.....3A3
Using the Content Analyzer.....3A4
Section 2. Content Analyzer Programs.....3B
Introduction.....3B1
Program Structure.....3B2
Procedure Structure.....3B3
Example:.....3B4
Declaration Statements.....3B5
Body of the Procedure.....3B6
Using Content Analyzer Programs.....3B7
Section 3. Content Analyzer Programs: Modifying.....3C
Introduction.....3C1
String Construction.....3C2
Example.....3C3
Controlling Which Statements are Modified.....3C4
Section 4. Executable Programs.....3D

PART TWO: Intermediate L10 Programming.....4

Section 1. The User Program Environment.....4A
Introduction.....4A1
Formatter.....4A2
Sequence Generator.....4A3
Content Analyzers.....4A4
User-Written Sequence Generators.....4A5
Section 2. Program Structure.....4B
An NLS user program consists of the following.....4B1
An example of an L10 program is provided here.....4B2
Section 3. Declarations.....4C
Introduction.....4C1
Variables.....4C2
Simple Variables.....4C3
Arrays.....4C4
Text Pointers.....4C5
Strings.....4C6
Referenced Variables.....4C7
Declaring Many Variables in One Statement.....4C8
Declaring Locals.....4C9
Section 4. Statements.....4D

Introduction.....4D1
Assignment.....4D2
IF Statement.....4D3
CASE Statement.....4D4
LOOP Statement 4D5
WHILE...DO Statement.....4D6
UNTIL...DO Statement.....4D7
DO...UNTIL/DO...WHILE Statement.....4D8
FOR...DO Statement.....4D9
BEGIN...END Statement.....4D10
EXIT Statement.....4D11
REPEAT Statement.....4D12
DIVIDE Statement.....4D13
PROCEDURE CALL Statement.....4D14
RETURN Statement.....4D15
GOTO Statement.....4D16
NULL Statement.....4D17
Section 5. Expressions.....4E
Introduction.....4E1
Primitives.....4E2
Operators.....4E3
Expressions.....4E4
Section 6. String Test and Manipulation.....4F
Introduction.....4F1
Current Character Position (CCPOS).....4F2
FIND Statement.....4F3
Content Analysis Patterns.....4F4
String Construction.....4F5
Example.....4F6
Text Pointer Comparisons.....4F7
Section 7. Executable Programs.....4G
Introduction.....4G1
Moving Around a File.....4G2
Input/Output.....4G3
Section 8. Error Handling -- SIGNALs.....4H
Introduction.....4H1
Trapping Signals.....4H2
Cancelling Signal Traps.....4H3
Specific Signals.....4H4
Section 9. Invocation of User Filters and Programs.....4I
Introduction.....4I1
Programs Subsystem 4I2
Examples of Analyzer-Formatter Programs 4I3

ASCII 7-BIT CHARACTER CODES.....5

INTRODUCTION 2

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. 2a

Occasionally one may need more sophisticated view controls than those available with the viewspec and viewchange features in NLS. 2b

For example, one may want to see only those statements that contain a particular word or phrase. 2b1

Or one might want to see one line of text that compacts the information found in several longer statements. 2b2

One might also wish to perform a series of routine editing operations without specifying each of the NLS commands over and over again. 2c

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. 2d

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. 2e

This document describes three general types of programs: simple filters that control what is portrayed on the user's teletype or display, programs that may modify the statements as they decide whether to print them, and those that, like commands, are explicitly given control of the job. 2f

User programs that control what material is portrayed take effect when NLS presents a sequence of statements in response to a command like Print. 2f1

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. 2f1a

User programs that modify files may gain control at the same point in processing as those that control the view. In their consideration of each statement, they may modify the contents of the statement. 2f2

For more complicated tasks, control may be passed explicitly to the program. In this case, a user program takes on aspects of a special-purpose command. 2f3

This document describes the L10 programming language used at ARC on the PDP10. 2g

Part One is intended for the beginning programmer. Section 1 is a primer for the Content Analyzer. The rest presents a hasty overview of L10 programming, with enough tools to write simple programs. Part Two is intended for the intermediate programmer. Many of the concepts in Part One are repeated in Part Two so that it may stand alone as an intermediate programmer's reference guide. 2g1

More complete documentation can be found in (7052,1). For examples of user programs which serve a variety of needs, consult the User Programs Library Table of Contents (user-progs,-contents,1). 2g2

PART ONE: Beginning L10 Programming 3

Section 1: Content Analyzer Patterns 3a

Introduction 3a1

Content analysis patterns cannot affect the format of a statement, 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. 3a1a

The Content Analyzer filter is created by typing in (or selecting from the text in a file) a string of a special form. This string is called the "Content Analyzer Pattern". 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. 3a1b

Some quick examples of Content Analyzer Patterns: 3a1c

'($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" in them.

SINCE (3-JUN-73 00:00) will show all statements edited since June 3, 1973

Content Analyzer Patterns describe certain things the system must check before printing a statement; 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 pattern; 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. 3a1d

Patterns 3a2

Elements of Content Analyzer Patterns 3a2a

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 Strings

'c	the given character (e.g. a lower case c)
"string"	the given string (may include non-printing characters, such as spaces)

Character classes

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
NP	any non-printing character (e.g. space)

Special characters

SP	a space
TAB	tab character
CR	a carriage return
LF	line feed character
EOL	a carriage return (followed by line feed)
ALT	alt mode character

Special elements

ENDCHR	beginning and end of every statement; can't scan past it
TRUE	is true without checking anything in statement
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 JUN 1973 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-70	17 APRIL 70
APR-17-70	17/5/1970
APR 17 70	5/17/70
APRIL 17, 1970	

Examples of valid times:

1:12:13	1234:56
1234	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)

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.

Combining Elements 3a2b

These elements 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. Several operators can modify the elements:

NUMBER -- multiple occurrences

A number preceding an 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.

$'- 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.

$3LD means from zero to three letters or digits
5$7PT means from 5 to 7 (inclusive) printing characters

[ ] -- floating scan

To do other than a character by character check, enclose an element or series of elements in square brackets [ ]. The Content Analyzer will scan a statement until the element 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 statement begins with the string "start" (or, if it is in the middle of a pattern, check the next 5 characters to see if they 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 other than a letter or digit, such as punctuation

More sophisticated patterns can by written by using the logic features of L10. Generally, an expression is executed left to right. The following operations are done in the given order:

( )
/
NOT
AND
OR

3a2c

( )

Parentheses (and square brackets for floating scans) may be used to group elements.

/

/ means "either or"; the element will be true if either element is true.

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

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. This test always passes.

i.e. It is user 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.)

NOT

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.

Since the slash is executed first, NOT D / 'h will be true if the next character is neither a digit nor the letter "h".

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.

Examples 3a3

D 2$LD / ["CA"] / ["Content Analyzer"] 3a3a

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

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

BEFORE (25-JAN-72 12:00) 3a3b

This pattern will match those statements created or modified before noon on 25 January 1972.

(ID = HGL) OR (ID = NDM) 3a3c

This pattern will match all statements created or modified by users with the identifiers "HGL" or "NDM".

[(2L (SP/TRUE) /2D) D '- 4D] 3a3d

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 YU 4-1234, YU4-1234, and 984-1234.

[ENDCHR] < "cba" 3a3e

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.

Using the Content Analyzer 3a4

Content Analyzer Patterns may be entered in two ways: 3a4a

CA means "Command Accept", a control-D or, in TNLS (by default), a carriage return

1) First you must enter the Programs subsystem with the command:

Goto Programs CA

2) Patterns may be typed in from the keyboard,

Compile Content (analyzer pattern) PATTERN CONFIRM

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

3) or they may be addressed from a file.

Compile Content (analyzer pattern) ADDRESS CONFIRM

In this case, it will begin reading the pattern from the first character addressed and continue until it finds a semicolon (;) so be sure to put a semicolon at the end of the pattern in the file.

Entering a Content Analyzer Pattern automatically does two things: 3a4b

1) It reads the characters in the pattern and compiles executable instructions from them making a small user program, and

2) It takes those instructions and "institutes" them as the current Content Analyzer search 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

Institute Program PROGRAM-NAME CA (as) Content (analyzer) CONFIRM

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 all you have instituted may be listed with the command

Show Status (of programs buffer) CONFIRM

Programs may build up in your program buffer until you have no room for additional patterns. To clear the program buffer, use the Programs subsystem command:

Delete All (programs in buffer) CONFIRM

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

To invoke the Content Analyzer: 3a4c

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

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

In DNLS, if no statements from the CM on pass the Content Analyzer, 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.

When viewspec k is on, the instituted Content Analyzer search program will check until it finds one statement that passes the requirements of the pattern. Then, the rest of the output (branch, plex, 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. Note that i, j, and k are mutually exclusive.

Most of the commands ignore the Content Analyzer in their editing. The following Editor subsystem commands offer the option of specifying viewspecs (which may turn on the Content Analyzer) which apply only for the purpose of that one command and affect what the command works on: 3a4d

Copy

Delete

Move

Substitute

Section 2: Content Analyzer Programs 3b

Introduction 3b1

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. 3b1a

Program Structure 3b2

If you specify a Content Analyzer Pattern, the actual program that is compiled looks like this (with the word "pattern" standing for whatever you typed in): 3b2a

```
PROGRAM name

(name) PROCEDURE;

    IF FIND pattern THEN RETURN(TRUE) ELSE RETURN(FALSE);

    END.

FINISH
```

All L10 programs must begin with a header statement. If the program is to be compiled into your program buffer space, the header statement is 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. 3b2b

```
e.g.  PROGRAM first
      FILE deldir
```

(The Content Analyzer 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.)

The body of a program consists of a series of Declaration statements and Procedures (in any order). In the above case, the program consisted of only one small procedure. When the program is loaded into your programs buffer space, the declarations reserve space in the system for variables. When the program is run, the first procedure is called. It may call other procedures and access global variables in the program or in the NLS system. 3b2c

```
e.g.  DECLARE x, y, z;
      DECLARE TEXT POINER stid;   (described below)
      (first) PROCEDURE; ...
```

The end of the program is delimited by the word FINISH. 3b2d

Comments may be enclosed in percent signs (%) anywhere in the program, even in the middle of L10 statements. The L10 compiler will ignore them. 3b2e

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. 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. 3b2f

Procedure Structure 3b3

Each procedure must begin with a header statement. The header statement is a name enclosed in parentheses followed by the word PROCEDURE, and terminated by a semicolon. 3b3a

```
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. 3b3b

The procedure must end with the terminal statement: 3b3c

```
END.
```

Example: 3b4

PROGRAM compare 3b4a

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

Declaration Statements 3b5

Content Analyzer programs can deal with text pointers and with string variables, while patterns cannot. 3b5a

Text Pointers 3b5b

A text pointer points to 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 3b5c

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 be surrounding the name with asterisks.

e.g. *lstring* is the string stored in the variable named "lstring".

Body of the Procedure 3b6

RETURN Statement 3b6a

No matter what it does, every procedure must return control to the procedure that called it (minor exceptions to be noted later). 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. Some of the many possible statements will be described here; others will be introduced in Part Two of this document. 3b6b

FIND Statement 3b6c

One of the most useful statements for Content Analyzer programs is the FIND statement. The FIND statement specifies a string pattern to be tested against the statement, 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 and the values of all text pointers set within the statement will be reset.

FIND pattern ;

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

↑ pos

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

←NUM pos

back up the specified text pointer by the specified number (NUM) of characters. If NUM is not specified,

one will be assumed. Backup is in the opposite direction of the current scan direction.

pos

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

SF(pos)

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

SE(pos)

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

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.

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

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 ... ;

IF Statement 3b6d

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

Using Content Analyzer Programs 3b7

Once the Content Analyzer program has been written (in an NLS file), there are three steps in using it. First, the program must be "compiled," i.e. translated into machine-readable code. Then, the compiled code must be "loaded" into a space reserved for user programs (the user programs buffer). Finally, the loaded program must be "instituted" as the current Content Analyzer program. 3b7a

There are two ways to compile and load a program: 3b7b

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

First, enter the Programs subsystem with the command:

```
Goto Programs CA
```

Then you may compile the program with the command:

```
Compile L10 (user program at) ADDRESS CONFIRM
```

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

```
Compile File (at) ADDRESS (using) L10 (to file) FILENAME
CONFIRM
```

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 REL (file) FILENAME CONFIRM
```

Once a compiled program has been loaded, it must be instituted. This is done with the Programs subsystem command:
3b7c

Institute Program PROGRAM-NAME
(as) Content (analyzer program) CONFIRM

The named program will be instituted as the current Content Analyzer program, and any previous program will be deinstituted (but will remain in the buffer).

To invoke the Content Analyzer using whatever program is currently instituted, use the viewspec i, j, or k, as describe in the last section (3a4c).
3b7d

Section 3: Content Analyzer Programs: Modifying Statements 3c

Introduction 3c1

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. 3c1a

A Content Analyzer program has several limitations. It can manipulate only one file and it can look at statements only in the order in which they are presented by the NLS sequence generator. 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. The user may write a program to which he can explicitly pass control to overcome these limitations (covered in Section 7 of Part Two -- 4g). 3c1b

String Construction 3c2

Statements and the contents of string variables may be modified by either of the following two statements: 3c2a

ST pos ← strlist ;

The whole statement will be replaced by the string list.

ST pos pos ← strlist ;

The statement from the first position to the second position will be replaced by the string list.

pos may be a previously set text pointer or the SF(pos)/SE(pos) construction.

String variables may also be modified with the string assignment statement: 3c2b

stringname ← strlist ;

The string list (strlist) may be any series of string designators, seperated by commas. The string designators may be any of the following (other possibilities to be described later): 3c2c

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

pos pos

two text pointers previously set in either a statement or a string

stringname

a string name in asterisks, refering to the whole string

E.g.: 3c2d

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

Example: 3c3

PROGRAM delsp 3c3a

```
% Content analyzer.  Deletes all leading spaces from
statements. %
DECLARE TEXT POINTER pt; %reserves space for
                          ("declares") a text pointer
                          named "pt"%
(delsp) PROCEDURE ;
   IF FIND 1$SP †pt THEN %scans over leading spaces,
                          then sets pointer%
      ST pt ← pt SE(pt); %replaces statement with text
                          from pointer to statement end%
   RETURN (FALSE) ;      %return, don't display%
   END.
FINISH
```

Controlling Which Statements are Modified 3c4

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. 3c4a

In DNLS, the system will call the Content Analyzer program whenever the display is recreated (e.g. viewspec f and the Jump commands). If the program returns TRUE, it will only run on enough statements to fill the screen. It is safer to have the program 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 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. 3c4b

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 Unlock and return to a good file. 3c4c

Section 4: Executable Programs 3d

When it is necessary for the program to interact with the user, to work on more than one file, or to skip around in a file, an Executable program must be written. Executable programs may include any of the features of Content Analyzer programs plus other abilities. The discussion of Executable programs will be postponed to Section 7 of Part Two (4g) so as to first establish a firmer foundation of L10 constructs. 3d1

PART TWO: Intermediate L10 Programming 4

Section 1: The User Program Environment 4a

Introduction 4a1

User-written Content Analyzer programs run in the framework of the portrayal generator. They may be invoked in several ways, described below, whenever one asks to view a portion of the file, e.g., with a Print command in TNLS, with any of the output to printer commands, and with the Jump command in DNLS. 4a1a

All of the portrayal generators in NLS have at least two sections -- the formatter and the sequence generator; if the user invokes a Content Analyzer program of his own, the portrayal generator will have one additional part -- the user program. 4a1b

Executable programs are independent of the portrayal generator, although they are welcome to make use of it. They are called as procedures by the Programs subsystem, and have all the powers of any other NLS procedure. 4a1c

Sequence Generator 4a2

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. 4a2a

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

When the sequence generator finds a statement that passes all the viewspec requirements, it returns the statement to the formatter and waits to be called again for the next statement in the sequence. 4a2b

One of the viewspecs that the sequence generator pays

particular attention to is "i" -- the viewspec that indicates whether a user filter is to be applied to the statement. If this viewspec is on, the sequence generator passes control to a user 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.) 4a2c

Formatter 4a3

The formatter section 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. 4a3a

The formatter works by calling the sequence generator, formatting the text returned, then repeating this process until the sequence generator decides that the sequence has been exhausted (e.g. the branch has been printed) or the formatter has filled the desired area (e.g. the display screen). 4a3b

Content Analyzers 4a4

The NLS Portrayal Generator, made up of the formatter, the sequence generator, and user filters, is invoked whenever the user requests a new "view" of the file, for example through the use of the TNLS "Print" command or any of the output to printer commands. Thus if one had a user content filter compiled, instituted, and invoked, one could have a printout made containing only those statements in the file satisfying the pattern. 4a4a

When a user writes an content analyzer filter program, the main routine must RETURN to the Portrayal Generator. The RETURN must have an argument which is checked by the sequence generator. If the value of that argument is TRUE, the statement will be passed to the formatter to be displayed or printed; if the value is FALSE, it will not be displayed. 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). 4a4b

User-Written Sequence Generators 4a5

A user may provide his own sequence generator to be used in lieu of the regular NLS sequence generator. Such a program may call the normal NLS sequence generator, as well as content analysis filters and Executable L10 programs. It may even call other user-written sequence generators. 4a5a

This technique provides the most powerful means for a user to reformat (and even create) files and to affect their portrayal. However, since writing them requires a detailed knowledge of the entire NLS program code, the practice is limited to experienced NLS programmers. 4a5b

Section 2: Program Structure 4b

An NLS user program consists of the following elements, which must be arranged in a definite manner with strict adherence to syntactic punctuation: 4b1

The header - 4b1a

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

The body - 4b1b

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 (4c).

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) ;
```

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.

LOCAL and REF statements will be covered in Section 3 (4c).

L10 statements will be covered in Sections 4 through 5 (4d).

the statement that terminates the procedure (note the final period):

END.

The program terminal statement - 4b1c

FINISH

Comments may be enclosed in percent signs (%) anywhere in the program, even in the middle of L10 statements. They will be ignored. 4b1d

Except for within literal strings, 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. 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. 4b1e

An example of an L10 program is provided here. The reader should easily understand this program after having studied this document. 4b2

4b2a

```
PROGRAM delsp
   % Content analyzer.  Deletes all leading spaces from
   statements. %
   DECLARE TEXT POINTER pt; %reserves space for
                            ("declares") a text
                            pointer named "pt"%
   (delsp) PROCEDURE ;
      IF FIND 1$SP ↑pt THEN %scans over leading spaces,
                            then sets pointer%
         ST pt ← pt SE(pt); %replaces statement holding
                            pt with text from pointer
                            to statement end%
      RETURN (FALSE) ;      %return, don't display%
      END.
   FINISH
```

Section 3: Declarations 4c

Introduction 4c1

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 program). 4c1a

There are various types of declarations available; the most frequently used are discussed here. (Complete documentation is available in the L10 Reference Guide -- 7052,) 4c1b

Variables 4c2

Five types of variables are described in this document: simple, arrays, text pointers, strings, and referenced. Each can be declared on two levels: global or local. 4c2a

Global Variables 4c2b

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

Variables specified in these declarations are outside any procedure and may be used by all procedures in the program. Many globals are defined as part of the NLS system; user programs have complete access to these. Be very careful about changing their values, however.

Local Variables 4c2c

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 prodecure'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 an error results.

Simple Variables 4c3

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. 4c3a

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 of the word respectively; e.g.

```
num.LH ← 2 ;
num.RH ← 3 ;
```

Fields may be defined by the user with RECORD statements (not explained in this document). 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.

Declaring Simple Global Variables 4c3b

```
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 = constant ;

constant 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, causing the latter's address to be used as the value of name

Examples:

```
DECLARE x2 = 5 ;     %x2 contains the value 5%
DECLARE x3 = "OUT"; %x3 contains the word OUT%
DECLARE xx = x1;     %xx contains the address of x1%
```

Arrays 4c4

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": 4c4a

```
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 4c4b

DECLARE name[num] ;

num is the number of elements in the array if the array is not being initialized.

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 = (num, num, ...) ;

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 4c5

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 free standing strings or an NLS statement. 4c5a

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 a string identifier and a character count. 4c5b

The first word, called an stid, contains three fields:

```
stfile -- the file number
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.

The second word contains a character count, with the first position being 1.

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 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 strange errors.

Declaring Text Pointers 4c5c

```
DECLARE TEXT POINTER pt ;
```

The names p1, p2, p3, p4, and p5 are globally declared an reserved for system use.

Strings 4c6

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: 4c6a

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. 4c6b

Declaring Strings 4c6c

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

```
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"

Referenced Variables 4c7

Reference Declarations 4c7a

After a simple variable has been declared, the REF statement can define it to be a pointer to 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 one-word parameters. A variable which contains a pointer to something rather than the thing itself may be passed as an argument to a procedure. If, in the called procedure, one wishes to access the thing itself, the pointer identifier may be declared to be a reference by the REF construction.

Example:

If the simple variable "astr" in the current procedure has been REFed and contains the address of the string "str" local to some other procedure, then:

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

Unreferenced Variables 4c7b

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

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%
```

REFing Simple Variables 4c7c

Once a variable has been declared, it may be REFed with the statement:

REF var ;

Declaring Many Variables in One Statement 4c8

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. 4c8a

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 4c9

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. 4c9a

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. 4c9b

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: 4c9c

```
(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.

The statement which calls procname may look like:

```
procname (locvar, 2) ;
```

var1 will be initialized to the value of the local variable locvar and var2 will get the value 2.

Section 4: Statements 4d

Introduction 4d1

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 (4e). 4d1a

Assignment 4d2

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. 4d2a

```
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. 4d2b

```
(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+b 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: 4d2c

```
var1 ← var2 ← var3 ← exp ;
```

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.

IF Statement 4d3

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. 4d3a

```
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. 4d3b

e.g. 4d3c

```
IF y=z THEN y←y+1 ELSE y←z ;
```

CASE Statement 4d4

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. 4d4a

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

where relop = any relational operator (>=, <, =, IN, etc.) see Section 5 (4e3).

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 relatonal 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 4d4b

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: x ← y;                    %Executed if c = a%
   > b: (x, y) ← (x+y, x-y);      %Executed if c > b%
   ENDCASE y ← x;                 %Executed otherwise%

CASE char OF
   = D: char ← '1;    %if char = the code for a digit%
   > UL: char ← '0;   %if char = the code for an
                       upper-case letter%
   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. 4d4c

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

LOOP Statement 4d5

The statement following the word "LOOP" is repeatedly executed until control leaves by means of some transfer instruction within the loop. 4d5a

```
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 Statement 4d6

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 CASE (described below). 4d6a

```
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. 4d6b

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 hs two globally defined fields:

L -- actual length of contents of string variable
M -- maximum length of string variable

UNTIL...DO Statement 4d7

This statement is similar to the WHILE...DO statement except that 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. 4d7a

UNTIL exp DO statement ;

Example:

```
UNTIL a>b DO a ← a+1 ;
```

DO...UNTIL/DO...WHILE Statement 4d8

These statements are like the preceding statements, except that the logical test is made after the statement has been executed rather than before. 4d8a

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). 4d8b

FOR...DO Statement 4d9

The FOR statement causes the repeated execution of the statement following "DO" until a specific terminal value is reached. 4d9a

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 4e3c)

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

e.g. FOR i UP UNTIL > 7 DO a ← a + t[i] ; 4d9b

Optionally, the user may initialize the variable and may increment it by other than the default of one. 4d9c

FOR var ← exp1 UP exp2 UNTIL relop exp3 DO statement;
DOWN

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 4d10

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. 4d10a

```
BEGIN statement ; statement ; ... END ;
```

Example:

```
IF a >= b*c THEN
   BEGIN
   a←b;
   c←d+5;
   END
ELSE
   BEGIN
   a←c;
   b←d+2;
   c←b*d*7
   END ;
```

EXIT Statement 4d11

This construction provides for forward branches out of CASE or iterative statements. The optional number (num) specifies the number of lexical levels of CASE or iterative statements respectively that are to be exited (if loops are nested within loops). If a number is not given then 1 is assumed. All of the iterative statements (LOOP, WHILE, UNTIL, DO, FOR) can be exited by the EXIT LOOP construct. A CASE statement can be left with an EXIT CASE instruction. EXIT and EXIT LOOP have the same meaning. 4d11a

```
EXIT LOOP num  or  EXIT num
EXIT CASE num
```

where num is an optional integer.

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 4d12

This construction provides for backward branches 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. 4d12a

```
   REPEAT LOOP num
```

REPEAT CASE num (exp) or REPEAT num (exp)

If an expression is given with the REPEAT CASE, then it is evaluated and used in place of the expression given at the head of the specified CASE statement. If the expression is not given, then the one at the head of the CASE statement is reevaluated. 4d12b

Examples: 4d12c

```
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 4d13

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: 4d13a

```
DIV exp , quotient , remainder ;
```

The central connective in the expression must be '/. Quotient and remainder are variable names in which the respective values will be saved after the division. 4d13b

e.g.

```
DIV a / b, a, r ;
```

a will be set to a/b to the greatest integer with r getting the remainder

PROCEDURE CALL Statement 4d14

This statement is used to direct program control to the procedure specified. 4d14a

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

Where procname = the name of a procedure 4d14b

exp = any valid L10 expression (explained in Section 5 -- 4e). The set of expressions separated by commas is the argument list for the procedure. 4d14c

var = any variable. The set of variables is used to store the results of the procedure if there is more than one result. 4d14d

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. 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. 4d14e

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: 4d14f

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. 4d14g

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. 4d14h

e.g. af();

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. 4d14i

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,sysgd,). They should be used with care. 4d14j

RETURN Statement 4d15

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. 4d15a

```
RETURN ;

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

GOTO Statement 4d16

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. 4d16a

```
(label):,statement ;

e.g.  (there): a ← b + c ;
```

GOTO provides for unconditional transfer of control to a new location. 4d16b

GOTO label ;

e.g. GOTO there ;

GOTO statements make debugging difficult and are not considered good style; they can usually be eliminated by use of procedure calls and the iterative statements. 4d16c

NULL Statement 4d17

The NULL statement may be used as a convenience to the programmer. It does nothing. 4d17a

NULL ;

Example:

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

Section 5: Expressions 4e

Introduction 4e1

This section will describe the composition of the expressions which are an integral part of many of the statements described in the last section. 4e1a

Primitives 4e2

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. 4e2a

Each of the following is a valid primitive: 4e2b

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 (4d14f).

an assignment (see below)

classes of characters; described in Sections 1 of Part One (3a2a3)

MIN (exp, exp, ...) the minimum of the expressions

MAX (exp, exp, ...) the maximum of the expressions

TRUE has the value 1

FALSE has the vaue 0

VALUE (astring) given the address of a string containing a number, has the value of the number

READC (see below)

CCPOS (see below)

FIND

used to test text patterns and load text pointers for use in string construction (see Section 6 -- 4f3); 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 /4e2c

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.

-any string of up to five characters enclosed in quotation marks

e.g. "aa" represents the code for 141141B

-the following synonyms for commonly used characters:

ENDCHR -endcharacter as returned by READC

SP -space

EOL -Tenex's version of CR LF

ALT -Tenex's version of altmode or escape (=33B)

CR -carriage return

LF -line feed

TAB -tab

BC -backspace character

BW -backspace word

C. -center dot

CA -Command Accept

CD -Command Delete;

Assignments 4e2d

An assignment can be used as a primitive in an expression.

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

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.

READC - ENDCHR 4e2e

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 (4f2) and (4f3).

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 4e2f

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 to set the current character position for use in text pattern matching. This is discussed in detail in Section 6 below (4f2).

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 *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 4e3

Primitives may be combined with operators to form expressions. Four types of operators will be described here: arithmetic, relational, interval, and logical. 4e3a

Arithmetic Operators 4e3b

Operator	Meaning
unary +	positive value
unary -	negative value
+	addition
-	subtraction
*	multiplication
/	integer division (remainder not saved)
MOD	a MOD b gives the remainder of a / b
.V	a .V b = bit pattern which has 1's wherever either an a or b had a 1 and 0 elsewhere.
.X	a .X b = bit pattern which has 1's wherever either an a had a 1 and b had a 0, or a had a 0 and b had a 1, and 0 elsewhere.
.A	a .A b = bit pattern which has 1's wherever both a and b had 1's, and 0 elsewhere.

Relational Operators 4e3c

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 4e3d

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]

OUT (primitive, primitive) %equivalent to NOT IN%

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

Example:

x IN [1,100)

is the same as

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

Logical Operators 4e3e

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.

Operator	Evaluation
OR	a OR b = TRUE if a = TRUE or b = TRUE = FALSE if a = FALSE and b = FALSE
AND	a AND b = TRUE if a = TRUE and b = TRUE = FALSE if a = FALSE or b = FALSE
NOT	NOT a = TRUE if a = FALSE = FALSE if a = TRUE

Expressions 4e4

Introduction 4e4a

An expression is any constant, variable, special expression form, or combination of these joined by operators and parentheses as necessary to denote the order in which operations are to be performed.

Special L10 expressions are: the FIND expression which is used for string manipulation, 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.

Order of Operator Execution-- Binding Precedence 4e4b

The order of performing individual operations within an equation is determined by the heirarchy 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 4e4c

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 of exp2.

Example:

```
y ← IF x IN[1,3] THEN x ELSE 4;
                %if x = 1, 2, or 3, 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 operator (>=, <, =, IN, etc. See Section 5 -- 4e3c)

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 on 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 incude more than one relop and expression, seperated 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 4e4d

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

Section 6: String Test and Manipulation 4f

Introduction 4f1

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, 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. 4f1a

Current Character Position (CCPOS) 4f2

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. It is moved through the statement or through strings by FIND expressions. It may be set to a particular position by the statement: 4f2a

```
CCPOS pos ;
      or
CCPOS *stringname*[exp] ;
```

pos is a position in a statement or string that may be expressed as any of the following: 4f2b

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. The scan direction over the text will remain unchanged.

e.g. CCPOS pt1 ;

String Front -- left of the first character

SF(stspec)

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

stspec is a string specification that may be expressed as an stid (e.g. the first word of a previouly declared text pointer) or 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 within the string.

If a string (*stringname*) is given after CCPOS, then the position is moved to that string. The scan direction is set left to right. 4f2c

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* ;

FIND Statement 4f3

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 and the values of all text pointers set within the statement will be reset. 4f3a

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. 4f3b

e.g. IF FIND pattern THEN ... ;

FIND Patterns 4f4

A string pattern may be any valid combination of the following logical operators, testing arguments, and other non-testing parameters: 4f4a

Pattern Matching Arguments-- 4f4b

(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 pattern string constant pattern should be reversed. In the above example, one would have "CBA".

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 element following the dash 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 position. 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. 3LD 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 NIC ident of the user who created or last edited the statement is tested by this construction.

SINCE datim

if the string being tested is the text of an NLS statement, this test is TRUE if the statement was created 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.

Format of date and time for pattern matching

Acceptable dates and times follow the forms permitted by the TENEX system's IDTIM JSYS described in detail in the JSYS manual. It

accepts "most any reasonable date and time syntax."

Examples of valid dates:

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

Examples of valid times:

```
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 00:00)
```

These may not appear in Content Analysis patterns, but are valid elements in program FIND statements:

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.

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

Logical Operators-- 4f4c

These combine and delimit groups of patterns. Each compound group is considered to be a single pattern with the value TRUE or FALSE. If text pointers are set within a test pattern and the pattern is not TRUE, the values of

those text pointers are reset to the values they had before the test was made. (See examples below.)

OR
AND
NOT
/

Other Elements-- 4f4d

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:

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.

String Construction 4f5

One may modify an NLS statement or a string with the statement: 4f5a

ST pos ← strlist ;

The whole statement or string will be replaced by the string list.

ST pos pos ← strlist ;

The statement or string from the first position to the second position 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: 4f5b

ST *stringname*[exp TO exp] ← strlist ;
means the same as
stringname[exp TO exp] ← strlist ;

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 "SI" is optional when assigning a strlist to the contents of a string variable. The statement then resembles any simple assignment statement.

The string list (strlist) may be any series of string designators, seperated by commas. The string designators may be any of the following: 4f5c

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 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 "6.0"

Examples: 4f5d

```
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 TO st.L];
```

assuming low and high are declared simple variables.

Example: 4f6

Let a "word" be defined as an arbitrary number of letters and digits. The two statements in this example delete the word pointed to by the text pointer "t", and if there is a space on the right of the word, it is also deleted. Otherwise, if there is space on the left of the word it is deleted. 4f6a

The text pointers x and y are used to delimit the left and right respectively of the string to be deleted. 4f6b

```
IF (FIND t < $LD ↑x > $LD (SP ↑y / ↑y x < (SP ↑x / TRUE)) )
THEN
   ST x y ← NULL;
```
4f6c

The reader should work through this example until it is clear that it really behaves as advertised. 4f6d

Text Pointer Comparisons 4f7

This may be used to compare two text pointers. 4f7a

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

pt1 and pt2 are a 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: Executable Programs 4g

Introduction 4g1

For most applications, it is sufficient to accept statements one at a time from the sequence generator and assume an initial character position of the beginning of the statement (a Content Analyzer program). When one has more complex applications, one may have to write more complex programs which are explicitly passed control. These are not called by the sequence generator but are passed control from the Programs subsystem (see Section 9 -- 4i2). Therefore they must provide themselves with statements on which to work. They should not return a value (as did the simpler Content Analyzer type programs), but should just return control to the calling subsystem. All the capabilities described above are available to such programs. In addition, the program may skip around files, between files, and may interact with the user. 4g1a

Moving Around a File 4g2

Generally, a simple variable or a text pointer will have to be declared to hold the statement identifier (stid) of the current statement. (The first sord 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. 4g2a

In the NLS file system, two basic pointers are kept with each statement: to the substatement and to the successor. 4g2b

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 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.

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". 4g2c

getup(stid) - returns the stid of the up

getnxt(stid) - returns stid of next statement

getbck(stid) - returns the stid of the back

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

Input/Output 4g3

Input and output must be handled quite differently for TNLS and DNLS. There are three system globals which may prove of service in making this distinction: 4g3a

fulldisplay
typewriter
nlmode - the current value, either fulldisplay or typewriter

Example:

```
IF nlmode=fulldisplay THEN something ELSE other-thing;
```

There are a few procedures that work in both DNLS and TNLS: 4g3b

These return the ASCII value of a character from the keyboard input buffer:

input( ) - get next character from keyboard input buffer

inpcuc( ) - get character, forced upper-case, from the keyboard input buffer

lookc( ) - return the next character in the input buffer without advancing the buffer pointer

dismes(type,astring) - given a type number and the address of a string, will print the message on the user's teletype or (in DNLS) display it in the teletype simulation window (above the command feedback line).

type=0: clear message area; astring not necessary
=1: put out message and leave it there
=2: display message for a few seconds (same as 1 for TNLS)
>1000: display for n microseconds (same as 1 for TNLS)

Remember, a dollar sign preceding a variable means the address of that variable.

```
e.g.  dismes (2, $strvar) ;
```

A temporary string may be declared in the procedure call for the use of that procedure alone:

```
dismes (1, $"string of text to be displayed") ;
```

levset(stid,astring) - given an stid and the address of a string containing levadj characters (u's and d's), evaluates levadj and returns a target stid and 0 if new statement is to be down from target or 1 if successor. Used in routines which insert statements.

TNLS 4g3c

There are no standard L10 constructs for TNLS I/O. The following procedures should be of help:

txtlit(astring) - passed the address of a string, appends text from keyboard to string

levadj(stid,astring) - given an stid and the address of a string variable, gets a string of levadj characters (u's and d's) from the user and puts them in the string

tbug(atp) - passed the address of a text pointer, gets address from user

tbug2(atp1,atp2) - get two bugs, the second relative to the first

typeas(astring) - passed the address of a string, types string on tty. The programmer may declare a temporary string in cases like this. e.g.

```
typeas ($"this will print out") ;
```

crlf( ) - type a carriage return-line feed on the tty (You may also have a carriage return in a string passed to typeas.)

DNLS 4g3d

There are some standard L10 statements for DNLS I/O:

INPUT

INPUT may be followed by any sequence of the following; backup within the command (backspaces) is handled automatically:

BUG tp - get a bug selection from the cursor and store the resulting text pointer in tp

STID tp - get a bug from the cursor or a SP followed by a statement name, number or SID, and store the resulting text pointer in tp

LEVADJ str - get a sequence of level adjust characters (u or d) and store them in the string str

TEXT str - get a string of characters (up to a CA or Center-Dot), echoing them in the text area of the display, and store them in the string str

STRING str - like TEXT except echoes in the name area

NAME str - get a string of characters forced upper-case, echoing them in the name area of the display, and store them in the string str; the characters may be typed in or a word may be bugged

WORD str - like NAME except not forced upper-case

NUMBER str - like NAME except inputs a number, typed or bugged

statement; - any standard L10 statement, followed by a semicolon if necessary to delimit the end of the statement; the statement will be executed at that point in the input sequence

char - succeeds if specified character is input; may be any of the characters mentioned under "Primitives" or

```
CA  - Command Accept
CD  - Command Delete
ALT - Alt Mode, Escape
BC  - Backspace Character
BW  - Backspace Word
C.  - Center Dot
```

Example (the Replace Text command):

```
INPUT BUG b1 BUG b2 (BUG b3 BUG b4 CA flag←TRUE; /
TEXT lit CA flag←FALSE) ;

IF flag THEN ST b1 b2 ← b3 b4
       ELSE ST b1 b2 ← *lit* ;
```

DSP -- the Command Feedback line

One may control the text of the command feedback line with the following L10 statement:

DSP [dsp-element) ;

where dsp-element is any sequence of the following:

< - clear command feedback line

← - move arrow to far left

↑ - set arrow under start of nxt word
... - replace last word currently in command feedback line with next word

a word - including letters or digits only; will be added to command feedback line

To display special characters, surround them with quotation marks.

The Command Feedback line may hold up to 30 characters.

Additionally, the following procedures may be of service; some take no parameters:

an( ) - turn arrow on

af( ) - turn arrow off

qm( ) - turn question mark on

qmoff( ) - turn question mark off

dn(astring) - given the address of a string, will display the string in the name register; as with dismes(astring), you may declare a temporary string as the argument

litdpy(astring) - given the address of a string, will clear file display area and display contents of the string

rstlit( ) - restores file area after a litdpy( )

Section 8: Error Handling -- SIGNALs 4h

Introduction 4h1

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) 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 and the program will stop. You may trap signals and regain control by setting up the response in advance. 4h1a

Trapping Signals 4h2

To trap error signals with any error value: 4h2a

```
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 and will be executed whenever a signal is received through the end of the procedure or until it is changed. A signal trap set inside a loop will only remain in effect within the loop. Any subsequent ON SIGNAL statements will at that point change the signal response. 4h2b

Only signals generated by procedures called by the procedure will be trapped by that procedure's signal trap. It will not trap signals generated in the same procedure. 4h2c

The signal response may be any (block of) L10 statement(s). It will be executed, then 4h2d

- if you have an explicit program control statement (RETURN, GOTO, EXIT LOOP), control will be passed accordingly, or

- if the signal trap includes no explicit program control statement, another signal will be generated.

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. 4h2e

If the signal trap applies to a loop, an EXIT LOOP or REPEAT LOOP is a valid signal program control statement. 4h2f

Cancelling Signal Traps 4h3

If, after setting up a signal response, you wish to cancel it so that the signal will just bubble on up, you may do so with the statement: 4h3a

```
ON SIGNAL ELSE ;
```

Specific Signals 4h4

When a signal is generated, an NLS system global variable, sysgnl, is given a specific value (the value of the signal). Each value represents a certain type of error. Also, a system global variable, sysmsg, is given the address of a string which holds an error message. 4h4a

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 if you wish to trap specific signals: 4h4b

```
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,const,). 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. 4h4c

Section 9: Invocation of User Filters and Programs 4i

Introduction 4i1

The user-written filters described in this document may be imposed through the NLS command "Goto Programs". 4i1a

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

These NLS commands are used to compile, institute and execute User Programs and filters. 4i1b

Compilation--

is the process by which a set of instructions in a program is translated from the L10 language written in an NLS file into a form which the computer can use to execute those instructions.

Institution--

is the process by which a compiled Content Analyzer program is linked into the NLS running system for use as a filter.

Execution--

is the process in which control is passed to a compiled Executable program.

This section additionally presents, in detail, examples of the use of the L10 programming language to construct user analyzer filters and reformatters. These programs were written by members of ARC who are not experienced programmers. They do not make use of any constructions not explained in this manual. 4i1c

Programs Subsystem 4i2

Introduction 4i2a

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

goto programs CA

This subsystem enables the user to compile L10 user programs as well as Content Analyzer patterns, control how these are arranged internally for different uses, define how programs are used, and interrogate the status of user programs.

Programs subsystem commands 4i2b

After entering the Programs sbsystem, the system expects one of the following commands:

Show Status of programs buffer

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

Show Status (of programs buffer) CONFIRM

When this command is executed the system will print:

-- the names of all the programs in the stack, including those generated for simple Content Analysis patterns, starting at the bottom of the stack. This stack contains the symbolic names of all compiled programs and a pointer to the corresponding compiled code. The stack is arranged in order of compilation with the most recently compiled program at the top of the stack.

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

-- the current Content Analyser Program or "None"

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

-- the user Sort Key program or "None"

Compile

L10 Program

This subcommand compiles the program specified.

Compile L10 (user program at) ADDRESS CONFIRM

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

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

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

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

File

The user program buffer is cleared whenever the user resets or logs out of the system. If one has a long program which will be used periodically, he may wish to save the compiled code in a file which can be retrieved with the Load REL File command. The command to do this is:

Compile File (at) ADDRESS (using) L10 CA (to file) FILENAME CONFIRM

The FILENAME must be the same as the program name. The program will then be compiled and stored in the 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:

1) replace the word PROGRAM at the head of the file with the word FILE, and

2) position the CM (in DNLS, the top of the screen) at the FILE (ex PROGRAM) statement.

Content Analyzer Pattern

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

Compile Content (analyzer pattern) SELECTON CONFIRM

The pattern must begin with the first visible after the SELECTON address, or at that point you may type it in.

When this command is executed, the pattern specified is compiled into the buffer, its name is put on the stack, and it is instituted as the Content Analyzer filter.

Load REL file

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

Load Rel (file) FILENAME CONFIRM

If the FILENAME is specified without specifying an extension name, this subcommand will search the connected directory, then the <user-progs> 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

Sort key extractor and sequence generator programs are more complex and are generally limited to experienced L10 programmers. Some are available in the User Programs Library (user-progs,-contents,1).

Delete

All

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

Delete All (programs in buffer) CONFIRM

Last

This subcommand deletes the top (or most recent) program on the stack. The program is deinstituted if instituted, its name removed from the stack, and its space in the buffer marked as free.

Delete Last (program in buffer) CONFIRM

Run Program

This command transfers control to the specified program.

Run Program PROGNAME CONFIRM
NUM

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 instituted by its number. This first program loaded into the buffer is number one.

Institute Program

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

Institute Program PROGNAME CA
NUM
(as) CA (content analyzer) CA
Content (analyzer) CA
Sort (key extractor) CA
Sequence (generator) CA

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

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

Deinstitute Program

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

```
Deinstitute Content (analyzer program)   CA
            Sort (key extractor program)
            Sequence (generator program)
```

Set Buffer size

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 initial size is set to 4 pages. This may be increased with the subcommand:

```
Set Buffer (size) NUMBER CONFIRM
```

where NUMBER is the number of pages (512 words each) to be allocated to the user programs buffer.

If you get an "Error in Loading" message when attempting to compile a program or load a REL file, try increasing the buffer size.

You may reset the buffer size (to four pages) with the command:

```
Reset Buffer (size) CONFIRM
```

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 4i3

The following are examples of user programs which selectively edit statements in an NLS file on the basis of text searched for by the pattern matching capabilities. Examples of more sophisticated user programs, including sort keys and user sequence generator programs, can be found in the <user-progs> directory through the file (user-progs,-contents,). One can find out how the standard NLS commands work by tracing them through, beginning with (nls, nctrl, 2). A table of contents to all the global NLS routines available to the user can be found in (nls, sysgd, 1). 4i3a

Example 1 -- Content Analyzer program 4i3b

```
PROGRAM outname % removes the text and delimiters () of NLS
statement names from the beginning of each statement %
   DECLARE TEXT POINTER sf;
   (outname)PROCEDURE;
      IF FIND $NP '( ['')] ↑sf THEN %found and set
                                    pointer after name%
         BEGIN
         ST sf ← sf SE(sf);
         RETURN(TRUE);
         END
      ELSE RETURN(FALSE);
      END.
   FINISH
```

Example 2 -- Content Analyzer program 4i3c

```
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
         ST pt pt ← "[CHANGED]";  %the substring of zero
                                  length is replaced with
                                  "[CHANGED]"%
      RETURN(FALSE);
      END.
   FINISH
```

Example 3 -- Executable program 4i3d

```
FILE toc %This program will generate a table of contents
branch with statement numbers  %
   (toc) PROCEDURE ;
      % declarations %
         LOCAL level, da, vspec, last, place ;
         LOCAL TEXT POINTER ptr ;
         LOCAL STRING num[5] ;
         REF da ;
         num.L ← ptr ← 0;  %initialization%
      % input file and number of levels %
         IF nlmode=typewriter
            THEN
               BEGIN
               crlf( ) ;
               typeas( $"Table of Contents generator:
               Select file ");
               tbug ($ptr) ;  %get a bug from the tty%
               crlf( ) ;
               typeas ($"Number of levels of depth:  ") ;
               txtlit ($num) ;  %get a text string from the
               tty%
               crlf( ) ;
               typeas( $"running... ");
               END
            ELSE %display%
               BEGIN
               dn( $"") ;  %clear the name register%
               DSP (< Table of Contents ↑ Select file) ;
               INPUT STID ptr CA;
               DSP (←< Levels of depth) ;
               INPUT NUMBER num CA ;
               DSP (< Table of Contents being genera) ;
               dn( $"ted"); %command feedback line too
               short %
               END;
      % set to origin %
         ptr.stpsid ← origin ;
         ptr[1] ← 1 ;
         level ← VALUE ($num); %evaluate number string%
         level ← MIN (50, MAX (1,level)); %levels of depth%
      % insert table of contents statement %
         ptr ← cis (ptr, $"Table of Contents", down);
         %command insert statement procedure%
      % get viewspec words %
         &da ← dsparea (lcda( )); %get address of display
         area records, which hold all information about
         display window, e.g. viewspecs%
         vspec ← da.davspec ; %copy viewspec word%
         vspec.vslev ← level ;  %adjust level viewspec%
```

```
            vspec.vsbrof ← vspec.vsplxf ← FALSE;  %adjust
            branch or plex only viewspec%
         % assimilate group to table of contents %
            place ← ptr ;
            last ← getsuc (place) ;
            cea [ptr, getsuc(ptr), getail(ptr), 0, vspec,
            da.davspc2, da.dausqcod, da.dacacode); %command
            execute assimilate procedure, using modified copy
            of first viewspec word and the rest from the
            display area descriptors%
         % for all statements in table of contents %
            UNTIL (place ← getnxt(place)) = last DO
            dotoc(place) ;  %turns statement into line for
            table of contents%
         % move table of contents to under st 1 %
            cmg [ptr, getsuc(ptr), getprd(last), $"d");
            %command move group procedure%
         % recreate display %
            IF nlmode=fulldisplay THEN alldsp( ) ELSE crlf( ) ;
         RETURN ;
         END.
      (dotoc) PROCEDURE (stid) ; %passed stid, replaces

**statement with table of contents line %
         % declarations %
            LOCAL length;
            LOCAL STRING dots[70], stnum[50], st[2000] ;
            LOCAL TEXT POINTER end ;
         % initializations %
            length ← st.L ← stnum.L ← 0;
            *dots* ←
            "..................................................
            ....................." ;
         % get st number %
            stnum.L ← 0;
            fechno (stid, $stnum); %put statement number in
            string%
         % get first line %
            *st* ← SF(stid) SE(stid) ;
            length ← (65 - (3*getlev(stid)+stnum.L)); %maximum
            length%
            IF length < st.L THEN
               BEGIN
               st.L ← length ;  %truncate statement%
               FIND SE(*st*) [NP] ↑end > ; %back up to end of
               last word%
               *st* ← SF(*st*) end ;
               END;
         % format string %
```

```
            dots.L ← (length + 2) - st.L; %calculate number of
            dots%
            *st* ← *st*, *dots*, *stnum*; %constuct table of
            contents string%
         % replace statement %
            ST stid ← *st* ;
         RETURN;
         END.
      FINISH toc
```

Procedures Used in Examples; references taken from <NLS>SYSGD 4i3e

Format of references:

(proc-name) (link to source code) st-num-of-source-code

(formal, parameters, if, any)

comment taken from source code file

(alldsp) (nls,dspgen,alldsp) 3A
recreate display for all display areas
(cea) (nls,corenl,cea) 7A
(target,src1,src2,levstg,vspec1,vspec2,usqcod,cacode)
Core NLS Assimilate Command
(cis) (nls,corenl,cis) 9H
(stid,astrng,levstg)
Core NLS Insert Statement Command
(cmg) (nls,corenl,cmg) 11L
(stid1,stid2,stid3,levstg)
Core NLs Move Group Command
(crlf) (nls,inpfbk,crlf) 6G
type a carriage return-line feed
(dn) (nls,inpfbk,dn) 8E1
(astrng)
display string in name area
(dsparea) (nls,dactrl,dsparea) 5M
(dano)
get da entry address from display number -- returns
FALSE if da entry is not allocated
(fechno) (nls,seqgen,fechno) 4J
(stid,astr)
Puts statement number of stid in string. Give the STID
as the first argument, and the address of the string
which is to contain the statement number as the second.
The statement number will be built in the string. If
the structure is not intact or the statement vector
cannot be built, a call to RERROR or an EXCEED CAPICITY
ERROR may result.

(getail) (nls,strmnp,getail) 10A
(stid)
Given an stid, this procedure returns the stid of the tail of the current plex

(getlev) (nls,seqgen,getlev) 4I
(stid)
Called with STID, returns level of that statement.

(getnxt) (nls,strmnp,getnxt) 10G
(stid)
This procedure finds the sequentially "next" statement, i.e. the substatement, successor, or successor of up, etc, of the stid passed as argument. Ignores all viewspecs.

(getprd) (nls,strmnp,getprd) 10D
(stid)
Given an stid, this routine returns the predecessor; if the psid heads a plex, the stid itself is returned

(getsuc) (nls,filmnp,getsuc) 2H1
(stid)
The stid for the successor field is returned. If there is no successor, the stid of the up is returned.

(lcda) (nls,dactrl,lcda) 5J
returns number of display area where bug resided at last input character

(tbug) (nls,txcmnd,tbug) 5A
(ptr)
given the address of a text pointer, gets an address selection from the TNLS user and puts it in the text pointer.

(txtlit) (nls,inpfbk,txtlit) 5B
(astrng)
passed the address of a string, appends text from keyboard input buffer to string

(typeas) (nls,inpfbk,typeas) 6C
(astrng)
Given the address of a string, types the string on the user's teletype.

ASCII 7-BIT CHARACTER CODES

5

Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII
Tab	011	/	057	B	102	U	125	h	150
LF	012	0	060	C	103	V	126	i	151
FormFeed	014	1	061	D	104	W	127	j	152
CR	015	2	062	E	105	X	130	k	153
SP	040	3	063	F	106	Y	131	l	154
!	041	4	064	G	107	Z	132	m	155
"	042	5	065	H	110	[	133	n	156
#	043	6	066	I	111	\	134	o	157
$	044	7	067	J	112	]	135	p	160
%	045	8	070	K	113	↑	136	q	161
&	046	9	071	L	114	←	137	r	162
'	047	:	072	M	115			s	163
(	050	;	073	N	116	a	141	t	164
)	051	<	074	O	117	b	142	u	165
*	052	=	075	P	120	c	143	v	166
+	053	>	076	Q	121	d	144	w	167
,	054	?	077	R	122	e	145	x	170
-	055	@	100	S	123	f	146	y	171
.	056	A	101	T	124	g	147	z	172

```
(J18969) 11-SEP-73 17:03;  Title:  Author(s): Stanford Research
Institute /&SRI-ARC; Sub-Collections: SRI-ARC; Clerk: NDM;
Origin: <USERGUIDES>L10-GUIDE.NLS;95, 11-SEP-73 12:44 NDM ;   .D=On;
.PN=0; .PES;
 .DefaultFont=8p,5; .BP=J; .PxFont[1]=14p,6; .PxFont[1]=12p,6;
.PxFontShow=1,2;
 .V1Font=8p,0; .H1="L10 USERS' GUIDE .Split; SRI-ARC .GD; 18969";
.H1Font=12p,6,0;
 .F="page .GPN;"; .FP=FR; .FFont=12p,6,0; .PNType=2;
 .LMBase=2,1.0; .LM=-3; .RM=72,8.0; .BRM=68,7.5; .SN=Off;
 .SNF=72,8.0; .SNFShow=Off; .SNFFont=6p,1,Light; .SNFFontShow=<=2;
 .PxPShow=1; .YBS=0,2p; .PxFShow=1,2; .PxFYD=1; .PxFYU=1; .PxFYS=1;
```

JEW 10-SEP-73 14:59 18976

Proposed Design for Initial Privacy Features

Please comment if you're so inclined, distribute to anyone else who might be interested if you know of such a person, and ignore otherwise.

Proposed Design for Initial Privacy Features

INTRODUCTION 1

The following are proposed changes to the SRI-ARC system which would support an initial, limited level of ident-keyed access control over NLS files. These changes are thought simple enough to be implemented within a reasonable time frame, yet consistent with what is believed to be the long-term solution to this fairly complicated problem. 1a

This proposal is based upon the belief that idents, not TENEX directories, must inevitably become the basis for identifying users within the SRI-ARC system. Hence, the one major change proposed here is a change to the monitor, one that requires that a user identify himself by ident, rather than by directory. 1b

This change would not be required if users could be placed in one-to-one correspondence with directories. 1b1

This proposal would permit the creator of an NLS file to share it with -- and only with -- any desired set of users. Initially, to share a file with another user will mean to allow him to read it via NLS, and possibly (provided one additional constraint is met) to edit it. 1c

In the implementation described here, the task of actually checking a user's access to a file is assigned to NLS, rather than to the monitor. This is done only to simplify the initial implementation. The check must eventually be performed by the monitor if a level of security consistent with that of current TENEX access controls is to be provided. 1d

THE PROPOSAL 2

VERIFYING THE USER'S IDENTITY 2a

Use of an ident must be restricted to its owner. Therefore: 2a1

(P1) It's proposed that a password be associated with each individual ident and required of the user at login (i.e., by the LOGIN JSYS. A supportive change must also be made to the EXEC.). An NLS ident and password, rather than a TENEX directory and password, would thus become the basis for gaining access to the system. The monitor whould infer the user's directory from his ident (something which is always possible) instead of the reverse as is done now (which is only sometimes possible), use the directory thus obtained to proceed with the login, and then simply save the login ident in a job-global cell available to NLS. 2a2

There currently exists a sequential file giving ident as a function of directory. It's suggested that to implement the above, the file be inverted and NLS passwords included in it. 2a2a

(P2) It's proposed that a JSYS be provided which returns the login ident. 2a3

RESTRICTING ACCESS TO FILES 2b

Access to a file must be restricted to the set of users specified by the creator of the file. Therefore: 2b1

(P3) It's proposed that the NLS 'load file' primitive check the origin statement of a file being loaded for an optional field of the form: 2b2

Access List: <identi> ... ,<identn>; 2b2a

containing a list of individual and/or group idents; and if such a field exists, that NLS deny the user access to the file (i.e., refuse to load the file) unless the list contains the login ident or that of a group which contains it (or unless the user is an enabled wheel). 2b3

To make this check efficient: 2b3a

(P4) It's proposed that at NLS initialization, the login ident be retrieved from the monitor via the JSYS provided, and stored in an NLS global. 2b3a1

(P5) It's proposed that a list of all those groups of which the user is a member be maintained in the ident file for each individual ident, and that this list be copied to an NLS global at initialization. Verifying access to the file thus requires only a comparison of strings, rather than an appeal to the ident system. 2b3a2

This access check always occurs IN ADDITION TO the normal TENEX access checks, implying the following: 2b3b

(1) A user cannot effectively be granted write access to an NLS file unless he and the file share the same directory or TENEX directory group. 2b3b1

(2) Among users who share a directory or directory group, access to any NLS file within it can be controlled with complete freedom: ANY subset of those users can be

granted exclusive read/write access to ANY of those files. 2b3b2

(3) Once a file is journalized, the access list (if there is one) effectively becomes the set of users with read access to the file, since TENEX access controls deny write access to every user. 2b3b3

(P6) It's proposed that an NLS command be provided to establish or replace the access field in the origin statement, verifying the list of idents entered by the user. 2b4

The fact that the access list resides in the origin statement of the file is an artifact of the initial implementation; the user is not expected to deal with it directly via NLS editing machinery (though he can't be prevented from doing so). 2b4a

(P7) It's proposed that the access list be verified at Journal submission, since the user may have edited it by hand. 2b5

AVOIDING FORGERY 2c

To prevent one's signing someone else's name to a memo that he neither composed nor authorized: 2c1

(P8) It's proposed that the Journal ALWAYS take the login ident to be the clerk. 2c1a

RESTRICTING ACCESS TO DELIVERED MAIL 2d

To insure the integrity of delivered mail (necessary because its text may be included in the delivery), some of which may be of a private nature: 2d1

(P9) It's proposed that initial files be assumed private by the Journal. That is, whenever the Journal has occasion to create an initial file to receive delivered mail, it's proposed that it place the text 'Access List: <owner>;' in its origin statement. 2d1a

Of course, once the initial file is created, the user may change access to it if he desires. The Journal will never again interfere, so long as the user refrains from deleting the file. 2d2

SUMMARY OF SYSTEM CHANGES 3

It's proposed that: 3a

(P1) a password be associated with each individual ident and required of the user at login (i.e., by the LOGIN JSYS. A supportive change must also be made to the EXEC.). The monitor whould infer the user's directory from his ident instead of the reverse as is done now, use the directory thus obtained to proceed with the login, and then simply save the login ident in a job-global cell available to NLS. 3b

(P2) a JSYS be provided which returns the login ident. 3c

(P3) the NLS 'load file' primitive check the origin statement of a file being loaded for an optional field of the form: 3d

Access List: <ident1> ... ,<identn>; 3d1

containing a list of individual and/or group idents; and if such a field exists, that NLS deny the user access to the file (i.e., refuse to load the file) unless the list contains the login ident or that of a group which contains it (or unless the user is an enabled wheel). 3e

(P4) at NLS initialization, the login ident be retrieved from the monitor via the JSYS provided, and stored in an NLS global. 3f

(P5) a list of all those groups of which the user is a member be maintained in the ident file for each individual ident, and that this list be copied to an NLS global at initialization. Verifying access to the file thus requires only a comparison of strings, rather than an appeal to the ident system. 3g

(P6) an NLS command be provided to establish or replace the access field in the origin statement, verifying the list of idents entered by the user. 3h

(P7) the access list be verified at Journal submission, since the user may have edited it by hand. 3i

(P8) the Journal ALWAYS take the login ident to be the clerk. 3j

(P9) whenever the Journal has occasion to create an initial file to receive delivered mail, it place the text 'Access List: <owner>;' in its origin statement. 3k

18976 Distribution

Jeanne M. Leavitt, Rodney A. Bondurant, Jeanne M. Beck, Mark Alexander Beach, Judy D. Cooke, Marcia Lynn Keeney, Carol B. Guilbault, Susan R. Lee, Elizabeth K. Michael, Charles F. Dornbush, Elizabeth J. (Jake) Feinler, Kirk E. Kelley, N. Dean Meyer, James E. (Jim) White, Diane S. Kaye, Paul Rech, Michael D. Kudlick, Ferg R. Ferguson, Douglas C. Engelbart, Beauregard A. Hardeman, Martin E. Hardy, J. D. Hopper, Charles H. Irby, Mil E. Jernigan, Harvey G. Lehtman, Jeanne B. North, James C. Norton, Jeffrey C. Peters, Jake Ratliff, Edwin K. Van De Riet, Dirk H. Van Nouhuys, Kenneth E. (Ken) Victor, Donald C. (Smokey) Wallace, Richard W. Watson, Don I. Andrews

Proposed Design for Initial Privacy Features

(J18976) 10-SEP-73 14:59; Title: Author(s): James E. (Jim) White/JEW; Distribution: /SRI-ARC; Sub-Collections: SRI-ARC; Clerk: JEW; Origin: <WHITE>QUICK-PRIVACY.NLS;11, 10-SEP-73 14:51 JEW ;

CFD 10-SEP-73 16:26 18977

Meeting to resolve command language problems

A meeting to discuss and resolve existing problems in the proposed command language will be held on Wed, Sept. 12 at 10:00 AM. Sorry for the short notice, but we'd like to resolve the known problems ASAP. 1

18977 Distribution

James C. Norton, Richard W. Watson, Charles H. Irby, Michael D. Kudlick, Diane S. Kaye, Harvey G. Lehtman, Dirk H. Van Nouhuys, N. Dean Meyer, Jeanne M. Beck,

CFD 10-SEP-73 16:26 18977

Meeting to resolve command language problems

(J18977) 10-SEP-73 16:26; Title: Author(s): Charles F. Dornbush/CFD ; Distribution: /JCN RWW CHI MDK DSK HGL DVN NDM JMB ; Sub-Collections: SRI-ARC; Clerk: CFD ;

More on NLS Command Language Syntax for HELP Users

This responds to Dean Meyer's note (18826,) in which he correctly pointed out some deficiencies in my earlier note (18818,). In the present note, it is recommended that we not use the SSEL concept, and I suggest that we discuss this at (or after) CFD's meeting on the command language ambiguity he discovered.

More on NLS Command Language Syntax for HELP Users

INTRODUCTION 1

In (18826,) Dean Meyer has correctly pointed out some deficiencies in the NLS command language syntax scheme I described in (18818,). 1a

This note responds to Dean's suggestions (and implicitly to most of those of JMB in (18940,)). It also includes ideas generated from discussions Dean and I had. These discussions were very useful to me in defining more precisely the particular problems concerning definition of a "selection" (the SSEL-DSEL-LSEL concepts). I wish to acknowledge Dean's interest and understanding of these problems, and his patience with me. 1b

The subject of ADDRESSES needs wider discussion than just between Dean and me, so I propose that we have a meeting to discuss the issues raised below soon. Since Dean's full-time summer employment ends this Friday (Sept 14), the meeting should be this week. I suggest we discuss it during the meeting on the use of "[TOWHERE]" that CFD has asked for, scheduled for Wednesday. 1c

ADDRESSES: (18826,2a) 2

Definitions 2a

The definitions of SSEL (source selection), DSEL (destination selection), and LSEL (literal selection) may be written as: 2a1

	TNLS	DNLS	2a2
SSEL =	ADDRESS / (<option>TYPEIN)	ADDRESS / TYPEIN	
DSEL =	ADDRESS	ADDRESS	
LSEL =	TYPEIN / (<option>ADDRESS)	TYPEIN / ADDRESS	2a3

where in TNLS, ADDRESS = DAE <accept>
in DNLS, ADDRESS = BUG / (<option>DAE <accept>) 2a3a

and TYPEIN = LIT <accept>
<accept> = <control-d>
<option> = <control-u> 2a3b

(Note: when designating TEXT or a GROUP the above definitions must of course be modified to allow for two selections, not one.) 2a4

SSEL 2b

The main problem from a documentation standpoint stems from the introduction of the "SSEL" concept, to distinguish it from "DSEL". 2b1

As I understand it, SSEL was introduced to generalize the designation of "source" operand selections. These occur only in the commands APPEND, COPY, MOVE. 2b2

The generalizations define, in a natural way, two alternatives --- an ADDRESS and a LITERAL --- whenever it is possible that an operand selection might be either typed in as a LIT or selected from a file. 2b3

However, the SSEL concept seems to be an unnecessary generalization, because in the three commands in which it is used it is virtually certain that the user would not want to type a LIT for the "source selection". 2b4

In fact, if for "source" one were to type a LIT, then the APPEND, COPY, or MOVE command would perforce be changed to an INSERT. 2b5

So one objection I have is in allowing anyone using the current command language to change commands in midstream, except via the <control-x> mechanism. 2b6

(I don't mean to preclude us from moving in this direction for future versions of the command language. But it seems to me that that is a separate research effort itself, and shouldn't be approached by making isolated changes to the existing language.) 2b6a

The other objection I have is that the SSEL concept is especially confusing from a documentation standpoint. It requires additional explanations and a notation (acronym) for a situation that will practically never arise. 2b7

What one would invariably type for the source is an ADDRESS not a LIT. This is in keeping with the definition of "source" meaning "a string or structure already in a file". 2b7a

If we introduce a new concept that isn't going to be used much if at all, it seems to me the learning proces is bound to be more difficult, and the overall form and simplicity of the language is more obscured, not less. 2b7b

I propose therefore that we eliminate (or at least not document) the "SSEL" concept, and simply use instead the "DSEL" concept, which is just an ADDRESS. 2b8

LSEL 2c

The case for LSEL is different. Dean pointed out to me that

there are many cases where the concept of LSEL would be useful as a "global acronym". I agree, provided we use a more descriptive acronym than "LSEL". I therefore propose the following list of global acronyms to replace those in (18818, 3b1a): 2c1

```
OPERAND (replaces LSEL)
        =  TYPEIN / ADDRESS in DNLS
        =  TYPEIN / (<option>ADDRESS) in TNLS
ADDRESS  (replaces DSEL and SSEL)
        =  BUG / (<option>DAE <accept>) in DNLS
        =  DAE <accept> in TNLS
TYPEIN     =  LIT <accept>
FILENAME   =  OPERAND for the special string "filename"
STRING     (replaces "TEXT-ENTITY")
STRUCTURE (replaces "STRUCTURE-ENTITY")
LEVEL-ADJUST     (replaces "LEVADJ")
```
2c1a

Note: If anyone has an alternative acronym for OPERAND or LEVEL-ADJUST, both Dean and I would be happy to consider it. 2c2

JMB has suggested CONTENT instead of OPERAND. How does that feel? Any other ideas? 2c2a

OTHER ITEMS: 3

(18826, 2b) With the above modifications, it doesn't seem necessary to separate the command summary into two documents. Command language differences between DNLS and TNLS have already been greatly reduced, and the remaining ones (certain viewspecs, window commands, etc, as well as in the above definitions of global acronyms) can be noted appropriately. 3a

(18826, 2c) LEVEL is not as accurate as LEVADJ, but it has much more connotation, which was the intent. We have compromised by choosing LEVEL-ADJUST as indicated in the above table. 3b

(18826, 3) I agree that the fewer acronyms the better. Where noise words are descriptive enough, a global acronym should be used, as Dean suggests. I erred in using OLDSTRING and NEWSTRING in my SUBSTITUTE example (18818, 7b2), because I didn't know what the correct noise words were. Use of OPERAND (or its equivalent) is fine there, given good noise words. But in a command like PROTECT FILE a local acronym seems far more preferable than a long string of noise words. 3c

(18826, 4) I gave a bad solution to the problem of "ANSWER" in (18818, 3b3). The problem, in my opinion, is that the end of a command always ought to be a <confirm>. The way to achieve this

in the cases where an "answer" occurs at the end of a command, is to define ANSWER to be YES or NO or "null". That is what I propose. Then in a syntax expression, ANSWER<confirm> is unambiguous. 3d

(18826, 5) Dean and I agreed to tryout some syntax examples on persons with no prior knowledge of NLS, in order to see what convention for "space" makes most sense. The main choices are 3e

(1) use <sp> for the space that must be typed, and use an actual space for readability 3e1

(2) use an actual space for the space that must be typed, and do not use <sp> at all. 3e2

(18826, 6) I think Dean's suggestion of <control-y> is better than <ctl>y, for the reasons he stated. 3f

(18826, 8) I certainly don't want to de-emphasize the importance of structure, but I think the LEVEL-ADJUST field should be optional for two reasons: 3g

1) it isn't always used, even when it can be, and
2) its use as a NON-optional field would conflict (as at present) with statements beginning with a "d" or "u" that is not followed immediately by a space. 3g1

(18826, 9) I prefer the scheme of YES or NO (as modified above in the acronym ANSWER) rather than an option key to "cycle back". 3h

(18826, 10) CFD discovered through testing that we made a mistake in defining the option "TOWHERE" immediately preceding "DSEL", which itself has an optional alternative in it. This is not parseable, as he has pointed out, because in effect we have defined two consecutive optional fields. A separate meeting is being held to resolve this conflict. I propose that at that meeting we also bring up and resolve some of the above issues as well, especially that of "SSEL". 3i

18978 Distribution

Richard W. Watson, James E. (Jim) White, Elizabeth J. (Jake) Feinler, Harvey G. Lehtman, Kirk E. Kelley, Laura E. Gould, N. Dean Meyer, Jeanne M. Beck, Charles F. Dornbush, Dirk H. Van Nouhuys, Michael D. Kudlick, Diane S. Kaye, James C. Norton, Kirk E. Kelley, Harvey G. Lehtman, Elizabeth J. (Jake) Feinler, Jeanne B. North, Michael D. Kudlick, Charles H. Irby,

More on NLS Command Language Syntax for HELP Users

(J18978) 10-SEP-73 16:38; Title: Author(s): Michael D. Kudlick/MDK;
Distribution: /RWW JEW DIRT NIC-QUERY; Sub-Collections: SRI-ARC DIRT
NIC-QUERY; Clerk: MDK;
Origin: <KUDLICK>SYNX.NLS;8, 10-SEP-73 16:31 MDK ;

KIRK 10-SEP-73 17:41 18979

Appropriate location for SIGART in the group allocation scheme.

SIGART should be in the network group rather than NIC shouldn't it? 1

18979 Distribution

Michael D. Kudlick, James C. Norton, Ferg R. Ferguson, L. Stephen Coles, Richard E. Fikes,

KIRK 10-SEP-73 17:41 18979

Appropriate location for SIGART in the group allocation scheme.

(J18979) 10-SEP-73 17:41; Title: Author(s): Kirk E. Kelley/KIRK; Distribution: /MDK JCN WRF LSC REF; Sub-Collections: SRI-ARC; Clerk: KIRK;

DVN 10-SEP-73 22:21 18980

People's Computer Center Meeting on Computer Aided Instruction This Thursday in Menlo Park

The People's Computer Center workes near hear (1919 Menalto, Menlo Park) to disseminate computer services to the genral public, particularlly as used in teaching elementary and highschool kids. 1

Some of you may recall their newsletter distributed here a few months ago. 1a

This thursday at 3:00 PM they are having one of a continuing series of meetings on computer-aided instruction. The meeting will cover: 2

Future sites and schdules; the need for volunteers to sponsor meetings; possible financing problems. 2a

Questions to the People's Computer Company staff about how the company works, who uses recources and how, what's avialabe, etc. 2b

Computerland for Time Travelers, a computer fair at Lawrence Hall of Science Septermber 20-23. 2c

Browsing and conversation...guests are invited to stay for Thursday nite open house. 2d

The Center is on the corner of Menalto and Gilbert. Go east from Midlefield on Willow to Gilbert, turn right on Gilbert, and go a few blocks. 3

I have posted a fact sheet on the cork bulleten board. For more information, seek Phyllis Cole, downstairs at SRI. 4

18980 Distribution

James H. Bair, Laura E. Gould, Nancy J. Neigus, L. Peter Deutsch, Alan C. Kay, Thomas O'Sullivan, Sally McLellan, K. Diane Shaw, Mario C. Grignetti,
Jeanne M. Leavitt, Rodney A. Bondurant, Jeanne M. Beck, Mark Alexander Beach, Judy D. Cooke, Marcia Lynn Keeney, Carol B. Guilbault, Susan R. Lee, Elizabeth K. Michael, Charles F. Dornbush, Elizabeth J. (Jake) Feinler, Kirk E. Kelley, N. Dean Meyer, James E. (Jim) White, Diane S. Kaye, Paul Rech, Michael D. Kudlick, Ferg R. Ferguson, Douglas C. Engelbart, Beauregard A. Hardeman, Martin E. Hardy, J. D. Hopper, Charles H. Irby, Mil E. Jernigan, Harvey G. Lehtman, Jeanne B. North, James C. Norton, Jeffrey C. Peters, Jake Ratliff, Edwin K. Van De Riet, Dirk H. Van Nouhuys, Kenneth E. (Ken) Victor, Donald C. (Smokey) Wallace, Richard W. Watson, Don I. Andrews

DVN 10-SEP-73 22:21 18980

People's Computer Center Meeting on Computer Aided Instruction This Thursday in Menlo Park

(J18980) 10-SEP-73 22:21; Title: Author(s): Dirk H. Van Nouhuys/DVN;
Distribution: /SRI-ARC JHB LEG NJN LPD ACK TO SM2 KDS MCG;
Sub-Collections: SRI-ARC; Clerk: DVN;

JBN 11-SEP-73 03:45 18981

Request for Replies on NIC Questionnaire

To: NIC Users

From: Jeanne North
Network Information Center
Stanford Research Institute
Menlo Park, Calif. 94025

Re: Questionnaire on NIC Publications

Several of NIC's users have replied to the questionnaire in the ARPANET NEWS for June regarding the Network Directory and the Catalog of the NIC Collection. Thanks to those who have replied; your answers are very thoughtful and will be helpful.

However, not enough replies have been received to give us a strong base for certainty as to which aspects of the documents are useful, which are not useful, and what improvements could be made. If you have not replied, would you please take the time to print out the rest of this item, and mark the boxes and mail to me right away. This is especially important if you have changes to suggest.

Request for Replies on NIC Questionnaire

QUESTIONNAIRE on NIC Publications

Your name..... Organization.....

1. Please check applicable boxes:

NIC docs in hardcopy	own copy	have access	use 1/month	use more	use less
Directory of Participants					
Current Catalog					

2. Check level of use you make of each section of the Directory:

Directory of Participants	indis-pensable	very useful	useful	not useful	no opinion
Individuals, Brief (Name, phone)					
Individuals, Full entry					
Groups (name, address etc., of all members)					

Request for Replies on NIC Questionnaire

Index of Idents	:..:	:..:	:..:	:..:	:..:
Organizations (name, address of org, with names of people)	:..:	:..:	:..:	:..:	:..:

Would you miss the listings of people in each organization if they were discontinued?

Comments about Directory.....

3. Check level of use you make of each section of the Catalog:

Current Catalog of the NIC Collection	indis-pensable	very useful	useful	not useful	no opinion
Author Index	:..:	:..:	:..:	:..:	:..:
Number Index	:..:	:..:	:..:	:..:	:..:
Titleword Index	:..:	:..:	:..:	:..:	:..:
Listing (with abstracts)	:..:	:..:	:..:	:..:	:..:

Would you miss the abstracts if the Listing were discontinued?

Are RFC's almost the only items you refer to in the Catalog?

Comments about Catalog Indexes and Listing.....

18981 Distribution

Joseph B. Reid, William T. Misencik, Toshiyuki Sakai, Louis Pouzin, Yngvar Lundh, Robert H. Hinckley, Marvin Zelkowitz, Don D. Cowan, Louis F. Dixon, Michael O'Malley, Peter Kirstein, David J. Farber, Dave Twyver, Art J. Bernstein, Dave E. Liddle, A. Kenneth Showalter, D. D. Aufenkamp, Derek Leslie Arthur Barber, Tjaart Schipper, Richard M. Van Slyke, E. M. Aupperle, Hubert Lipinski, Robert F. Hargraves, C. D. (Terry) Shephard, Maurice P. Brown, Robert L. Ashenhurst, Michael D. Kudlick, Richard W. Watson,
Gregory P. Hicks, Gloria Jean Maxey, Roberta J. Peeler, Craig Fields, Ermalee R. McCauley, Margaret Iwamoto, Dee Larson, Robert E. Doane, Brenda Monroe, Jeanne B. North, Pam J. Klotz Cutler, Barbara Barnett, Stan Golding, Steve G. Chipman, John P. Barden, Martha A. Ginsberg, Shirley W. Watkins, Janet W. Troxel, Connie D. Rosewall, Anita L. Coley, Carol J. Mostrom, Harold F. Arthur, Peter R. Radford, Wayne R. Robey, Joshua Lederberg, Connie Hoog, James A. Blumke, David Hsiao, Michael L. Marrah, Vinton G. Cerf, Gerald L. Kinnison, Paul Baran, Henry Chauncey, J. T. Sartain, Robert N. Lieberman, Ralph Alter, Nils Maras, Philip H. Enslow, Robert M. Dunn
William K. Pratt, David C. Evans, Douglas C. Engelbart, Bertram Raphael, Daniel L. Slotnick, Carolyn E. Taynai, Easter D. Russell, Leonard B. Fall, Peggy D. Irving, Roy Levin, M. P. McCluskey, Pitts Jarvis, Barbara A. Nicholas, Jacquie A. Priest, Terence E. Devine, Paul M. Rubin, Paula L. Cotter, O. A. Hansen, Dan Dechatelets, Nancy C. Thies, Robert Silberski, Marcia Lynn Keeney, Margaret A. (Maggie) Bassett, J. A. Smith, Leina M. Boone, Diana L. Jones, Nancy J. Neigus, Terry Sack, Frances A. (Toni) McHale, Lucille C. (Lucy) Gilliard, Ed J. Collins, Gary Blunck, John F. Heafner, Kathy Beaman, David J. King, C. Jane Moody, Sue Pitkin, Jerry Fitzsimmons
Glenn J. Culler, Frank S. Cooper, Bruce G. Buchanan, Kenneth L. Bowles, Morton I. Bernstein, Paul Baran, Saul Amarel, Roy C. Amara, John E. Savage, Butler W. Lampson, William R. Sutherland, Thomas G. Stockham, Gene Raichelson, Michael O'Malley, Peter G. Neumann, Marvin Minsky, Robert E. Millstein, J. C. R. Licklider, Robert M. Balzer, Herbert B. Baskin, Robert P. Abbott, Peter Kirstein, William B. Kehl, Roland F. Bryan, James G. Mitchell, Jeanne B. North, Allen Newell, John McCarthy, Lawrence G. Roberts, Frank E. Heart, Edward L. Glaser, Thomas M. Marill, T. E. Cheatham, James W. Forgie, Keith W. Uncapher, Edward A. Feigenbaum, Leonard Kleinrock
Michael B. Young, Michael A. Padlipsky, Schuyler Stevenson, L. Peter Deutsch, John Davidson, Thomas O'Sullivan, Sol F. Seroussi, Scott Bradner, Robert H. Thomas, Michael J. Romanelli, Ronald M. Stoughton, A. D. (Buz) Owen, Robert L. Fink, Jeanne B. North, Steve D. Crocker, Thomas F. Lawrence, John W. McConnell, James E. (Jim) White, A. Wayne Hathaway, Patrick W. Foulk, Richard A. Winter, Harold R. Van Zoeren, Alex A. McKenzie, Abhay K. Bhushan, B. Michael Wilber, Edward A. Feigenbaum, Robert T. Braden, James M. Pepin, John T. Melvin, Joshua Lederberg, Paul J. Nikolai, Robert J. Gronek, Rein Turn, Mark Medress, Franklin Kuo, Howard Frank, Robert L. Fink
John F. Wakerly, Tom C. Rindfleisch, Leonard B. Fall, David L. Hyde, Gary Blunck, Tom P. Milke, Alan H. Wells, Chuck R. Pierson, Carl M. Ellison, Robert P. Blanc, Jay R. Walton, Terence E. Devine, David J. King, William L. Andrews, Milton H. Reese, Kenneth M. Brandon, Lou C.

Nelson, Jeffrey P. Golden, Richard B. Neely, Dan Odom, Ralph E. Gorin, Robert G. Merryman, P. Tveitane, Adrian V. Stokes, David L. Retz, Reg E. Martin, Gene Leichner, Jean Iseli, James E. (JED) Donnelley, William Kantrowitz, Michael S. Wolfberg, Yeshiah S. Feinroth, Anthony C. Hearn, Eric F. Harslem, Robert M. (Bob) Metcalfe, Bradley A. Reussow, Daniel L. Kadunce, George N. Petregal

KNS 11-SEP-73 11:13 18983

PCI/BPO MEETING

DEAR DR. ALTER
THANK YOU FOR YOUR MESSAGEWHICH I RECEIVED OK.
I WILL SEE YOU AND MR FOSTER ON THE MORNING OF
THE 25/SEPT.
KIND REGARDS.
KEITH SANDUM

1

18983 Distribution
Ralph Alter,

VGC 11-SEP-73 12:11 18984

rsponse to JBN prompt

Jeanne, thanks for setting up su-dsl account for me. I thought I had turned in the NIC questionnaire on network directory and catalog. If you didn't get a copy of my filled out form, let me know and I will do another for you. Vint

1

18984 Distribution
Jeanne B. North,

MDK 11-SEP-73 13:42 18985

Judy ... Please ask Peter Deutsch (LPD) if he still wants us to make his Journal deliveryy be "Onliae" at the NIC, rather than "Netw rk Online" as it is now. If so, would you please take care of that change? Thanks ... Mike.

1

18985 Distribution
Judy D. Cooke,

JDH 11-SEP-73 16:15 18986

On-line Host field in ident system

My proposal buried in (MJOURNAL, 18800, 1:w) on Interim Dual-site Ident System to "temporarily" implement "On-line host" or "NLS host" in the ident system has gotten by without comment so far. I'm assuming silence is consent. 1

18986 Distribution
James E. (Jim) White, Charles H. Irby, Diane S. Kaye,

JLM 11-SEP-73 18:23 18987

man i have finally got this to a few peopple i quit

11 Sept.-SADPR-85 1

Opening remarks by Lt Col o,Keffe 1a

He stated that their was a lot of interest at the high levels in fact he apparently briefed yesterday a group of high level officers on the study.They stressed that the computer must be used to better help the AF to manage their resources 1a1

The cuurent base level machines are running out of gas and room,ie the 3500,s and the 1050,s 1a2

To upgrade these systems it is important that it be done in the context of a overall plan-thus the sadpr study was born 1a2a

He stated that the current systems were designed and implemented on a functional basis and that it may have been okay then but it is no longer acceptible and certanily not efficent. 1a2b

Stated that the study was to deal with base level business and their were emerging technology,s around such as texteditors and communication systems like the ARPA net which offered the AF a much more efficent and powereful way of doing busiiness. 1a2c

He stated thaat the SIALOG was considered a good study as a point of departure as well as the Base Comm Study. 1a3

The following is the originaziton of the study. 1b

Director-lt Col O,Keeefe 1b1

Dep Dir Lt Col Hoffman from the Data Design center 1b2

Requirements-Major Zara 1b3

Concepts and Technology-Lt Col Conrraty 1b4

Resources-Mr zenlea 1b5

mitre project officer-j Mitchell 1b6

He stated that j mitchel had prepared a 1980 tchnology forecast which was just published 1b7

He also stated which is probably most significant that the study would result in a DAR which would then be implemented.As

i read the plan it does imply r&d though I am not too convinced at this point if they really mean it. 1b8

I was amazed to observe that I and frank are the only troups from the r&d side of the house.It is heavily manned by stems design center peopple.. 1b9

They intend to use the redactron system for the prepartion of the report,which is encouraging but i am nervous about their willigness to truly look at the 1980 time frame. 1b10

18987 Distribution

Frank J. Tomaini, Duane L. Stone, Edmund J. Kennedy, Richard H. Thayer, William P. Bethke,