APPENDIX A

SNOBOL

A String Oriented Language

GPO PRICE $ _____

CFSTI PRICE(S) $ _____

Hard copy (HC) 3.00

Microfiche (MF) _____

ff 653 July 65

September 1967

New Mexico State University

University Park, New Mexico

Phone: 646-4108

The STRIGOL language was designed and implemented by Edward H. Harris. The work was partially supported by a grant from the National Aeronautics and Space Administration Sustaining Grant, NGR-32-003-027.

INTRODUCTION

Since the late 1950's, a number of compiler-level programming languages have been developed for use in primarily non-numerical applications. These languages have been of two types: list processing languages such as LISP, IPL-V, and L-6, and string manipulation languages such as COMIT and SNOBOL. The STRIGOL programming language has been developed in order to meet the growing need for string manipulation languages. The basic operations in STRIGOL are designed to facilitate the examination and manipulation of strings of data. In addition, limited arithmetic capabilities and internal subroutine capabilities are available. Three sample programs which are included in Appendix A illustrate the types of problems which STRIGOL is best suited for: example 1 is a program to find the longest common segment of two lists of data; example 2 performs a translation from the normal infix notation used in an arithmetic expression into reverse Polish notation, such as is done during the compilation process; example 3 is a program to edit text into a more readable format, and thus illustrates one application of STRIGOL in the area of linguistic analysis. The goal throughout the entire development of STRIGOL has been to design a language which is simple to learn and use, but which provides sufficient power for all types of symbol manipulation problems.

This manual is intended as a reference manual for the STRIGOL programming language, in particular, the CDC 3300 implementation currently in use at New Mexico State University. It is assumed that the user of this manual is already familiar with STRIGOL. Included is a complete description of the STRIGOL language, the operating environment of the CDC 3300 implementation, and a complete explanation of all compilation and execution time messages. The appendices contain sample STRIGOL programs, a description of the internal specifications of the language to enable the 3300 COMPASS program to use the STRIGOL subroutines in a COMPASS program, and a formal description of the STRIGOL scanning algorithm.

NOTATION

All notation used in this manual is exactly as is used in the CDC 3300 implementation with the exception of the quotation mark used to delimit literals. Because the quotation mark (") does not appear in the CDC 3300 character set, the not-equal sign (≠) is used in its place in the actual implementation.

Contents

I. PROPERTIES OF 3300 STRIGOL

A STRIGOL program consists of a series of statements begun by a PROGRAM statement and terminated by an END statement. Statements are normally executed sequentially; however, this can be changed based on logical decisions during execution.

1.1 Character Set

The STRIGOL character set is composed of letters, digits, and special characters. The alphabetic characters are the letters A through Z; the digits are the characters 0 through 9; the alphabetic characters together with the digits are called the alphameric characters.

There are 17 special characters which have specific meaning to the STRIGOL compiler. The names and graphics for these special characters are:

Name	Graphic
Equal or Replacement symbol	=
Plus	+
Minus	-
Asterisk or Multiply symbol	*
Slash or Divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Quotation mark or not equal symbol	≠
Left bracket	[
Right bracket	]
Less than symbol	<
Greater than symbol	>
Less than or equal symbol	≤
Greater than or equal symbol	≥
Dollar sign	$

In addition to these characters, any character in the CDC 3300 character set can be used as a data character.

The special characters which have meaning to the STRIGOL compiler are divided into three classes: operators, delimiters, and separators.

Operators

Operators used in STRIGOL are divided into three types: arithmetic operators, comparison operators, and string operators.

The arithmetic operators are:

+ denoting addition
- denoting subtraction
* denoting multiplication
/ denoting division

The comparison operators are:

= denoting equal to
≠ denoting not equal to
> denoting greater than
< denoting less than
≥ denoting greater than or equal to
≤ denoting less than or equal to

The string operator is:

, denoting concatenation

Delimiters

The delimiters in STRIGOL are:

≠ encloses string constants
( ) denotes a filler in a SCAN statement
[ ] denotes the scanning pattern in a SCAN statement

Separators

The separators in STRIGOL are:

= used to indicate replacement in replacement and SCAN statements
$ indicates a substring variable
/ separates branch field in READ, SCAN, BACKSCAN, and IF statements

1.2 Identifiers

An identifier is a string of alphameric characters, not contained in a string constant or comment. The first character must be alphabetic, and the total length of an identifier must not be greater than 8 characters. Imbedded blanks are not considered to be a part of an identifier.

Identifiers are used in STRIGOL for the following:

string names
numerical variable names
statement labels
routine names
program name

1.3 Statement Format

STRIGOL source statements are normally written on a coding sheet one to a line in columns 10-72. If a statement is too long to fit on one line, it may be continued by putting any character other than a blank or a "/" (slash) in column 9 of every line which is to be continued except the last line. (Note: This is the opposite of how continuation lines are indicated in FORTRAN). There is no limit to the number of continuation lines.

Columns 1-8 in the first line of a source statement may contain a statement label to allow the statement to be referenced elsewhere in the program. Statement labels

must conform to the rules for identifiers and cannot be used elsewhere in the program as variable names, routine names, or as other statement labels. The statement label may appear anywhere in columns 1-8 and may contain blanks; however, any imbedded blanks are not considered to be a part of the statement label. If continuation lines are labeled, the statement label will be printed during compilation, but they will be ignored.

Columns 73-80 are not examined by the STRIGOL compiler and may therefore be used for program identification, sequencing, or any other purpose.

Comments to explain the program may be written in columns 2-72 of a line if an asterisk is placed in column 1. Comment lines may appear anywhere in the program before the END statement, except that they may not appear in the middle of a statement containing continuation lines. The comments are not processed by the STRIGOL compiler, but are listed during the compilation process. Comment lines may not be continued.

As a special feature of 3300 STRIGOL it is possible to intermix STRIGOL statements and COMPASS statements in a STRIGOL program. If the character "/" (slash) is placed in column 9 in a statement, it is interpreted by the STRIGOL compiler to be a COMPASS statement. There is no restriction as to the type of COMPASS statements which can appear within a STRIGOL program except that a COMPASS statement cannot be the first statement in the program and it must appear before the END statement. For the condition of the various registers before and after calls to the STRIGOL execution time subroutines, the programmer is referred to Appendix B of this manual.

With the exception of COMPASS statements, all statements in a STRIGOL are completely free formatted within the allowed columns. Blanks may included or left out arbitrarily within a source statement.

II. CONSTANTS, VARIABLES, FUNCTIONS AND EXPRESSIONS

2.1 Constants

Numerical Constants

A numerical constant in 3300 STRIGOL is any integer within the range $-2^{23}+1$ to $+2^{23}-1$. The sign is optional if the number is positive, and if left unsigned, a numerical constant is assumed to be positive.

Examples

```
      3
     +1
     -0
 -12345
```

String Constants

A string constant or literal is any continguous string of letters, digits, special characters, or blanks enclosed in quotation marks ("). A literal must contain at least one character, and if it is more than one character long, it cannot contain a quotation mark. There is no maximum length restriction for a literal.

Examples

```
"123.45XYZ$"
"""
"THIS IS A STRING CONSTANT"
```

2.2 Variables

A variable name must conform to the rules for identifiers. All numerical variables must be declared as such (see NUMBER statement); any variable not declared as numerical will be assumed to be the name of a string. The same variable name, consequently cannot be used for both a numerical variable and a string name. All numerical variables have the value zero and all strings are null at the beginning of program execution.

Examples

```
A
NAME1
A1X1Y2
R1234567
```

Substring Variables

General Form:

```
name$n
```

where name is a string variable and n is a numerical variable or unsigned numerical constant. By using substring variables, only the first n characters in name are accessed. If the value of n is greater than the length of name, the entire string is accessed. To illustrate, if:

STR = "12345"
then the value of STR$3 is "123", and
the value of STR$7 is "12345".

If n is equal to zero, name$n will act as a null string.

2.3 Functions

Although there are no facilities in STRIGOL for allowing user definition of functions, a number of basic functions are included in the STRIGOL I language.

Numerical Functions

The numerical functions LENGTH and NUMBER return numbers as values. These functions are used as arguments in IF statements and in numerical expressions.

LENGTH Function
General Form:

LENGTH(name)

where name is a simple string variable. This function returns the number of characters in the string name as its value. If the string is null, the value of the function is zero.

NUMBER Function
General Form:

NUMBER(name)

where name is a simple string variable or a substring variable consisting only of numeric characters with an optional leading sign character. If name is not numeric or is a null string, an error message will be printed and the value returned will be zero. Any blanks in the string will be treated as zeros; consequently, the string "3 " upon conversion to a number would have the value 30. If a sign is specified, it must be the first character in the string. To illustrate; if:

N1 has the value "-163XYZ"
then the value of NUMBER(N1$4) is -163.

Boolean Functions

Boolean functions are functions which return the value TRUE or FALSE, and can only be used as arguments in IF statements.

NULL Function
General Form:

NULL(name)

where name is a simple string variable. The function has the value TRUE if name is a null string, and FALSE if it contains one or more characters.

NUMERIC Function
General Form:

```
NUMERIC(name)
```

where name is a simple string variable or a substring variable. The function will return the value TRUE if name consists of a possible leading plus or minus sign followed only by numeric characters or blanks. If a sign is specified it must be the first digit in the string; it cannot be preceeded by blanks. The following strings or substrings are examples of valid numerical strings:

```
     5
    +10
000397
   -37
-   3
    +0
```

The following strings are not numerical and consequently the function NUMERIC would return the value FALSE:

```
     +
    +A
3.27E-2
   3.17
 10,000
```

String Functions

The string function STRING is used to convert numbers into strings, primarily for use in output. The STRING function can be used only in string expressions.

STRING Function
General Form:

```
STRING(numb)
    or
STRING(n,numb)
```

where n is a numerical constant or variable and numb is a numerical variable. Using the first form of the STRING function, the resulting string will contain no leading zeros or blanks, and will have a leading sign only if numb has a negative value. Some examples of the first form of the STRING function follow:

```
N=-1,          STRING(N) = "-1"
N=127,         STRING(N) = "127"
N=+127,        STRING(N) = "127"
N=0            STRING(N) = "0"
```

Using the second form of the STRING function, the value of the function is a string n characters long containing a leading sign character only if numb is negative and containing leading zeros if the number of digits in numb is less than n. For example, if:

NUMB1 has the value -12

then the value of STRING(5,NUMB1) is "-0012". If the number of digits in numb is greater than n, an error message is printed and the least significant digits are discarded. In the preceeding example, the value of STRING(2,NUMB1) would be "-1".

If the value of n is negative, the value of the function is a string whose length is the absolute value of n and containing leading blanks instead of zeros. In the previous example, the value of STRING(-4,NUMB1) would be "- 12". If the length parameter is a numerical variable, it may not be preceeded by a minus sign.

The following examples illustrate the second form of the STRING function. To illustrate, if:

NUM=-3,	STRING(2,NUM)="-3"
NUM=-3,	STRING(-2,NUM)="-3"
NUM=+0	STRING(-3,NUM)=" 0"
N=7,NUM=1	STRING(-N,NUM) is illegal
NUM=123	STRING(0,NUM)="123"
NUM=123	STRING(-2,NUM)="12"

2.4 Expressions

Numerical Expressions

Numerical expressions in 3300 STRIGOL are formed by combining numerical constants, numerical variables, and calls to the numerical functions LENGTH and NUMBER using the arithmetic operators +,-,*,/ denoting addition, subtraction, multiplication, and division, respectively. During evaluation of a numerical expression all arithmetic operations are truncated rather than rounded. Thus, the value of

```
5/2
```

is 2 and the value of

```
5/3
```

is 1 rather than being rounded to 2 as might be expected.

Normally, numerical expressions are evaluated from left to right with multiplication and division taking precedence over addition and subtraction; however, by using parentheses to obtain desired groupings, the normal evaluation procedure can be overriden.

Thus

```
A/B*C
```

is evaluated as

```
(A/B)*C
```

rather than

```
A/(B*C)
```

as might be expected.

The following sequence of statements illustrates the evaluation of numerical expressions:

```
ALPHA = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
N = LENGTH(ALPHA)+1
M = N*((N-7)/2)
K = -(M*N/M/N)+7
```

As a result of evaluation of these statements, N would have the value 27, M the value 270, and K the value 6.

Other examples of valid numerical expressions are:

```
1
A-3
AXY+LENGTH(STR)
LENGTH(STX)+3*(-NUMBER(X$3)+R)
```

String Expressions

A string expression in STRIGOL consists of a string constant, a string variable, a substring variable, a call to the string-valued function STRING, or any combination of these separated by commas, where the use of a comma indicates concatenation. To illustrate, if:

```
S1="GRASS123"
```

and

```
S2="HOPPER"
```

then the expression

```
S1$5,S2
```

has the value "GRASSHOPPER".

Examples

```
"1"
"A",B
"S1",S2,STRING(N)
" ",WORD,"""","WORD",""""
X$17,"X$17",STRING(3,LX),"STRING(3,LX)",X$N,"."
```

III. DECLARATION STATEMENTS

3.1 NUMBER Statement

General Form:

```
NUMBER name1,name2,...
```

The NUMBER statement is a declarative statement which specifies that name1,name2,... are numerical variables, and thus cannot be used as string names. All numerical variables must be declared as such in a NUMBER statement. NUMBER statements can appear anywhere within a STRIGOL program; however, any numerical variable used in a STRIGOL source statement must be preceeded by a NUMBER statement declaring the variable name.

Examples

```
NUMBER N
NUMBER S1,A,R,XYZ,X
```

3.2 PROGRAM Statement

General Form:

```
PROGRAM name
```

where name must conform to the rules for identifiers. The PROGRAM statement indicates the beginning of a 3300 STRIGOL source program and consequently must be the first statement in a STRIGOL program. Name cannot appear anywhere within the program as a numerical variable, string name, statement label, or as the name of a routine.

3.3 END Statement

General Form:

```
END
```

The END statement indicates the end of a STRIGOL source program and must therefore be the last card in the source deck.

Execution of the program begins with the first executable statement providing no serious errors were encountered during compilation. If the program falls into the END statement, or if it is labeled and control is transferred to it, an exit back to the operating system occurs.

IV. ARITHMETIC REPLACEMENT STATEMENT

General Form:

```
nv=ne
```

where nv is a numeric variable and ne is a numerical expression. The current value of nv is replaced by the value of the numerical expression ne after it has been evaluated according to the previously stated rules for evaluation of numerical expressions; thus, after execution of the following two statements, the value of N would be 2:

```
N=1
N=N+1
```

Other examples of valid substitution statements are:

```
A=-17
RXZ=NUMBER(X$RX2)-7
Z=Z*(Z/2+7-(N*(6-R))/2)
```

V. CONTROL STATEMENTS

Control statements in STRIGOL are used to alter the normal order of execution of statements within a STRIGOL program.

5.1 GO TO Statement

General Form:

GO TO label

Execution of this statement causes the statement labeled label to be the next statement executed in a STRIGOL program.

Examples

GO TO A
Go To STATE12

5.2 EXIT Statement

General Form:

EXIT

This statement stops execution of a 3300 STRIGOL program and causes control to be returned to the operating system.

5.3 IF Statement

General Form:

IF(arg) /t,f

where arg is a Boolean relation or Boolean function, and t and f are statement labels.

Execution of this statement causes a transfer to the statement labeled t if ARG has the value TRUE and to f if arg has the value FALSE. Either the t or f branch of the IF statement can be omitted; if control is transferred to the omitted branch, program control will fall through to the next executable statement.

The functions NULL and NUMERIC can be used for arg, or it can be a Boolean relation of the form

a ℓ b

where a and b are numerical constants, numerical variables, or calls to the numerical functions LENGTH or NUMBER; the operands a and b cannot be more complicated numerical expressions containing arithmetic operators. ℓ is one of the Boolean operators $=, \neq, >, <, \geq, \leq$. The value of the IF statement is TRUE if, respectively, a is equal to b, not equal to b, greater than b, less than b, greater than or equal to be, or less than or equal to b.

Examples

```
IF(NULL(X))                    /A,LOOP
IF(NUMERIC(S$N))               /,REPEAT
IF(A=7)                        /EXIT
IF(LENGTH(Z)≥13)               /N,M
IF(NUMBER(X$Q)<LENGTH(X))      /,DONE
```

VI. STRING PROCESSING STATEMENTS

String processing statements in STRIGOL are used to create, delete, examine, and alter strings.

6.1 Replacement Statement

General Form:

```
sn=se
```

where sn is a simple string variable or a substring variable. This statement replaces the contents of the string sn with the value of the string expression se. If sn is a substring variable of the form s$n, only the first n characters of the string s are replaced by se. For example, the following replacement statement would delete the first three characters from the string called XYZ:

```
XYZ$3=
```

In the following example, N is assumed to be a numeric variable:

```
N=0
STR = "ABC"
STR$N = "123"
STR$4 = STR,STR$N, "X"
```

After executing these statements, the value of STR would be "123ABCXBC".

The third statement

```
STR$N = "123"
```

appends "123" to the beginning of STR. In the last statement, STR$N acts as a null string.

Examples

```
S="1"
LINE$3=LINE,"LINE$M","X"
A="""","B",""""
TEXT="DOG",STRING(NN),"CAT",STRING(M)
```

6.2 SCAN Statement

General Form:

```
SCAN name [sel] = se2                    /s,f
```

Where name is called the scan reference and is a simple string variable or substring variable, sel and se2 are string expressions, and s and f are statement labels. The string expression sel is called the scanning pattern.

The SCAN statement is used to examine a string for a succession of substrings of a specified form. In the generic case, name is the string being examined and sel is the string expression name is being examined for. To illustrate the use of the SCAN statement, the following statement would examine the string called INPUT1 for an occurrence of the string "CAT":

```
SCAN INPUT1 ["CAT"]
```

The success or failure of a SCAN statement can affect program flow and can have other consequences which will be described in detail later.

Since a scan of this type can either succeed or fail, a method is available to control the execution of a STRIGOL program based on the success or failure of a SCAN statement. For example, the following SCAN statement will examine the string called TEXT for the occurrence of the string expression

```
" ",WORD," "
```

and will branch to the statement labeled X if the scan was successful, and to the statement labeled Y if the scan was unsuccessful:

```
SCAN TEXT [" ",WORD," "]                    /X,Y
```

By using substring variables, even more power can be obtained in a SCAN statement. For example, the SCAN statement

```
SCAN WORDS$3 ["CAT"]
```

would examine the first three characters in WORDS for the three characters in the literal "CAT". The SCAN statement

```
SCAN XYZ$4 [STR$5]
```

will fail regardless of the length of XYZ unless STR has less than five characters. Another example of the use of substring variables in a SCAN statement is:

```
SCAN CARD$10 [CARD$9," "]                   /LOOP
```

In the above statement, control would be transferred to the statement labeled LOOP only if the tenth character in the string CARD is a blank.

It is often desirable in examining strings to do more complex pattern analysis than is possible using ordinary string concatenation.

By using a feature called a "filler", a greater generality may be obtained in the SCAN statement. There are two types of fillers, unrestricted fillers which are denoted by string names bounded by parentheses, and fixed-length fillers which are denoted by substring variables bounded by parentheses. A typical example of the use of unrestricted fillers would be to examine the string LINE1 for an occurrence of the literals "AB" and "Y" not necessarily adjacent in LINE1. An unrestricted filler can match any string; consequently, the SCAN statement

```
SCAN LINE1 ["AB",(ANYTHING),"Y"]
```

would accomplish the desired scan. If the value of LINE1 is ABBAXYZ, then the scan would succeed with the filler ANYTHING matching the characters BAX.

As one of the consequences of a successful scan, all fillers are given the value of the portions of the string they match. In the preceeding example, ANYTHING would be given the value BAX exactly as if the replacement statement

```
ANYTHING = "BAX"
```

had been executed. Because of this, the scan reference cannot also be used as a filler in the same SCAN statement.

A fixed-length filler can match any string of a specified number of characters. The first four characters in the string LIST could be given the name FIRST4 by the SCAN statement.

```
SCAN LIST [(FIRST4$4)]
```

If N had the value 4, the statement could have been written

```
SCAN LIST [(FIRST4$N)]
```

As a second example, consider the following statements:

```
VOWELS = "AEIOU"
SCAN VOWELS [(V1$1),(V2$1),(V3$1),(V4$1),(V5$1)]
```

The scan would succeed, and V1,V2,V3,V4, and V5 would be given the values A,E,I,O,U, respectively.

In many applications it is necessary to alter the contents of a string based on the results of a scan. To do this, the string replacement statement is combined with the basic form of the SCAN statement thereby allowing replacement, deletion, or rearrangement of strings. In particular, if the scanning pattern is followed by a replacement sign and then by a string expression, the portion of the scan reference which is matched in a successful scan is replaced by the value of the string expression to the right of the replacement sign. If the value of the string expression is null or is omitted, the matched portion is deleted. For example, consider the following sequence of statements:

```
CARD = "QUEEN OF SPACES"
SCAN CARD ["QUEEN"] = "KING"
```

After executing these two statements, the value of CARD would be KING OF SPACES. The following example will delete all occurrences of the letter K from the string X and will then transfer control of the program to the statement labeled DONE:

```
DELETE SCAN X ["K"] =                    /DELETE,DONE
```

Thus, if success and failure options are specified, replacement is performed before transferring of program control. A more complicated use of the SCAN statement is illustrated by the following example:

```
EXP = "A+B"
SCAN EXP [(F1),"+",(F2)]=F1,F2,"+"
```

After executing these statements the value of EXP would be changed to AB+, the value of F1 would be A, and the value of F2 would be B.

Dynamic scanning is a special case in scanning in which tentatively defined fillers can be referred to dynamically during the scanning process. If a string or substring in the scanning pattern is the same name as a filler used to the

left of it in the pattern, the current value of the filler is used in the scan.

Dynamic scanning has not yet been fully implemented in this version of 3300 STRIGOL. An attempt to use dynamic scanning may result in an erroneous failure indication.

There are five phases to the actual scanning process:

i) evaluation of string expressions
ii) the actual scan
iii) assignment of values to fillers (naming)
iv) replacement
v) branching

i) Evaluation. Before beginning the actual scanning process, all fields are evaluated. Any calls to the STRING function are done, and temporary strings are created with appropriate values. Substring variables are checked for legality; no negative lengths are allowed. Also, all fillers are set to null strings, and their previous contents are released to available storage. If at any point in this procedure, a negative substring variable is encountered, an error message is printed, but evaluation continues with the invalid substring being treated as a null string. If the scan reference is an invalid substring, an error is printed and the job is aborted.

ii) Scanning. Elements of the scanning pattern must match consecutive portions of the scan reference. In general the scanning process can be described as follows:

> Scanning proceeds from left to right, each scanning pattern element matching the shortest possible portion of the scan reference depending upon the type of pattern element.

In more complicated cases, further clarification of the scanning process may be necessary. The following rules provide more detailed descriptions of the scanning process.

> a) The scanning process proceeds element by element from left to right starting at the first element of the scanning pattern. The elements must match consecutive portions of the scan reference.
>
> b) An attempt is made to match the first element starting at the first character in the scan reference. If this is not possible, an attempt is made starting at the second character, and so on. If this fails, the entire scan is unsuccessful.
>
> c) When an element is successfully matched, a forward scan is attempted for the next element in the scanning pattern.
>
> d) If at any point an element is not matched, rescan is attempted for the preceeding element if it is an unrestricted filler. Rescan is an attempt to lengthen the portion of the string reference being matched by a pattern element and is necessary because the entire scan cannot succeed with the previous match. If rescan fails, the entire procedure regresses one level and is retried again. If the beginning of the scanning pattern is reached, the first element in the scanning pattern is moved down one character, and the entire scan is restarted.

e) When the last element in the scanning pattern has been matched, the scan is successful. The scan fails if there is no match for the first element, or if at any point during forward scanning there is an insufficient number of characters in the scan reference to match a pattern element.

The various techniques of forward scanning and rescanning depend on the type of element involved in the scanning pattern. In all cases, however, the element must match a portion of the scan reference starting at the next character following the portion matched by the preceeding element. The details for each type of pattern element follow:

a) Strings. In this case, a string is considered to be a string variable, a literal, or a substring variable after it has been evaluated. In forward scanning, a string must match a portion of the scan reference which is identical to its value. If this fails due to insufficient length in the scan reference, no rescan is attempted, and the entire scan fails. A null string or null-valued substring always matches.

b) Fixed-length fillers. In forward scanning a fixed-length filler matches any substring of the length specified. If the scan reference is not long enough, no rescan is attempted, and the entire scan fails. If the length specification is zero, a fixed-length filler always matches.

c) Unrestricted fillers. During forward scanning, an unrestricted filler always matches a null portion of the scan reference. During rescanning, one character is added to the value of the filler previously matched. If the scan reference is of insufficient length, the entire scan is unsuccessful. As a result of this, once an unrestricted filler is encountered during a scan, no pattern element matched previously can everybe moved further along the value of the scan reference. If the match fails beyond the filler, there is no possibility that it can be made to succeed by rematching anything before the filler. Some consequences of this will be seen shortly. As a special case, if an unrestricted filler is the last element of the scanning pattern, rather than match a null portion of the scan reference, it will be extended to match the remaining value of the scan reference. Also as a consequence of the scanning algorithm, an unrestricted filler which appears at the beginning of the scanning pattern will match everything up to the first character matched by the second element of the scanning pattern. Therefore, if the only element in the scanning pattern is an unrestricted filler, it will match the entire value of the scan reference.

A more formal presentation of the scanning algorithm is given in Appendix C.

iii) Naming. If the scan fails, all fillers are set to a null value. If the scan succeeds, fillers are given the values associated with the portion of the scan reference they match. If the same name appears in more than one filler, the name is associated with the rightmost value of the filler. For example, consider the following statements:

```
X = "ABCDE"
SCAN X ["A",(F1),"C",(F1)]
```

After executing these statements, the value of F1 would be DE, not B.

iv) Replacement. If a scan is unsuccessful this phase is omitted. If a scan is successful, after naming is done, the portion of the scan reference matched by the scanning pattern is replaced by the value of the string expression to the right of the replacement sign. In the case where a terminating unrestricted filler has been extended to the end of the value of the scan reference, that portion of the scan reference is also accessed during replacement. Any fillers named during the scan which are also used during replacement will use the values given them as a result of the scan rather than their values before undertaking the scan.

v) Branching. If at any point during the scanning process a complete scan failure occurs, all fillers are nulled, no replacement is done, and the failure branch of the SCAN statement is taken. On the other hand, if a scan is successful, the success branch is taken after naming and replacement are done.

6.3 Examples of Scanning

The following examples illustrate some of the situation which can arise in scanning. Whenever fillers are used, their values upon completion of the scan are given.

Example 1:

```
X = "X Y Z"
SCAN X [(A)," ",(B)]
```

the scan succeeds with

```
A = "X"
B = "Y Z"
```

Example 2:

```
ALPHA = "ABCDEFGHIJKLMNOP"
SCAN ALPHA [(X$5),(Y),"K",(Z)]
```

The scan succeeds with

```
Y = "ABCDE"
Y = "FGHIJ"
Z = "LMNOP"
```

Notice that Z matches the rest of the scan reference ALPHA since it is an unrestricted filler and is the last element of the scanning pattern.

Example 3:

```
NUMBERS = "0123456789"
SCAN NUMBERS$6 ["78"]
```

The scan fails. Although the characters "78" do occur in the string NUMBERS, by restricted the scan reference to the first six characters in NUMBERS, the

portion of the string containing these characters is not accessed.

Example 4:

```
LIST = "A,B,C,D"
SCAN LIST [X$0] = "X,Y,"
```

The scan succeeds since a null-valued substring or a null string always matches. After replacement,

```
LIST = "X,Y,A,B,C,D"
```

Example 5:

```
DATA = "123."
SCAN DATA [(A),(NUM$3),"."]
```

The scan succeeds with

```
A null
NUM = "123"
```

Example 6:

```
STR = "HOUSE"
SCAN STR$3 [(A),(B),(C)]
```

The scan succeeds with

```
A null
B null
C = "HOU"
```

This illustrates a fact that may not be obvious from the formal scanning rules: if two or more unrestricted fillers occur together and the scan is successful, all of these fillers but the last will always end up null.

Example 7:

```
CHAR = "ABCD"
SCAN CHAR [(X$2),(Y$3)]
```

The scan fails since the scan reference is of insufficient length.

Example 8:

```
INPUT = "987654"
SCAN INPUT$5 [INPUT$4,"5"]
```

The scan succeeds. This technique provides a simple way of examining only a specified portion of the scan reference.

Example 9:

```
LIST = "A,B,C,D"
SCAN LIST [(X),",",(Y),","(X),",",(Y)]
```

The scan succeeds with

```
X = "C"
Y = "D"
```

The earlier defined values for X and Y are lost.

Example 10:

```
STR =
SCAN STR [(FIL)]
```

The scan succeeds with FIL null

6.4 BACKSCAN Statement

General Form:

```
BACKSCAN name [sel] = se2                    /s,f
```

where name, sel, se2, s and f are defined exactly as in the SCAN statement. The BACKSCAN statement is like the SCAN statement, except that it examines the scan reference in a reverse direction. It is very simple to understand the BACKSCAN statement if the following rules are used to think about it:

i) Reverse the order of the characters in the scan reference.

ii) Scan the string normally, doing all naming and replacement as in the SCAN statement.

iii) Reverse the order of the characters in the scan reference after replacement, but do not reverse the characters in fillers defined during the scan.

A few examples of backscanning will illustrate these points:

Example 1:

```
LABEL BACKSCAN CARD$1 [" "] =                /LABEL
```

This backscan will delete trailing blanks from the string called CARD.

Example 2:

```
DATA = "XYZ"
BACKSCAN DATA [(A)]
```

The backscan succeeds with

```
A = "ZYX"
```

Example 3:

```
INPUT = "ABC00"
BACKSCAN INPUT$4 ["00",(X)]
```

The backscan succeeds with

```
X = "CB"
```

Notice that in this case, INPUT$4 accesses the last four characters in INPUT.

Example 4:

```
Z = "ABC"
BACKSCAN Z [(X$1),(Y$2)] = Y,X
```

The scan succeeds with

```
X = "C"
Y = "BA"
Z = "CAB"
```

VII. INPUT-OUTPUT STATEMENTS

7.1 READ Statement

General Form:

```
READ(lun) Data                          /s,f
```

where lun is a numerical constant or variable within the range 1 to 60 giving the logical unit number of the input device, data is a simple string variable or a substring variable, and s and f are statement labels. The READ statement reads one physical record of data of up to 4096 (this number may be changed by using the B-option on the STRIGOL card) characters in length from the input device that has the logical unit number lun, and gives it the name data. The input record must be written in BCD. Upon completion of the read, control transfers to the statement labeled s unless an end-of-file was encountered during the read operation in which case control is transferred to statement f. Any failure of the input device such as a feed failure or parity error causes the input operation to be automatically retried up to a maximum of three times. If the operation still fails, an error is printed and the entire job aborted.

In the case of the READ statement, the use of a substring variable indicates a restriction on the maximum length of the input record. If the length specified is negative, an error is printed and the job is aborted.

If the length specified is zero, the entire read operation is inhibited and the end-of-file branch is selected. If the length specification is less than 4096, the maximum length of the input record is restricted to that length. If it is greater than 4096, the maximum record length remains at 4096 characters. If the length of the input record exceeds the maximum allocated buffer size, trailing characters will be lost.

Several default options are assumed in the READ statement. If lun is omitted, input is automatically taken from logical unit number 60, the standard system input unit. Either the s or f branch or both can also be omitted. If the omitted branch option is selected, control is transferred to the next executable statement. To illustrate the READ statement, the following statement will read 17 characters from the card reader, given them the name INPUT, and if an end-of-file is not encountered, control will be transferred to a statement labeled LOOP:

```
READ INPUT$17                           /LOOP
```

Examples

```
READ(01) A$N                            /X,Y
READ STRING
READ(INPUT) CARD                        /,DONE
```

7.2 WRITE, PRINT Statements

General Form:

```
WRITE(lun) se

     or

PRINT(lun) se
```

where lun is a numerical constant or variable with a value of between 1 and 61, and se is a string expression. If lun is omitted, output is automatically on logical unit number 61, the standard system output unit. The WRITE statement writes one physical record of data (up to 4096 characters) on logical unit number lun. The contents of the strings named in se remain unchanged. The following is an example of a WRITE statement that might be used to title a page of output:

```
WRITE "1",TITLE," ","PAGE",STRING(PAGE)
```

If SE is a null string, the WRITE statement is not executed.

If output is on the printer, the first character of the output record is used as a carriage control character and is not printed. The most commonly used carriage control characters and their meaning are:

Char.	Meaning
" "	Single space after printing.
"0"	Double space after printing.
"-"	Triple space after printing.
"1"	Page eject before printing, single space after printing.
"*"	No space after printing; the next line will be printed on top of this line.

If the output record is over 136 characters in length, the overflow of the output record will be single spaced on succeeding lines.

Examples

```
WRITE(62) "0EXAMPLE"
WRITE(02) A,B,"B",BL$N
PRINT "-N = ",STRING(3,N)
```

7.3 ENDFILE Statement

General Form:

```
ENDFILE lun
```

where lun is defined is in the READ statement except that it must be a magnetic tape unit. The ENDFILE statement writes a file mark on the magnetic tape unit specified by logical unit number lun.

7.4 REWIND, UNLOAD Statements

General Form:

```
REWIND lun
UNLOAD lun
```

where lun is defined as in the READ statement. These statements are used to rewind and unload the magnetic tape units specified by logical unit number lun. In either case, if the previous operation on the tape unit was a write, a file mark is written on the tape before it is rewound or unloaded.

VIII. ROUTINES

A routine in STRIGOL is a series of statements begun by a ROUTINE declaration statement, and ended by an ENDROUTINE declaration statement. Routines allow the user to execute a series of statements several times within a STRIGOL program without having to write the statements each time. A routine is entered from the main part of the program by using an EXECUTE statement, and control is returned through or branching to the ENDROUTINE statement. If a routine is encountered in the normal flow of program execution, the program will bypass the routine. Examples of routines and their uses will be given shortly.

8.1 ROUTINE Statement

General Form:

```
ROUTINE name
```

where name is an identifier specifying the name of this particular routine. The name used cannot appear elsewhere in the program as a variable name, a statement label, or as the program name. Routines can appear anywhere within the source program. Routine declarations cannot be nested, i.e., if a ROUTINE statement is encountered between another ROUTINE statement and its matching ENDROUTINE statement, it will be diagnosed as an error by the STRIGOL compiler.

8.2 EXECUTE Statement

General Form:

```
EXECUTE name
```

where name is the name of a routine and must therefore also appear in a ROUTINE declaration statement. This statement causes control to be transferred to the first statement in the routine called name. Execution of statements within the routine continues until a RETURN statement is encountered or until program control falls through to the ENDROUTINE statement; upon encountering either of these statements, control returns to the statement following the EXECUTE statement. It is permissible for an EXECUTE statement to appear within a routine as long as the routine being executed is not, either directly or indirectly, the name of the routine in which the EXECUTE statement appears. In other words, recursive calls to routines are not permitted in STRIGOL.

8.3 ENDROUTINE Statement

General Form:

```
ENDROUTINE
```

The ENDROUTINE statement is used to indicate the end of a routine. If no matching ROUTINE statement has been encountered prior to the ENDROUTINE statement, an error will be printed. If during execution of a routine, control falls through or is transferred to the ENDROUTINE statement, execution of the routine is terminated and main program execution continues from the statement following the EXECUTE statement which was used to enter the routine.

8.4 RETURN Statement

General Form:

```
RETURN
```

The RETURN statement is used to terminate execution of a routine and to return control back to the main part of the program. A RETURN statement can only occur within a routine; if encountered anywhere else, it will be diagnosed as an error. It is not necessary for a routine to have a RETURN statement, and it is also possible for a routine to have more than one RETURN statement.

A typical use for routines would be in special purpose input or output routines. For examples, consider the following routine:

```
ROUTINE WRITE01
N = 80-LENGTH(FIELD)
WRITE(01) FIELD,BLANKS$N
ENDROUTINE
```

This routine appends up to a maximum of eighty blanks onto the output record called FIELD and writes the resulting eighty character record on logical unit number 01. Here it is assumed that BLANKS is a string consisting entirely of blanks and is at least eighty characters in length.

Another use for routines is illustrated by the following example:

```
ROUTINE LINECHEK
LINES=LINES+1
IF (LINES≤55)                          /RET
EXECUTE TOTOTALER
PAGE=PAGE+1
WRITE "1",BLANKS$70,"PAGE",STRING(-3,PAGE)
RET RETURN
ENDROUTINE
```

This routine illustrates several features of routines. Simply by executing the statement

```
EXECUTE LINECHEK
```

the following things will occur: first, a counter called LINES is incremented; then, if the value of LINES is less than or equal to 55, control is returned to the main part of the program via the RETURN statement labeled RET; on the other hand, if the value of LINES is greater than 55, a routine called TOTALER is executed, presumably computing and printing totals on the bottom of the page; a page counter called PAGE is incremented; a page header is printed on top of the next page; and control is returned to the main part of the program via the RETURN statement.

IX. OPERATING ENVIRONMENT

9.1 Job Setup

The deck setup to compile and run a 3300 STRIGOL is as follows:

```
$JOB,...
$STRIGOL,...
    PROGRAM name
      :
      :
    STRIGOL Source Deck
      :
      :
    END
$LOAD,56
$RUN,...
Data (if any)
```

STRIGOL Card

$STRIGOL,L,A,S,P,X,B=ℓ,I=n

L	List source language on standard output (Lun 61).
A	List assembly language on standard output (Lun 61).
S	At end of compilation time, generate a list of all strings defined during compilation. Since any string name used in a STRIGOL program is considered to be defined implicitely, this option can be useful in detecting possible keypunch errors which can result in accidentally defining new strings.
P	Punch relocatable binary deck on standard punch unit (Lun 62).
P=n	Punch relocatable binary deck on logical unit n.
X	Generate load-and-go on standard load-and-go (Lun 56).
X=n	Generate load-and-go on logical unit n.
B=ℓ	Modify maximum execution time input/output buffer length to be ℓ characters in length. If omitted or illegal, the normal buffer length of 4096 characters will be assumed.
I=n	Indicates that the source deck is on logical unit n. If this parameter is omitted, the source deck is inputted from the standard input (Lun 60).

The parameters on the STRIGOL card may be in any order, and any or all parameters may be omitted.

Compilations of STRIGOL programs can not be stacked under a single STRIGOL card.

(Note: FINIS cards are not required and are not allowed in a STRIGOL job.)

9.2 Compiler Pseudo Instructions

Compiler pseudo instructions are commands to the STRIGOL compiler which can be used to format the source program listing. They themselves do not appear in the listing. Compiler pseudo instructions are written like regular STRIGOL statements in columns 10-72 of the source deck. The pseudo instructions and their effects on the source listing are:

(i)	EJECT	Eject to a new page in the listing of the source program.
(ii)	SPACE	Print one blank line in the source program listing.
(iii)	SPACE n	Print n blank lines in the source program listing.

(Note: Three blank lines are automatically printed upon encountering a ROUTINE declaration statement.)

9.3 Compilation Time Messages

During compilation a check is made to determine if certain errors have occurred. All error messages appear immediately following the statement in which the error was detected in the following format:

↑ message *↑*

In addition, at the end of compilation additional information is printed out if error conditions such as undefined or multiply defined statement labels were detected anywhere in the program. Due to the interaction of error conditions, the occurrence of some errors may prevent the detection of others until those which have been detected are corrected.

Fatal Errors

Fatal errors result in an immediate termination of compilation. The fatal error messages are:

(i) ILLEGAL LUN SPECIFIED ON I-OPTION, meaning that a non-numerical logical unit number was detected in the I-OPTION on the STRIGOL card.

(ii) SYSTEM CONTROL CARD READ, JOB ABORTED, indicating that a card containing "$" in column 1 has been encountered.

(iii) EOF CARD READ, JOB ABORTED, meaning that an end-of-file card has been detected during compilation.

Informative Messages

Informative messages indicate either a trivial error in syntax, or an error condition which can be corrected by the STRIGOL compiler. The informative messages are:

(i) ILLEGAL I/O BUFFER SIZE SPECIFIED ON B-OPTION. This condition occurs if the buffer length specification on the B-option on the STRIGOL card is not numeric, or is less than zero. The normal I/O buffer length of 4096 characters is assumed and compilation continues.

(ii) PROGRAM CARD MISSING, ASSUME PROGRAM JOB., meaning that the first statement in the source program was not a PROGRAM statement. The name JOB. is assumed and compilation continues.

(iii) ENDROUTINE CARD MISSING, indicating that the END card was encountered before a routine was terminated by an ENDROUTINE statement. The missing ENDROUTINE card is automatically inserted by the compiler.

(iv) PRECEEDING STATEMENT WILL NOT BE EXECUTED, indicating that a non-labeled statement has been detected following an unconditional transfer statement; there is no way in which the statement can be executed.

Serious Errors

Serious errors will prevent the program from executing and will inhibit the printing of an assembly language listing if requested on the STRIGOL card. The error count which appears at the end of the program source listing is the count of serious errors. The serious errors are:

(i) EMBEDDED ROUTINE CARD ENCOUNTERED, indicates that while compiling a routine, a second ROUTINE card was encountered before the ENDROUTINE card was encountered for the first routine. This usually occurs when nested routine definition is attempted.

(ii) RETURN CARD ENCOUNTERED OUT OF SEQUENCE, indicating that a RETURN card was encountered other than in a routine.

(iii) ENDROUTINE CARD ENCOUNTERED OUT OF SEQUENCE, meaning that an ENDROUTINE card was read before a matching ROUTINE card.

(iv) ERROR IN IF STATEMENT, indicating that the branch field is missing or illegally specified in an IF statement.

(v) SYNTAX ERROR IN ARITHMETIC EXPRESSION, usually meaning that an extra or missing operator has been detected in an arithmetic replacement statement.

(vi) UNBALANCED ARITHMETIC EXPRESSION, indicating that a different number of left and right parenthesis or an illogical ordering of left and right parenthesis has been encountered in an arithmetic replacement statement.

(vii) INVALID READ STATEMENT, meaning that the input string name is missing in a READ statement.

(viii) INVALID WRITE STATEMENT, usually meaning that the logical unit was illegally specified on a WRITE or PRINT statement.

(ix) ILLEGAL LOGICAL UNIT NUMBER, indicating that the logical unit number on an input/output statement is not in the range 1 to 63.

(x) INVALID SCAN STATEMENT, meaning that the "[" was missing in a SCAN statement.

(xi) SCAN REFERENCE MISSING, indicating that no scan reference appears between the word "SCAN" and the "[" in a SCAN statement.

(xii) NO] IN SCAN STATEMENT, meaning that the "]" was missing in a SCAN statement.

(xiii) LITERAL CANNOT BE USED AS A SCAN REFERENCE, indicating that the scan reference is a literal; the scan reference must be a string or a substring.

(xiv) PROGRAM NAME CANNOT APPEAR WITHIN A PROGRAM, indicating the program name is also being used as a statement label, numeric variable, string name, or a routine name.

(xv) name IS AN INVALID NAME, meaning that name is not a valid identifier. This can also result from an illegal syntax construction in a statement.

(xvi) INVALID LITERAL, indicating that a literal has been encountered in a string expression which contains no characters, a quotation mark, or is missing the right delimiting quotation mark.

(xvii) SYNTAX ERROR IN STRING EXPRESSION, meaning that an illegal construction has been encountered.

(xviii) MISSING COMMA IN STRING EXPRESSION, indicating that two elements appear in a string expression with no comma separating them.

(xix) SCAN REFERENCE ALSO USED AS A FILLER, meaning that in a SCAN statement, the scan reference also appears as a filler in the scanning pattern.

(xx) ILLEGAL USE OF A FILLER IN A STRING EXPRESSION, indicating that a filler appears in a string expression other than as an element of the scanning pattern in a SCAN statement.

(xxi) name IS AN UNDECLARED NUMERIC VARIABLE, indicating that name appears contextually as a numeric variable, but has not previously been declared in a NUMBER statement.

(xxii) ILLEGAL SCAN REFERENCE, meaning that the scan reference is an expression rather than a string or substring.

Near the end of compilation, additional messages are printed out as follows:

(i) STRINGS DEFINED DURING COMPILATION. A list of all strings which were implicitly defined during compilation is generated if the S-option was requested on the STRIGOL card.

(ii) THE FOLLOWING STATEMENT LABELS / ROUTINES WERE MULTIPLY-DEFINED / UNDEFINED / UNREFERENCED .

A list of all multiply-defined, undefined, and unreferenced statement labels and routines is printed out. The unreferenced statement labels and routines lists are informative messages only.

(iii) THE FOLLOWING | ROUTINES / NUMBERS / STATEMENT LABELS | WERE ALSO USED AS | NUMBERS / STATEMENT LABELS / STRING .

A list of all identifiers used more than once for a different purpose is printed out. Identifiers must be uniquely defined within a program.

At the end of compilation one of two messages appears:

(i) n ERRORS DURING COMPILATION, EXECUTION SURPRESSED. This indicates that n serious errors were detected during compilation; all further processing of the program is discontinued.

(ii) NO ERRORS DURING COMPILATION, indicating that the source program contained no syntactic errors.

It is possible that errors will be detected during the assembly phase which were not detected during the compilation phase. If this occurs, the following message will be printed:

NUMBER OF LINES WITH DIAGNOSTICS n

Normally this indicates that COMPASS coding was included in the STRIGOL source deck, and that errors occurred in these statements. If this diagnostic occurs and no COMPASS coding was done, consult NMSU Computer Center Systems Programmers.

9.4 Execution Time Messages

The standard format used in printing the majority of execution time diagnostics is the following:

ERROR IN RRRRRRRR CALLED FROM XXXXX, $M_1 \ldots M_n$

where RRRRRRRR is the name of the routine in which the error occurred, XXXXX is the address from which the routine was last called, and $M_1 \ldots M_n$ is the error message.

The execution time error messages and their meaning is:

(i) BAD TAPE ON LUN nn, JOB ABORTED. During an I/O operation a non-recoverable parity error was encountered on a magnetic tape nn.

(ii) COMPARE ERROR ON LUN nn, JOB ABORTED. A card was read on logical unit nn which generated a persistent compare error. This usually means that the card is off-punched.

(iii) BINARY CARD LUN nn, JOB ABORTED. A binary card was read on logical unit nn. Binary I/O is not allowed in 3300 STRIGOL.

(iv) ILLEGAL I/O ON LUN nn, JOB ABORTED. An attempt was made to issue an I/O operation on logical unit nn which would not be meaningful; e.g., attempting to rewind the typewriter.

(v) NEGATIVE STRING LENGTH IN READ STATEMENT, JOB ABORTED. The length specifications was given in a READ statement, but it was less than zero.

(vi) END-OF-JOB CARD ON LUN nn, JOB ABORTED. The system end-of-job card was read on logical unit nn. The job is aborted immediately upon encountering the end-of-job card.

(vii) ERASE ERROR ON LUN nn, JOB ABORTED. During an attempt to recover from a parity error on an output operation on logical unit nn, a parity error occurred during an erase. No further recovery is possible if this occurs.

(viii) NEGATIVE SUBSTRING LENGTH. A substring variable with a negative substring specification has been detected. If this occurs in a NUMERIC test in an IF statement, the string in question will not be numeric. If the left part of a string replacement statement is a negative substring, the replacement statement will be ignored.

(ix) NEGATIVE SUBSTRING IN REPLACEMENT EXPRESSION. This indicates that a negative substring has been detected in a string expression. It is treated as a null string.

(x) STATIC STRING LENGTH OVERFLOW. This error message is generated if the length of the string expression in a WRITE or PRINT statement exceeds the maximum I/O buffer length. Trailing characters are discarded.

(xi) TRUNCATION ERROR XXXXXXXX. In a call to the STRING function, insufficient string length was allowed to generate the complete value of the number. XXXXXXXX is the value of the number which could not be converted.

(xii) STRING NOT NUMERIC. In a call to the NUMBER function, the string could not be converted to a number because it contained other than numeric digits. The value returned is zero.

(xiii) NULL STRING. An attempt was made to convert a null string to a number via the NUMBER function. The value returned is zero.

(xiv) NEGATIVE FILLER LENGTH. A negative length was specified in a fixed-length filler. The filler is nulled and ignored.

(xv) SCAN TABLE OVERFLOW, JOB ABORTED. More than 20 elements occurred in the scanning pattern. This is an implementation restriction which can be altered.

(xvi) NEGATIVE SCAN REFERENCE LENGTH, JOB ABORTED. The scan reference is a negative substring.

(xvii) NEGATIVE PATTERN ELEMENT LENGTH. A substring in the scanning pattern has a negative length specification. The substring is treated as a null string.

Appendix A

Sample Programs

```
          PROGRAM LONGSEG
*    THIS PROGRAM FINDS THE LONGEST COMMON SEGMENT OF TWO LISTS.
          NUMBER N
READ      READ LISTS                                          /,DONE
DEBLANK   SCAN LISTS [≠ ≠]=                                   /DEBLANK
          SCAN LISTS [(A),≠,≠,(B)]
          WRITE ≠-≠,LISTS
          EXECUTE LCOMSEG
          WRITE ≠ ≠,LONG
          GO TO READ


          ROUTINE LCOMSEG
*    ROUTINE LCOMSEG FINDS THE LONGEST COMMON SEGMENT OF THE TWO LISTS,
*    A AND B.  THE ROUTINE RETURNS THE LONGEST SEGMENT UNDER THE NAME
*    LONG.  THE ROUTINE FINDS THE LONGEST MATCHING SEGMENT STARTING WITH
*    THE FIRST CHARACTER; THEN STARTING WITH THE SECOND CHARACTER; ETC.
          LONG=
LOOP      N=1
          SAVE=
LOOP1     SCAN B [A$N]                                        /,DELA
          SAVE=A$N
          N=N+1
          IF (N≤LENGTH(A))                                    /LOOP1
DELA      IF (LENGTH(SAVE)≤LENGTH(LONG))                      /DELB
          LONG=SAVE
DELB      SCAN A [(F$1)]=                                     /,RET
*    IF A IS SHORTER THAN THE LONGEST SEGMENT SO FAR, RETURN.
          IF (LENGTH(A)>LENGTH(LONG))                         /LOOP
RET       RETURN
          ENDROUTINE
DONE      END

NO ERRORS DURING COMPILATION
```

```
ABCDE,DEABC
ABC

ABCDEFGHA,BCBCDAEFGHXYZ
EFGH
```

Example 1

Sample Output

```
          PROGRAM POLISH
*    THIS PROGRAM DOES A TRANSLATION OF A STATEMENT IN NORMAL INFIX
*    NOTATION TO REVERSE POLISH NOTATION.  UNARY OPERATORS ARE NOT
*    ALLOWED AND ALL OPERANDS MUST BE ONE CHARACTER IN LENGTH.
*

*    TABLE IS THE HIERARCHY TABLE
          TABLE=≠^=0^(0^+1^-1^*2^/2^)3≠
READ      READ CARD                                              /,EXIT
DEBLANK   SCAN CARD [≠ ≠]=                                       /DEBLANK
          PRINT ≠-≠,CARD
*    EXTRACT A CHARACTER
LOOP      SCAN CARD [(CHAR$1)]=                                  /,PRINT
*    CHECK TO SEE IF THE CHARACTER IS AN OPERATOR
          SCAN TABLE [≠^≠,CHAR,(N1$1)]                           /OP
*    IF IT IS AN OPERAND, ADD TO TARGET STRING
          TSTRING=TSTRING,CHAR
          GO TO LOOP
OP        SCAN CHAR [≠(≠]                                        /LOOP2
          SCAN CHAR [≠)≠]                                        /,CHECK
*    ON A ), EXTRACT EVERYTHING ON STACK TO PREVIOUS (
          BACKSCAN STACK [(FILL),≠(≠]=
          TSTRING=TSTRING,FILL
          GO TO LOOP
CHECK     IF (NULL(STACK))                                       /LOOP2
          BACKSCAN STACK [(CHAR1$1)]=
          SCAN TABLE [≠^≠,CHAR1,(N2$1)]
          IF (NUMBER(N1)≤NUMBER(N2))                             /MORE
          STACK=STACK,CHAR1,CHAR
          GO TO LOOP
*    LEFT PARENTHESES GO ONTO THE OPERATOR STACK
MORE      TSTRING=TSTRING,CHAR1
          GO TO CHECK
LOOP2     STACK=STACK,CHAR
          GO TO LOOP
PRINT     BACKSCAN STACK [(FILL)]=
*    EMPTY OPERATOR STACK
          TSTRING=TSTRING,FILL
          WRITE ≠0≠,TSTRING
*     NULL TARGET STRING FOR NEXT DATA CARD
          TSTRING=
          GO TO READ
EXIT      EXIT
          END

NO ERRORS DURING COMPILATION
```

Example 2

```
A=B+C

ABC+=

A=B*C+D

ABC*D+=

A=B+C-D

ABC+D-=

A=B*(C+D*(A/(R-B)))

ABCDARB-/***=
```

Example 2

Sample Output

```
          PROGRAM EDITOR
*    THIS PROGRAM READS IN TEXT AND PRINTS N CHARACTERS PER LINE, EXTRA
*    BLANKS ARE INSERTED BETWEEN WORDS TO FILL OUT LINES.  THE INPUT TEXT
*    IS ASSUMED TO BE FREE FORMATTED.
          NUMBER N,L
*    READ IN N AND CONVERT IT TO A NUMBER
          READ NN$2
          N=NUMBER(NN)+1
          L=NUMBER(NN)
          PRINT ≠ INPUT TEXT....≠
*    READ IN THE INPUT TEXT
READ      READ INPUT                                              /,EDIT
          TEXT=TEXT,INPUT,≠ ≠
          PRINT ≠       ≠,INPUT
          GO TO READ
EDIT      PRINT ≠-TEXT EDITED TO ≠,STRING(L),≠ CHARACTERS PER LINE,...,≠
*    DELETE EXCESS BLANKS FROM TEXT
DEBLANK   SCAN TEXT [≠  ≠]=≠ ≠                                    /DEBLANK
*    REMOVE POSSIBLE LEADING BLANK
          SCAN TEXT$1 [≠ ≠]=
MORE      SCAN TEXT [(WORD),≠ ≠]=                                 /,LASTLINE
          LINE=LINE,WORD,≠ ≠
          IF (LENGTH(LINE)=N)                                     /PRINT
          IF (LENGTH(LINE)<N)                                     /MORE
          L=LENGTH(LINE)-LENGTH(WORD)-1
          LINE=LINE$L
          TEXT$0=WORD,≠ ≠
*    INSERT BLANKS TO FILL OUT LINE
          EXECUTE INSERT
PRINT     PRINT ≠       ≠,LINE
          LINE=
          GO TOMORE
LASTLINE  PRINT ≠       ≠,LINE



          ROUTINE INSERT
*    THIS ROUTINE INSERTS BLANKS INTO LINE UNTIL THE LENGTH OF LINE IS
*    EQUAL TO NN CHARACTERS
          BACKSCAN LINE$1 [≠ ≠]=
          L=NUMBER(NN)-LENGTH(LINE)
          BLANK=
          LINEX=
LOOP      LINEX=
          BLANK=BLANK,≠ ≠
LOOP1     SCAN LINE [(WORD),BLANK]=                               /,MOREX
          LINEX=LINEX,WORD,BLANK,≠ ≠
          L=L-1
          IF (L≠0)                                                /LOOP1
          LINE=LINEX,LINE
          RETURN
MOREX     LINE=LINEX,LINE
          GO TO LOOP
          ENDROUTINE
          END

NO ERRORS DURING COMPILATION
```

Example 3

```
INPUT TEXT....
      SOME OF THE CHARACTERISTICS OF A PREDICTIVE ANALYZER, A SYSTEM OF
    SYNTACTIC ANALYSIS NOW OPERATIONAL ON AN IBM 7094, ARE DELINEATED.  THE
    ADVANTAGES AND DISADVANTAGES OF THE SYSTEM ARE DISCUSSED IN COMPARISON TO
    THOSE OF AN IMMEDIATE CONSTITUENT ANALYZER, DEVELOPED AT THE RAND CORPORATION
    WITH ROBINSONS ENGLISH GRAMMAR.  IN ADDITION, A NEW TECHNIQUE IS DESCRIBED
    FOR REPETITIVE PATH ELIMINATION FOR A PREDICTIVE ANALYZER, WHICH CAN NOW CLAI
    EFFICIENCY BOTH IN PROCESSING TIME AND CORE STORAGE REQUIREMENT.


TEXT EDITED TO 65 CHARACTERS PER LINE....
    SOME OF THE CHARACTERISTICS OF A PREDICTIVE ANALYZER, A SYSTEM OF
    SYNTACTIC  ANALYSIS  NOW  OPERATIONAL  ON  AN  IBM  7094,  ARE
    DELINEATED.  THE  ADVANTAGES  AND DISADVANTAGES OF THE SYSTEM ARE
    DISCUSSED  IN  COMPARISON  TO  THOSE  OF AN IMMEDIATE CONSTITUENT
    ANALYZER,  DEVELOPED  AT  THE  RAND  CORPORATION  WITH  ROBINSONS
    ENGLISH  GRAMMAR.  IN  ADDITION, A NEW TECHNIQUE IS DESCRIBED FOR
    REPETITIVE  PATH  ELIMINATION FOR A PREDICTIVE ANALYZER, WHICH CAN
    NOW  CLAIM  EFFICIENCY  BOTH  IN PROCESSING TIME AND CORE STORAGE
    REQUIREMENT.
```

Example 3

Sample Output

Appendix B

Internal Specifications

Data Structure

All strings are stored internal to a STRIGOL program as a threaded list, headed by a three-word string specifier. The name of a string is the address of the specifier rather than of the data string, since it is likely that a data string will be completely moved from where it was originally stored during the course of program execution. Each character in a data string is stored in the first character of a separate word in memory with the lower 15-bits in the word containing the address of the next word of the string. The string specifier contains the address of the first word and last word on the list and the current length. If the string is null, all three words in the specifier are zero. The following example illustrates this data structure.

If the address of the specifier for a string called XYZ is 77352 then after executing the STRIGOL statement

```
XYZ = "ABC123"
```

the string XYZ might be organized in core as follows:

77352	52161	52161	A	60000
77353	14100			
77354	6	60000	B	25121
		25121	C	77331
		77331	1	50147
		50147	2	14100
		14100	3	0

The pointer in the last word of a string always has a value of zero. It should be fairly obvious from this example that it is not likely that characters in a string will be stored in consecutive core locations.

There is one exception to the universal use of this type of data structure. Literals can never be changed, so, in order to economize on the amount of core usage, literals are stored in a completely different manner. The characters in a literal are stored as a continguous string of characters, and like the normal data strings, literals are also headed by a three word specifier, but in a somewhat different format. A literal specifier contains the first character address, the last character address and the length of the literal. Since the length of a literal is always equal to the LCA-FCA+1, it may seem redundant to carry along the length; however, as it turns out, this type of data storage is also used in the input-output routines, and these routines require both types of information for reasons which will be explained later. Using the literal from the preceding example, if the specifier is at location 77756, the literal could be represented as follows:

77756	377123	
77757	377130	A B C 1 2 3
77760	6	

Most of the processing in a STRIGOL program is done by executing machine language subroutines and passing a parameter list to the subroutine to allow the appropriate data manipulation. In most of these routines, part or all of the parameter list corresponds to a string expression. As it turns out, the format of the parameter list for a string expression is independent of the subroutine using the expression; in other words, if the same string expression is used in a SCAN statement, a PRINT statement, and a replacement statement, the parameter list corresponding to the string expression would be identical in each case.

For each element in a string expression, there corresponds a one, two, or three word element in the parameter list, depending on the type of string element. In the case of a simple variable, a one word parameter element is generated by the STRIGOL compiler. For example, to the following portion of a string expression

```
...,XX,...
```

would correspond the following COMPASS coding in the assembly listing.

```
LST         XX
```

The opcode LST is a special indicator to the STRIGOL subroutines that the address portion of the machine word contains the address of a simple string variable.

A substring variable generates a two word parameter element. If the substring variable has a constant length field, such as in the following example:

```
...,A$5,...
   LSTM  A
         5
```

The substring variable

```
B$N
```

would result in the parameter element

```
LSTA        B
            N
```

A literal is somewhat of a special case, since it has no name. To solve this problem, a unique name of the form

```
L.nnn
```

is created to identify the literal, where nnn is a unique three digit number for each literal used in a program. In the previous example, the operator field began with the letters "LST"; this is a reflection of the fact that a data string is stored as a list.

Similarly, since a literal is stored as a linear, contiguous array of characters, the operator field for the parameter begins with "LIN". Thus the literal

```
...,"12345",...
```

could result in the parameter element

```
LIN       L.013
```

which would indicate (if true) that this was the 13th literal declared in the program.

A string element which is a filler differs in its parameter representation only in that it is preceded by a word with an operator of "FILL" and no address field. For example the string element

```
...,(ABC$3),...
```

would translate to

```
FILL
LSTM      ABC
          3
```

as another example, consider the following string expression:

```
XX$3,"ABC",(F),"DEF",A,B$N,(F1$3)
```

If the two literals were the first declared in the program then the entire expression would be translated to

```
LSTM              XX
                  3
LIN               L.001
FILL
LST               F
LIN               L.002
LST               A
LSTA              B
                  N
FILL
LSTM              F1
                  3
```

by the STRIGOL compiler. If the literals were not the first declared in the program, the parameter list would differ only in the numbers of the literals.

It was mentioned previously that linear data storage was also used in conjunction with input-output operations. In particular, since it is, of course, not possible to directly write from a data string or read into a list structured data string, it is necessary to reserve a block of core to be used as an I/O buffer. In order to make the size of this I/O buffer as flexible as possible, the area for the buffer must be set up in the main program. In addition, the name of the specifier must be IOAREA. and it must be declared as an entry point. The first word of the specifier contains the character address of the first character in the I/O buffer, the second word should be zero, and the third word should be zero, and the third word is the number of characters available in the buffer. The following example will set up an I/O buffer 1000 characters long:

```
                ENTRY           IOAREA.
IOAREA.         00,C            IOBUFF
                OCT             0
                DEC             1000
IOBUFF          BSS,C           1000
```

STRIGOL Subroutine Calling Sequences

To simplify the COMPASS expansion of a string expression, the following conventions have been adopted:

> If se is a string expression, then [se] is its COMPASS expansion according to rules stated previously. Similarly, if str is a string or substring then [str] is its expansion. For example, if:
>
> se = A, (B$3), X$N
>
> then
>
> ```
> [se] = LST     A
>        FILL
>        LSTM    B
>                3
>        LSTA    X
>                N
> ```

For each STRIGOL subroutine, its general form in STRIGOL is given along with the generalized COMPASS expansion. A few specific examples are also given. All STRIGOL subroutines used from a COMPASS program must be declared as externam symbols.

STRIGOL	COMPASS
READ (lun) se /s,f	ENA lun RTJ READ [se] UJP s UJP f
READ X /GOOD,BAD	ENA 60 RTJ READ. LST X UJP GOOD UJP BAD

```
READ(LUNXX) INPUT$50     /,EOF      LDA     LUNXX
                                    RTJ     READ.
                                    LSTM    INPUT
                                            50
                                    UJP     *+2
                                    UJP      EOF

WRITE (lun)  se                     ENA     lun
    or                              RTJ     WRITE
PRINT (lun)  se                     [se]

WRITE "PAGE", PAGENO                ENA     60
                                    RTJ     WRITE.
                                    LIN     L.001
                                    LST     PAGENO

PRINT (10) DATA, VALUE$X            ENA     10
                                    RTJ     WRITE.
                                    LST     DATA
                                    LSTA    VALUE
                                            X

ENDFILE  lun                        ENA     lun
                                    RTJ     ENDFILE

ENDFILE OUTFILE                     LDA     OUTFILE
                                    RTJ     ENDFILE.

REWIND  lun                         ENA     lun
                                    RTJ     REWIND.

UNLOAD  lun                         ENA     lun
                                    RTJ     UNLOAD.

str=se                              RTJ     CONCAT.
                                    [str]
                                    [se]

A=B,C                               RTJ     CONCAT.
                                    LST     A
                                    LST     B
                                    LST     C

A$3=                                RTJ     CONCAT.
                                    LSTM    A
                                            3
```

ARRAY=X$3,"ABC",ARRAY$N		RTJ	CONCAT
		LST	ARRAY
		LSTM	X
		LST	3
		LIN	L.001
		LSTA	ARRAY
			N
EXECUTE name		RTJ	name
EXECUTE TRANSLAT		RTJ	TRANSLAT
ROUTINE name	name	UJP	**
ROUTINE TRANSLAT	TRANSLAT	UJP	**
RETURN (in routine name)		UJP,I	name
RETURN (in routine TRANSLAT)		UJP,I	TRANSLAT
GO TO name		UJP	name
GO TO LOOP		UJP	LOOP
EXIT (in program name)		UJP,I	name
ENDROUTINE (in routine name)		UJP,I	name
ENDROUTINE (in routine TRANSLAT)		UJP,I	TRANSLAT
str=STRING(numb)		ENQ	0
		LDA	numb
		RTJ	STRING.
		[str]	
X$3=STRING (count)		ENQ	0
		LDA	COUNT
		RTJ	STRING.
		LSTM	X
			3
str=STRING(size, numb)		ENQ,S	size
		LDA	numb
		RTJ	STRING.
		[str]	

ABC$X=STRING(M,N)	LDQ	M
	LDA	N
	RTJ	STRING.
	LSTA	ABC
		X
numb=NUMBER(str)	ENA	numb
	RTJ	NUMBER.
	[str]	
IF(NULL(str)) /y,n	LDA	name+2
	AZJ,EQ	y
	UJP	N
IF(NULL(FLAG)) /,OK	LDA	FLAG+2
	AZJ,EQ	*+2
	UJP	OK
IF (NUMERIC(str)) /y,n	RTJ	NUMERIC.
	[str]	
	AZJ,EQ	y
	UJP	n
IF(NUMERIC(CARD$N)) /LOOP	RTJ	NUMERIC.
	LSTA	CARD
		N
	AZJ,EQ	LOOP
IF (NUMBER(X$3)≥n) /BUMP, FLAG	ENA	T.001
	RTJ	NUMBER.
	LSTM	3
		X
	LDA	T.001
	LDQ	N
	AQJ,GE	BUMP
	UJP	FLAG
SCAN str [se] /s,f (BACKSCAN)	RTJ	SCAN.(BSCAN.)
	[str]	
	[se]	
	UJP	s
	UJP	f
SCAN str [se_1]=se_2 /s,f	RTJ	SCAN.(BSCAN.)
	[str]	
	[se_1]	
	OCT	-0
	[se_2]	
	UJP	s
	UJP	f

```
SCAN NAME[DATA]      /GOOD              RTJ       SCAN.
                                        LST       NAME
                                        LST       DATA
                                        UJP       GOOD

BACKSCAN CARD&3 [BLANK]=   /,REPEAT     RTJ       BSCAN.
                                        LSTM      CARD
                                                  3
                                        LST       BLANK
                                        OCT       -0
                                        UJP       *+2
                                        UJP       REPEAT

SCAN STR[(F$N),"x"]="x"                 RTJ       SCAN.
                                        LST       STR
                                        FILL
                                        LSTA      F
                                                  N
                                        LIN       L.001
                                        OCT       -0
                                        LIN       L.001
                                        UJP       *+1

BACKSCAN A$1 [A$1]=B$1,A   /WOOPS,C     RTJ       BSCAN.
                                        LSTM      A
                                                  1
                                        LSTM      A
                                                  1
                                        OCT       -0
                                        LSTM      B
                                                  1
                                        LST       A
                                        UJP       WOOPS
                                        UJP       C
```

Appendix C

The STRIGOL Scan