Georgia Institute of Technology SCHOOL OF INFORMATION AND COMPUTER SCIENCE / (404) 894-3152 / ATLANTA, GEORGIA 30332

G T L

PROGRAMMERS REFERENCE MANUAL

FOR THE

BURROUGHS B 5500

August 1974

G T L

PROGRAMMERS REFERENCE MANUAL

FOR THE

BURROUGHS B 5500

December 1971

ACKNOWLEDGMENTS

The GTL compiler almost certainly would not exist today if it were not for the dedicated effort and genius of a single person, Martin Alexander. His phenomenal talent was first recognized in the early 1960's when he both designed and coded a LISP interpreter in machine language (not assembly) for the Burroughs 220 computer in one weekend! Furthermore, he was embarrasssed to admit that as many as about 5 or 10 instructions had to be changed before it worked correctly! This success was followed by a LISP interpreter for the Burroughs B 5500, this time written in ALGOL, and now---GTL.

Although Martin Alexander, who most unfortunately is no longer employed at the Georgia Tech Computer Center, must be acknowledged as the sole creator of GTL, it is felt that he would insist on giving credit to the many people who have provided suggestions, advice, criticism, and assistance in the effort. Particular credit is due Marie Courtney, who implemented major portions of the double precision and complex arithmetic, checked and corrected most of the machine language intrinsic functions, and assisted in the design and implementation of many other features.

Credit is certainly due the Burroughs Corporation, whose excellent Extended ALGOL compiler provided the starting point for GTL.

This manual was prepared by the staff of the Rich Electronic Computer Center. The GTL Compiler is currently supported by the School of Information and Computer Science.

TABLE OF CONTENTS

I. INTRODUCTION 1-1

II. MISCELLANEOUS EXTENSIONS OF ALGOL 2-1
2.1 SINGLE PRECISION STANDARD FUNCTIONS 2-1
2.2 CASE EXPRESSIONS 2-1
2.3 FOR STATEMENT 2-2
2.4 EXIT STATEMENT 2-2
2.5 RETURN STATEMENT 2-2
2.6 ERROR STATEMENT 2-3
2.7 MATRIX MANIPULATION 2-3
2.8 POWERS OF TEN TABLE 2-5
2.9 SWAP STATEMENT 2-5
2.10 RANDOM NUMBER GENERATOR 2-6
2.11 STATEMENT LINE NUMBER DETERMINATION 2-7

III. DOUBLE PRECISION ARITHMETIC 3-1
3.1 INTRODUCTION 3-1
3.2 FORM FOR DOUBLE EXPRESSIONS 3-1
3.3 DOUBLE ARITHMETIC OPERATORS 3-2
3.4 DOUBLE RELATIONAL OPERATORS 3-2
3.5 DOUBLE STANDARD FUNCTIONS 3-2
3.6 RULES OF CONTEXT 3-3
3.7 DOUBLE PRECISION INPUT-OUTPUT 3-4
3.8 RESTRICTIONS 3-4
3.9 EXAMPLE PROGRAM 3-4

IV. COMPLEX ARITHMETIC 4-1
4.1 INTRODUCTION 4-1
4.2 FORM FOR COMPLEX EXPRESSIONS 4-1
4.3 COMPLEX ARITHMETIC OPERATORS 4-2
4.4 COMPLEX RELATIONAL OPERATORS 4-3
4.5 COMPLEX STANDARD FUNCTIONS 4-3
4.6 COMPLEX INPUT-OUTPUT 4-4
4.7 DOUBLE COMPLEX DECLARATOR 4-4
4.8 RESTRICTIONS 4-5
4.9 EXAMPLE PROGRAM 4-5

V. STRING PROCESSING 5-1
5.1 STRING VARIABLES 5-1
5.1.1 Simple String Variables and Arrays 5-1
5.1.2 Substring Variables 5-2
5.1.3 Formal String Variables 5-4
5.2 STRING DESIGNATOR 5-5
5.3 STRING EXPRESSIONS 5-6
5.3.1 String Expression Forms 5-6
5.3.2 The Quoted String 5-7
5.3.3 String Designator 5-7
5.3.4 String Assignment Statement 5-7
5.3.5 String Function Designator 5-8
5.3.6 SPACE Function 5-8

TABLE OF CONTENTS (Cont.)

5.3.7 The NIL Function 5-9
5.3.8 The String Skip Indicator 5-10
5.3.9 The QMARK Function 5-11
5.3.10 The Bit Expression 5-11
5.3.11 The Restricted Boolean Expression 5-11
5.3.12 The Restricted Arithmetic Expression 5-12
5.3.13 The Restricted Symbol Expression 5-12
5.3.14 The STRING Transfer Function 5-13
5.3.15 The SUBST Function 5-14
5.3.16 The FILL Function 5-15
5.3.17 The OCTAL Function 5-16
5.3.18 The String Repeat Expression 5-17
5.3.19 Parenthesized String Expression 5-17
5.4 THE STRING ASSIGNMENT STATEMENT 5-18
5.4.1 The Basic String Assignment Statement 5-18
5.4.2 String Assignment with SPACE 5-18
5.4.3 String Assignment with NIL 5-19
5.4.4 String Assignment with String Skip Indicator 5-19
5.4.5 String Assignment Overlap: A Warning 5-20
5.4.6 String Assignment Statement Containing String Length Assignment 5-21
5.4.7 The String FILL Statement 5-22
5.4.8 The String Addition Assignment Statement 5-22
5.4.9 The String Subtraction Assignment Statement 5-23
5.5 STRING COMPARISON 5-23
5.5.1 String Relational Expression 5-23
5.5.2 String Relation with SPACE 5-24
5.5.3 String Relation with NIL 5-25
5.5.4 String Relation with String Skip Indicator 5-25
5.5.5 String Pattern Matching 5-26
5.5.6 The SEARCH Function 5-27
5.6 BIT EXPRESSIONS 5-29
5.6.1 Bit Expression Form 5-29
5.6.2 Bit Primary 5-29
5.7 STRING ACTUAL PARAMETERS 5-30
5.7.1 Call-by-Value 5-30
5.7.2 Call-by-Name 5-30
5.8 USING STRINGS IN OTHER TYPES OF EXPRESSIONS 5-31
5.8.1 Arithmetic Expressions 5-31
5.8.2 Symbolic Expressions 5-33
5.9 USING AN ARRAY OR A STRING VARIABLE 5-33
5.10 OPTIMALITY OF STRING EXPRESSIONS 5-34
5.11 READING AND WRITING STRINGS 5-34
5.11.1 READ and WRITE Statements 5-34
5.11.2 GTL Input-Output Functions 5-34

VI. LISP 2 6-1
6.1 INTRODUCTION 6-1
6.2 S-EXPRESSION AND LISP RECORDS 6-1
6.2.1 Record and Field Designator 6-1
6.2.2 LISP Records 6-2
6.2.3 LISP Lists 6-5

TABLE OF CONTENTS (Cont.)

6.3 SYMBOL EXPRESSION 6-6
6.3.1 Definition 6-6
6.3.2 Quoted S-expressions 6-7
6.3.3 Numbers and Arithmetic Expressions 6-7
6.3.4 LISP Variables 6-7
6.3.5 Assignment Statements 6-8
6.3.6 The Field Designators 6-8
6.3.7 Conditional Expressions 6-9
6.3.8 LISP Function Designator 6-9
6.4 LISP STANDARD FUNCTIONS 6-10
6.4.1 CONS 6-10
6.4.2 LIST 6-10
6.4.3 RANDOM 6-10
6.4.4 APPEND 6-11
6.4.5 NCONC 6-11
6.4.6 SPACE and QMARK 6-12
6.5 BOOLEAN STANDARD FUNCTIONS 6-12
6.5.1 ATOM 6-12
6.5.2 ATSYM 6-12
6.5.3 NUMBERP 6-13
6.5.4 ALF 6-13
6.5.5 NULL 6-13
6.5.6 MEMBER 6-14
6.6 LISP RELATIONAL EXPRESSIONS 6-14
6.7 THE LISP ASSIGNMENT STATEMENT 6-15
6.8 THE LISP ITERATIVE STATEMENT 6-17
6.8.1 The ON Statement 6-17
6.8.2 The IN Statement 6-18
6.8.3 The WHILE Part 6-18
6.9 EXTENSIONS OF ARITHMETIC EXPRESSIONS 6-19
6.9.1 Arithmetic Expression Syntax Extension 6-19
6.9.2 The LENGTH Function 6-19
6.10 READING AND WRITING S-EXPRESSIONS 6-20
6.10.1 Output Functions 6-20
6.10.2 Input Functions 6-22
6.11 THE SYMBOL MONITOR 6-24
6.12 ATOMIC SYMBOLS 6-26
6.12.1 Types of Atomic Symbols 6-26
6.12.2 Nonstandard Atomic Symbols 6-27
6.12.3 Uniqueness of Atomic Symbols 6-28
6.13 THE LISP OBJECT LIST 6-29
6.13.1 The LISP Symbol Table 6-29
6.13.2 The OBLIST Function 6-30
6.13.3 The REMOB Statement 6-31
6.14 STRINGS AND ATOMIC SYMBOLS 6-31
6.14.1 Creation of Atomic Symbols 6-31
6.14.2 The ATCON Function 6-32
6.14.3 The MKATOM Function 6-32
6.14.4 The GENSYM Function 6-33
6.15 LISP REFERENCE VALUE TRANSFER FUNCTIONS 6-34
6.15.1 The CTSM Function 6-34
6.15.2 The SMTA Function 6-34

TABLE OF CONTENTS (Cont.)

6.15.3 The ATSM Function 6-35
6.16 THE CTR FIELD 6-36
6.17 PREFIX AND DOT OPERATORS 6-36
6.17.1 Prefix Field Designators 6-36
6.17.2 Boolean Prefix Operators 6-37
6.17.3 The Dot Operator 6-37
6.18 PROPERTY LIST OPERATORS 6-39
6.18.1 The Property List 6-39
6.18.2 ADDPROP 6-39
6.18.3 PROP 6-40
6.18.4 REMPROP 6-41
6.18.5 The Numeric Property Record 6-42
6.18.6 Reference Property Records 6-42
6.19 THE SYMBOL DEFINE DECLARATION 6-43
6.19.1 The Standard Declaration 6-43
6.19.2 CDR Field Initialization 6-44
6.19.3 The Asterisk Form 6-46
6.20 STORAGE RECLAMATION 6-47
6.20.1 Automatic Versus Programmed Storage Reclamation 6-47
6.20.2 Automatic Storage Reclamation 6-48
6.20.3 Programmed Storage Reclamation 6-49
6.21 AUTOMATIC STORAGE AND RETRIEVAL OF LISP LIST STRUCTURE . . . 6-49
6.21.1 The LISP "Memory" 6-49
6.21.2 The REMEMBER Statement 6-50
6.21.3 The RECALL Statement 6-51
6.22 THE INTERNAL REPRESENTATION OF LISP RECORDS 6-52
6.22.1 LISP Reference Values 6-52
6.22.2 Atomic Symbols 6-53
6.22.3 Atomic Number 6-54
6.22.4 Dotted Pairs 6-54
6.22.5 Other Types of Records 6-54
6.23 LISP SYSTEM CONTROL PARAMETERS 6-54
6.24 PROGRAMMED STORAGE RECLAMATION 6-56
6.25 LISP EXAMPLE PROGRAM 6-58

VII. RECORD PROCESSING 7-1
7.1 INTRODUCTION 7-1
7.2 BASIC CONCEPTS OF GTL RECORD PROCESSING 7-2
7.2.1 Reference Expressions 7-2
7.2.2 Field Designators 7-4
7.2.3 The Reference Assignment Statement 7-5
7.2.4 The Field Declaration 7-5
7.2.5 Indexed Fields 7-7
7.3 THE DISK SYSTEM 7-7
7.3.1 The Record Class Declaration 7-7
7.3.2 The RECORD File Declaration 7-9
7.3.3 The Record Designator 7-10
7.3.4 Record Relational Expressions 7-11
7.3.5 Transfer Functions 7-12
7.3.6 Storage Reclamation 7-12
7.3.7 Saving and Restoring Heads of Master Lists in Non-LOCAL Files 7-13

7.3.8 Printing Reference Values 7-15
7.4 THE CORE STORAGE PLEX PRECESSING SYSTEM 7-15
7.4.1 The Record Class Identifier 7-15
7.4.2 Field Designators 7-15
7.4.3 Record Designator 7-15
7.4.4 The SYMBOL PLEX Option 7-17
7.4.5 The ATSM Transfer Function 7-18
7.4.6 The RECALL and REMEMBER Statement 7-19
7.5 RECOMMENDED PRACTICES 7-19
7.6 EXAMPLE PROGRAM 7-20

VIII. SYNTAX-DIRECTED PARSING 8-1
8.1 INTRODUCTION 8-1
8.2 SYMBOL FORMAT EXPRESSIONS 8-9
8.2.1 Terminal Symbols 8-9
8.2.2 Nonterminal Symbols 8-12
8.2.3 NIL 8-12
8.2.4 Statements 8-13
8.2.5 Labels 8-13
8.2.6 RETURN 8-14
8.2.7 The SWITCH Option 8-14
8.2.8 The Error Message Option 8-15
8.2.9 Syntax and Semantics of SYMBOL FORMAT Expressions 8-16
8.3 SYMBOL FORMAT DECLARATIONS 8-18
8.4 SYMBOL FORMAT STATEMENTS , ... 8-19
8.5 SYMBOL FORMAT AUXILIARY DECLARATIONS 8-20
8.5.1 Syntactic Class Declaration 8-20
8.5.2 Class Variable Declaration 8-24
8.5.3 Getnext Procedure Declaration 8-24
8.5.4 Error Procedure 8-25
8.5.5 The Trace Option 8-25
8.6 RECOMMENDED PRACTICES 8-26
8.7 BOOLEAN PROCEDURE EQUIVALENT OF SYMBOL FORMAT DECLARATION . . 8-27
8.8 EXAMPLE PROGRAM 8-29

IX. GTL INPUT-OUTPUT FUNCTIONS 9-1
9.1 INTRODUCTION 9-1
9.2 THE OUTPUT FUNCTIONS 9-1
9.2.1 Extended WRITE Statement 9-1
9.2.2 The PRINT, PRIN, and TERPRI Statements 9-1
9.2.3 The FORMAT Option 9-3
9.2.4 Literal String 9-4
9.2.5 String Values 9-5
9.2.6 Real and Integer Values 9-5
9.2.7 Alpha Values 9-5
9.2.8 Boolean Values 9-5
9.2.9 Double Precision Values 9-5
9.2.10 Complex and Double Precision Complex Values 9-6
9.2.11 LISP Values 9-6
9.2.12 Reference Values 9-6
9.2.13 QMARK 9-6
9.2.14 SPACE 9-7

TABLE OF CONTENTS (Cont.)

9.2.15 SKIP 9-7
9.2.16 The NTS Statement 9-8
9.2.17 Conditional PRINT Statement 9-9
9.3 THE OUTPUT STATEMENT 9-10
9.3.1 The Standard Form 9-10
9.3.2 The Output Procedure 9-11
9.3.3 Setting Left and Right Margins 9-12
9.4 THE READ FUNCTIONS 9-14
9.4.1 Extended READ Statement 9-14
9.4.2 The GTL Read Mechanism 9-14
9.4.3 The SCAN Function 9-15
9.4.4 The READCON Function 9-16
9.4.5 The READN Function 9-17
9.4.6 The READ1 Function 9-17
9.4.7 The READ Function 9-17
9.5 THE INPUT STATEMENT 9-18
9.5.1 The Standard Form 9-18
9.5.2 The Input Procedure 9-19
9.5.3 Setting Left and Right Margins 9-20
9.5.4 Sign-Number Separation 9-21
9.6 REMOTE TERMINAL INPUT-OUTPUT 9-23
9.6.1 The FILE REMOTE Declaration 9-23
9.6.2 FILE REMOTE Side-Effects 9-26
9.6.3 READ and WRITE Statements 9-27
9.6.4 READ TWX 9-28
9.6.5 WRITE TWX 9-28
9.6.6 READN (TWX) 9-29
9.6.7 READN (TWXA) 9-30
9.6.8 TWXNUM 9-30
9.6.9 Conversational READ Statement 9-30
9.7 STANDARD VARIABLES AND SYSTEM CONTROL PARAMETERS 9-32
9.7.1 The Standard Variables 9-32
9.7.2 The Standard Variable TAB 9-33
9.7.3 The Standard Variable COL 9-34
9.7.4 System Control Parameters 9-35
9.8 SAMPLE INPUT AND OUTPUT STATEMENTS 9-37
9.8.1 Card Reader 9-37
9.8.2 Line Printer 9-38
9.8.3 Remote Terminal Files 9-38
9.8.4 Listing of Input Cards 9-39

APPENDIX A - EXAMPLES OF GTL PROGRAMS A-1
String Processing Example A-2
Lisp Processing Example A-3
Lisp Processing Example A-4
Syntax-Directed Parsing Example A-7
APPENDIX B - REMOTE TERMINAL CHARACTER SET B-1
APPENDIX C - CONVAL FUNCTION C-1
APPENDIX D - GTL RUN TIME ERROR MESSAGES D-1
APPENDIX E - REFERENCES E-1

I. INTRODUCTION

Since the beginning of Newell, Simon, and Shaw's list processing language, IPL, in 1954, the role of symbol manipulation languages in computer applications has become increasingly important. In 1965, a LISP interpreter was implemented on the Burroughs B 5500 here at Georgia Tech. For several years, it was used quite successfully in classroom instruction and in a few small scale symbol manipulation applications. Since the interpreter was too slow and too restrictive for any large scale applications, a decision was made to implement a high level symbol manipulation language by extending the existing, and excellent, B 5500 ALGOL compiler. The result was GTL, an acronym for Georgia Tech Language.

The GTL compiler is truly an extension of the Burroughs B 5500 ALGOL compiler; hence, it contains all features of Burroughs Extended ALGOL. (As used at Georgia Tech, STREAM PROCEDURES are prohibited.) Only one class of exception exists. The addition of certain GTL constructs to the ALGOL compiler has introduced new reserved words which cannot be used as defined identifiers by the programmer. These words are CAR, CDR, COMPLEX, CTR, EQ, FIELD, NEQL, NIL, RECORD, STRING, and SYMBOL.

In addition to its symbol manipulation capabilities, GTL also contains significant extensions to B 5500 ALGOL for numeric computation. GTL contains facilities for: double precision, complex, and double precision complex arithmetic; string manipulation; list processing (a non-standard version of LISP 2); record processing (linked disk records or "plex" processing); syntax-directed parsing; extended input-output functions (including special functions for remote terminal files); and other miscellaneous ALGOL

extensions (including additional intrinsic functions, the BASIC compiler matrix functions, an efficient means of swapping the contents of two arrays, a random number generator, and several other useful constructs).

Almost all of the major features of GTL were implemented prior to 1970. Some of the miscellaneous extensions, some of the inevitable error corrections, and updates to later versions of the ALGOL compiler have been accomplished since that time. All the features of GTL, as described in this manual, are currently being used by a large number of Georgia Tech students, faculty, and research workers. It is currently running under the Burroughs Mark XII Data Communications and Time Sharing Master Control Programs.*

Most of the features of GTL were implemented and made operational in successive stages. As each new feature was implemented it was described in a separate publication. Altogether, ten of these preliminary draft manuals were published between May 1968 and December 1969. The contents of these ten preliminary drafts have been consolidated into this single manual with a small amount of editing, rearrangement, and with the incorporation of some new material. The preliminary drafts are now obsolete, and this manual should be considered the official and complete documentation for GTL.

Comments, suggestions, or corrections to this manual or the GTL language are welcomed and should be forwarded to the Director, Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332.

---

*As of November 1971, GTL is being updated to Mark XIII.0 which provides a new COMPLEX Polish statement. Since this conflicts with the more convenient GTL COMPLEX construct, it is planned to omit this particular Mark XIII.0 feature. With this omission, GTL will no longer be a true extension of Burroughs ALGOL.

II. MISCELLANEOUS EXTENSIONS OF ALGOL

The GTL system contains a number of miscellaneous extensions of the ALGOL framework in which it is embedded. Those are described in detail below.

2.1 SINGLE PRECISION STANDARD FUNCTIONS

In addition to the standard (or "intrinsic") functions already provided by the B 5500 ALGOL compiler, GTL provides the following new single precision standard functions:

Name	Meaning
LOG	logarithm (base 10)
ARCSIN	inverse sine
ARCOS	inverse cosine
TAN	tangent
COTAN	cotangent
SINH	hyperbolic sine
COSH	hyperbolic cosine
TANH	hyperbolic tangent
GAMMA	gamma function
LNGAMMA	natural logarithm of gamma function
ERRORF	error function

2.2 CASE EXPRESSION

The syntax of expressions of type REAL, BOOLEAN, DOUBLE, COMPLEX, DOUBLE COMPLEX, SYMBOL, and "reference" (disk record address) has been extended by the inclusion of the "CASE expression", an expression having the same form

as the CASE statement of Burroughs Extended ALGOL with the statements replaced by expressions of the appropriate type. For example, if X, Y, and Z are REAL variables, then

CASE J OF BEGIN X; Y; Z; END

is an expression of type REAL, the value of which is the value of X if J is 0, Y if J is 1, or Z if J is 2, or an error termination otherwise.

2.3 FOR STATEMENT

The syntax of the FOR statement has been extended by allowing a single unsigned integer or simple variable to appear between the FOR and DO, indicating that the controlled statement is to be executed the number of times given by the value of the variable or integer. For example, if X is a real variable which has a value of 100, then

```
FOR X DO STMT
FOR 100 DO STMT
```

both have the effect of causing STMT to be executed 100 times.

2.4 EXIT STATEMENT

The word EXIT may be used in any block which is not a procedure body to cause an immediate exit from that block. The EXIT statement may appear anywhere in the block and may appear any number of times.

2.5 RETURN STATEMENT

The RETURN statement may be used to cause an immediate exit from any procedure in which it appears. If the procedure is typed, then the procedure

is given the value of the expression immediately following the word RETURN. The RETURN statement may appear anywhere in the procedure declaration and may appear any number of times (if the RETURN statement appears in a block, then that block must constitute the procedure body). For example, the LISP function MEMBER (a GTL standard function) may be defined as

```
BOOLEAN PROCEDURE MEMBER(X,Y); VALUE X,Y; SYMBOL X,Y;
     FOR Y IN Y DO IF X = Y THEN RETURN TRUE
```

2.6 ERROR STATEMENT

A convenient way of providing an immediate exit from any point in a program in which an error condition is detected is the ERROR statement. An execution of the ERROR statement will cause the value of its argument to be printed in a 2 character alpha format, together with the segment and relative address in the program of the ERROR statement. After the execution of the ERROR statement the program is immediately terminated. For example, execution of

```
ERROR("E3")
```

will cause "E3" to be printed and the program to be terminated.

2.7 MATRIX MANIPULATION

GTL provides a limited amount of matrix manipulation (using the intrinsic functions provided by Burroughs for the BASIC compiler). The matrix operations are addition, subtraction, multiplication, inversion, transposition, and assignment. There are 10 basic constructs which are illustrated below.

ARRAY A,B,C[0:10,0:10]   sample declaration

```
 1)  A:= B + C            addition
 2)  A:= B - C            subtraction
 3)  A:= B ⊗ C            multiplication
 4)  A:= 1 / B            matrix inversion
 5)  A:= B *              transpose
 6)  A:= B ⊗ primary      multiplication by a scalar
 7)  A:= B                simple assignment
 8)  A:= IDN              identity matrix assignment
 9)  A:= ZER              zero matrix assignment
10)  A:= CØN              unit matrix assignment
```

where primary is any arithmetic primary; e.g.,

```
A:= B ⊗ 2
A:= B ⊗ (SIN(X) + 1)
```

All arrays must be two dimensional and may never be specified as SAVE. The lower bounds of the arrays must be declared to be 0; however, they are treated as if they had lower bounds of 1; for example, the arrays declared above are considered to be 1 by 10 matrices. The intrinsic functions use the declared sizes of the arrays for their activities, not the amount of information the programmer has placed into the arrays, necessarily.

The last four modes of assignment are vastly more efficient than the equivalent open GTL code and should be used whenever that type of assignment is desired.

2.8 POWERS OF TEN TABLE

GTL provides access to a table containing powers of ten. It may be referenced with a construct of the form

TEN[<u>aexp</u>]

where <u>aexp</u> represents an arithmetic expression which, when integerized, will have a value from zero to 69. The value of this subscript should be the (integral) power of ten desired:

TEN[<u>aexp</u>] is equivalent to 10 * (<u>aexp</u>)

1/TEN[<u>aexp</u>] is equivalent to 10 * (-(<u>aexp</u>))

When used in a double precision context, it yields a double precision value; in a single precision context, its value is the double precision value truncated to a single word.

The use of this construct is encouraged since it provides a much more efficient means of calculating a power of ten than do the alternate forms. The object program uses the powers of ten table for I/O conversion, so its use will not further increase core requirements.

2.9 SWAP STATEMENT

The fastest and easiest way to swap two two-dimensional arrays is by the following construct:

SWAP(A1,A2)

where A1 and A2 are two array identifiers. The effect is to swap the contents and sizes of the two arrays. The actual implementation swaps only the pointers to the arrays, rather than the information in the arrays themselves.

2.10 RANDOM NUMBER GENERATOR

GTL contains a built-in random number generator which the programmer may reference directly through the arithmetic primary

CONVAL(0)

Each call on CONVAL(0) will generate a new random number between 0 and 1, but never 1. The arithmetic primary

CONVAL(1)

will return the previously-generated random number and will not generate a new one. If it is desired to change the stream of random numbers being generated, an alternate form of CONVAL(0) may be used, involving the following arithmetic primary

CONVAL(0,ae)

Depending on the value of ae, a different seed for random number generation will be used. In many applications, the following special form of CONVAL is used once at the beginning of the program:

CONVAL(0,TIME(4))

This presents the program with one of 64 different streams of random numbers, usually different each time the program is used, dependent upon the machine clock.

2.11 STATEMENT LINE NUMBER DETERMINATION

The line number of the current statement in a program may be accessed through the arithmetic primary

LINENUMBER

This is convenient in many applications, especially for debugging. If the programmer defines this identifier for his own use, it loses this meaning.

III. DOUBLE PRECISION ARITHMETIC

3.1 INTRODUCTION

In GTL, the declarator DOUBLE may be used in the same manner in which the declarator REAL is used in an ALGOL program. For example:

```
                    DOUBLE X, Y, Z
                  DOUBLE ARRAY DR[0:99]
         DOUBLE PROCEDURE DSINH (X); VALUE X; DOUBLE X;
            BEGIN DOUBLE Y;
                DSINH:= ((Y:= EXP(X)) - 1.0/Y) ⊗ .5
            END
```

Calculations with such variables, elements of arrays, and procedure values will automatically be done in double precision, subject to the rules of context (Subsection 3.6) and the available double precision operators and standard functions (Subsections 3.3, 3.4, 3.5). The Input-Output mechanism (Section IX) facilitates reading and writing double values.

3.2 FORM FOR DOUBLE EXPRESSIONS

A double expression has the form of an ordinary ALGOL arithmetic expression with double primaries and/or single precision primaries. A double primary can be a double variable, a double function designator (a call on a double-valued procedure or on a double standard function with its actual parameters, if any), a double assignment statement, a double expression within parentheses, READN in a double context (see Subsection 3.6 for context rules and Section IX for READN), or a constant appearing in a double context.

3.3 DOUBLE ARITHMETIC OPERATORS

The operators available for double precision arithmetic are +, -, ⊗, /, and MOD. DIV may be used between double primaries, but the calculation of the result will always be done in single precision.

3.4 DOUBLE RELATIONAL OPERATORS

All the relational operators, =, ≠, <, >, ≤, ≥, and their mnemonics, are available for double precision comparison. A comparison is a double precision comparison only when the expression on the left hand side of the relational operator is a double variable, double procedure, or double assignment statement. A double assignment statement is one in which the leftmost variable is double.

3.5 DOUBLE STANDARD FUNCTIONS

The available standard (or "intrinsic") functions of double expressions are as follows:

FUNCTION	MEANING
COS	cosine
SIN	sine
EXP	exponential function
LN	natural logarithm
LOG	common logarithm
SQRT	square root
ARCTAN	inverse tangent
LOPART	least significant part of double value
HIPART	most significant part of double value

The other functions available for single precision can be applied to double expressions, but the calculation of the function value will always be performed in single precision.

3.6 RULES OF CONTEXT

Whether single or double precision calculations are performed to evaluate an arithmetic expression depends on whether the arithmetic expression is in a single or double context. If the context is single, the calculations are done in single precision. If the context is double, all the calculations are done in double precision except for the operator DIV and intrinsic functions not avilable in double (which are done in single precision). When a double variable or double procedure is used in a single precision context, the double value is normalized and truncated to a single precision value. When a single precision variable, procedure, or standard function is used in a double context, it is converted to a double precision operand by setting the least significant part of the double operand to zero.

An arithmetic expression is in a double context in any one of the following cases and is otherwise in a single context:

1) If the arithmetic expression is on the right hand side of a := in an assignment statement, it is in a double context if, and only if, the variable immediately to the left of the := is a double variable.

2) If the arithmetic expression is an argument of a procedure for which the corresponding formal parameter is double, the arithmetic expression is in double context.

3) If the arithmetic expression is the expression on the right hand side of the relational operator of a double precision comparison (see Subsection 3.4).

3.7 DOUBLE PRECISION INPUT-OUTPUT

Ordinary ALGOL READ statements can be used to read single precision numbers to be used in double context, and ordinary ALGOL WRITE statements can be used to write a double value in single precision. ALGOL READ and WRITE statements cannot read and write double precision numbers. However, the I/O facilities of GTL facilitate the reading of double numbers and the writing of double numbers and editing phrases.

The GTL Input-Output system provides a very flexible and powerful means of reading and writing many types of data using any of the files which can be declared in normal ALGOL. The system is described in detail in Section IX of this manual.

3.8 RESTRICTIONS

If a double formal parameter is call-by-name and the corresponding actual parameter is a variable, this variable must be double.

If the actual procedure corresponding to a formal procedure is to have a double parameter, then in the actual procedure, that parameter must be double call-by-value, and the expression used as the corresponding parameter of the formal procedure must begin with a double variable. When the formal procedure and its double arguments are compiled, the compiler will print a warning message indicating the requisite type of the corresponding actual parameter of the actual procedure.

3.9 EXAMPLE PROGRAM

The following example double precision program is not intended to represent a practical program, but merely serves to illustrate some of the GTL double precision constructs. The GTL Input-Output system, which is described in Section IX, is also included in the example.

```
BEGIN COMMENT  FIND DOUBLE PRECISION ROOTS OF QUADRATIC EQUATIONS;
FILE IN INCD (2,10);
STRING CRD(80);
FILE OUT PRINTER 16(2,15);
STRING LIN (120);
REAL J,K;
PROCEDURE QUADSOLVE(A,B,C);
       VALUE A,B,C;
       DOUBLE A,B,C;
       BEGIN
       DOUBLE D;
       PRINT A,B,C;
       IF D := B * 2 - 4 × A × C < 0 THEN
              PRINT SPACE(10) #COMPLEX ROOTS#
       ELSE
              BEGIN
              C := (D := SQRT(D) + B) / A := 2 × A ;
              B := (D - B) / A ;
              PRINT #REAL ROOTS: # B, C;
              END;
       END OF QUADSOLVE;
       INPUT(INCD,CRD,80);
       OUTPUT(PRINTER,LIN,120);
       PRINT #QUADSOLVE PROGRAM#;
       COMMENT READ IN NUMBER OF TRIPLES TO BE READ IN;
       K := READN;
       FOR J:=1 STEP 1 UNTIL K DO QUADSOLVE(READN,READN,READN);
END.
```

THE PROGRAM HAD THE FOLLOWING CARD INPUT:

```
5
1 2 3  .002 3.14 .42  .001 .1 .1   .01 .01 .01  .08 .06 .07
```

AND THE OUTPUT LISTING WAS:

```
QUADSOLVE PROGRAM
1 2 3
          COMPLEX ROOTS
2@-3 3.14 .42
REAL ROOTS: 7.8486623064061050224428@2 1.56986623064061050224 4@3
1@-3 .1 .1
REAL ROOTS: 4.8989794855663561963944@1 9.8989794855663561963944@1
1@-2 1@-2 1@-2
          COMPLEX ROOTS
8@-2 6@-2 7@-2
          COMPLEX ROOTS
```

IV. COMPLEX ARITHMETIC

4.1 INTRODUCTION

In GTL, the declarator COMPLEX may be used in the same manner in which the declarator REAL is used in an ALGOL program. For example,

```
COMPLEX Y
COMPLEX ARRAY A[0:5]
COMPLEX PROCEDURE ROOT1(A,B,C);
     VALUE A,B,C;
     REAL A,B,C;
     ROOT1 := (-B + SQRT(B*2 - 4⊗A⊗C))/(2⊗A)
```

Such variables, elements of arrays, and procedure values will be automatically treated as complex numbers and may be used in the same manner as reals.

The Input-Output mechanism (Section IX) facilitates reading and writing complex values.

Complex arithmetic in double precision is also available (Subsection 4.7).

4.2 FORM FOR COMPLEX EXPRESSIONS

A complex expression has the form of an ordinary ALGOL arithmetic expression with complex primaries, with the exception of exponentiation (*), in which case the exponent must be real. (In other words, in A*B, A may be complex but B must be real.)

A complex primary can be an ordinary real-valued primary, a complex variable, a complex function designator (a call on a complex-valued procedure or on a complex standard (intrinsic) function with actual

parameters, if any), a complex assignment statement, a complex expression within parentheses, or :complex primary. The colon (:) in a complex primary indicates multiplication by i; i.e., the : is syntactically equivalent to SQRT(-1)⊗. For example, if X is a complex variable and X := 3+:7, then the real part of X is 3 and the imaginary part is 7. Since the colon means "i times," it must be followed by a primary; 7:, for instance, has no meaning.

Examples of complex expressions, where A and B are real variables and X and Y are complex variables:

```
A+:B
X/Y
COS(X+A) + :SIN(:ARG(X))
(X+Y)*A
::A - B      (i.e., -A-B)
X + :(A+B)
```

4.3 COMPLEX ARITHMETIC OPERATORS

The operators are +, -, ⊗, /, *, MOD, DIV. The meaning of the operators is illustrated by the table of equivalent algebraic expressions given below, where Z1 and Z2 are complex numbers such that Z1 = a + ib and Z2 = c + id and a,b,c,d, and R are real numbers.

EXPRESSION	DEFINITION
Z1+Z2	$(a+c) + i(b+d)$
Z1-Z2	$(a-c) + i(b-d)$
Z1⊗Z2	$(ac-bd) + i(bc+ad)$
Z1/Z2	$\frac{(ac+bd)}{(c^2+d^2)} + \frac{i(bc-ad)}{(c^2+d^2)}$

Z1*R	$\lvert Z1 \rvert^R e^{i\ R\ arg(Z1)}$
Z1 MOD Z2	$(ac+bd)MOD(c^2+d^2) + i((bc-ad)MOD(c^2+d^2))$
Z1 DIV Z2	$(ac+bd)DIV(c^2+d^2) + i((bc-ad)DIV(c^2+d^2))$

4.4 COMPLEX RELATIONAL OPERATORS

Two relational operators, = and ≠, and their mnemonics, are available for complex comparisons. Two complex expressions A and B are = if and only if the real part of A is equal to the real part of B and if the imaginary part of A is equal to the imaginary part of B. Otherwise, the ≠ relation is true. The left hand side of a complex relation must be a complex variable (including complex array elements) or a complex assignment statement (i.e., the leftmost variable must be a complex variable) and the right hand side can be any complex expression (including reals). For example, if X is complex and A is real, then X=A only if the real part of X equals A and if the imaginary part of X is zero.

4.5 COMPLEX STANDARD FUNCTIONS

The available intrinsic functions of complex expressions are given in the following chart. Assume X:= 1+:1 and Y:= 3+:4.

FUNCTION	MEANING	TYPE OF RESULT	EXAMPLE
ABS	absolute value	real	ABS(Y) = 5
ARG	argument	real	ARG(X) = .78540
CONJ	conjugate	complex	CONJ(X+Y) = 4-:5
SQRT	principal	complex	SQRT(:20-21) = 2+:5
IMAGPART	imaginary part	real	IMAGPART(Y) = 4
REALPART	real part	real	REALPART(Y) = 3

FUNCTION	MEANING	TYPE OF RESULT	EXAMPLE
SIN	sine	complex	SIN(X) = 1.2984 + :.63496
COS	cosine	complex	COS(X) = .83373 - :.98890
EXP	exponential function	complex	EXP(X) = 1.4687 + :2.2874
LN	principal value of natural logarithm	complex	LN(X) = .34657 + :78540

4.6 COMPLEX INPUT-OUTPUT

Ordinary ALGOL READ and WRITE statements can be used to read and write complex numbers if the real parts and the imaginary parts of the numbers are read and written separately as real numbers. However, the I/O facilities of GTL facilitate the reading of complex numbers and the writing of complex numbers and editing phrases.

The GTL Input-Output system provides a very flexible and powerful means of reading and writing many types of data using any of the files which can be declared in normal ALGOL. The system is described in detail in Section IX of this manual.

4.7 DOUBLE COMPLEX DECLARATOR

In a GTL program, COMPLEX declarations may be replaced by DOUBLE COMPLEX declarations for complex arithmetic in double precision. The "Rules of Context" described in Subsection 3.6 of this manual apply. The arithmetic operators available for DOUBLE precision COMPLEX are +, -, ⊗, /, *, MOD; the relational operators are = and ≠ and their mnemonics, and the intrinsic functions are REALPART, IMAGPART, ARG, ABS, CONJ. All the other operators and functions available for complex can be applied to DOUBLE COMPLEX, but the calculation will be done in single precision. When a DOUBLE COMPLEX variable, assignment statement, or typed procedure appears in the list of a PRINT statement, the real and imaginary parts are printed in double precision.

4.8 RESTRICTIONS

If a complex parameter is call-by-name and the actual parameter expression is a single variable, this variable must be complex.

If the actual procedure corresponding to a formal procedure is to have a complex parameter, then in the actual procedure, that parameter must be complex call-by-value and the expression used as the corresponding parameter of the formal procedure must begin with a complex variable. When the formal procedure and its complex arguments are compiled, the compiler will print a warning message indicating the requisite type of the corresponding actual parameter of the actual procedure.

4.9 EXAMPLE PROGRAM

The following example program uses a simplified portion of Robert Rodman's "Muller's Method for Finding Roots of an Arbitrary Function," (Algorithm 196, CACM, Vol. 6, August 1963), which finds real and complex roots of an arbitrary function. Given the starting values P1, P2, and P3, a limit MXM on the number of iterations, and convergence criteria EP1 and EP2, the procedure Muller listed below attempts to find a root to the function FUNCTION. This example also illustrates the GTL Input-Output system as described in Section IX. A listing of the compilation of the complete program and output is given. The card input was the following set of numbers, in order:

-1 0 1 30 @-8 @-8

The compilation listing is as follows:

```
BEGIN
FILE IN INFILE (2,10);
FILE OUT LINE 16(4,15);
STRING CRD(80),LIN(120);
COMPLEX PROCEDURE SPF(A,B);
        VALUE A,B;
        COMPLEX A,B;
        BEGIN
        A := SQRT(A);
        RETURN IF ABS(B+A) < ABS(B-A) THEN B-A ELSE B+A;
        END OF SPF;
PROCEDURE MULLER(P1,P2,P3,MXM,EP1,EP2,FUNCTION);
        VALUE P1,P2,P3,MXM,EP1,EP2;
        REAL P1,P2,P3,EP1,EP2;
        INTEGER MXM;
        COMPLEX PROCEDURE FUNCTION;
        BEGIN
        INTEGER ITC;
        COMPLEX X1,X2,X3,T1,FX1,FX2,FX3,H,LAM,DEL,G;
        LABEL M9,M8,M6;
        X1 := P1;  X2 := P2;  X3 := P3;
        FX1 := FUNCTION(X1);
        FX2 := FUNCTION(X2);
        FX3 := FUNCTION(X3);
        H := X3 - X2;
        LAM := IF X2 EQL X1 THEN 1 ELSE H / (X2 - X1);
        DEL := LAM + 1;
M9:     IF FX1 EQL FX2 AND FX2 EQL FX3 THEN
               BEGIN LAM := 1; GO TO M8; END;
        T1 := 4 × FX3 × DEL × LAM × (FX1 × LAM - FX2 × DEL + FX3);
        G := LAM × LAM × FX1 - DEL × DEL × FX2 + FX3 × (LAM + DEL);
        LAM := (-2 × FX3 × DEL) / SPF(G × G + T1, G);
M8:     ITC := ITC + 1;
        X1 := X2; X2 := X3; FX1 := FX2; FX2 := FX3;
        H := LAM × H;
M6:     DEL := LAM + 1;  X3 := X2 + H;  FX3 := FUNCTION(X3);
        IF FX2 NEQ 0 THEN
        IF ABS(FX3/FX2) GTR 10 THEN
               BEGIN LAM := LAM / 2;  H := H / 2;  GO TO M6; END;
        IF ABS((X3-X2)/X2) GTR EP1 AND ABS(FX3) GTR EP2 AND ITC LSS MXM
               THEN GO TO M9;
        PRINT #THE ROOT FOUND IS#, SPACE(5), X3;
        PRINT #THE FUNCTION EVALUATED AT THIS POINT IS#, SPACE(5), FX3;
        END OF MULLER;
COMPLEX PROCEDURE F(Z);
        VALUE Z;
        COMPLEX Z;
        RETURN Z×(Z×(Z×(Z×(Z×(Z×(Z×(Z+1)+3)+2)+3)-1)+3)-2)+1;
        COMMENT END OF DECLARATIONS;
        INPUT(INFILE,CRD,80);
```

```
      OUTPUT(LINE,LIN,120);
      NTS(*,11);
      MULLER(READN,READN,READN,READN,READN,READN,F);
END.
```

THE OUTPUT FROM THE PROGRAM LOOKS LIKE THIS:

```
THE ROOT FOUND IS         .67985375026
THE FUNCTION EVALUATED AT THIS POINT IS        2.0530168442
```

V. STRING PROCESSING

5.1 STRING VARIABLES

5.1.1 Simple String Variables and Arrays

In GTL, a string variable contains a string of characters; just as in ALGOL a variable of type REAL contains a number. String variables are declared with declarator STRING in the same forms as REAL, INTEGER, and BOOLEAN variables in ALGOL. The declaration of string variables which are not formal parameters of a procedure must also contain a "size part" which specifies the size of a string variable, i.e., the maximum number of characters which a string variable can contain. The simplest form of the size part is an unsigned integer enclosed in parentheses. The syntax of the string variable declaration is illustrated by the following examples:

```
STRING STR(5)
STRING CARD1, CARD2(80), LINE(120)
OWN STRING TEMP(26)
STRING ARRAY SR[0:9](10)
STRING ARRAY  SA1, SA2 [1:100](8)
```

Thus the simple string variable STR can contain at most 5 characters, CARD1 and CARD2 at most 80 characters, etc. The specification OWN in this context has the same functional meaning as other types of OWN variables in ALGOL. Similarly, each element of the string array SR can contain at most 10 characters.

The size of a string variable cannot exceed 8184 characters.

5.1.2 Substring Variables

The declaration of a string variable which is not a formal parameter of a procedure may contain the declaration of a substring variable. A substring variable is a string variable which references only a fixed part of (a "substring" of) the string variable which is declared. The substring variable identifier appears in the size part of the string declaration. The size part of the string variable declaration may now be defined as a list of one or more string length specifications enclosed in parentheses. Each string length specification is either

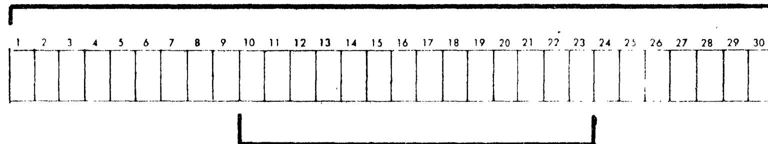
1) an unsigned integer, or

2) a substring variable identifier followed by a size part.

Two or more string length specifications are separated by commas. The sum of the unsigned integers in the size part determines the length of the string being declared. For example,

```
STRING A(9, B(14), 7)
```

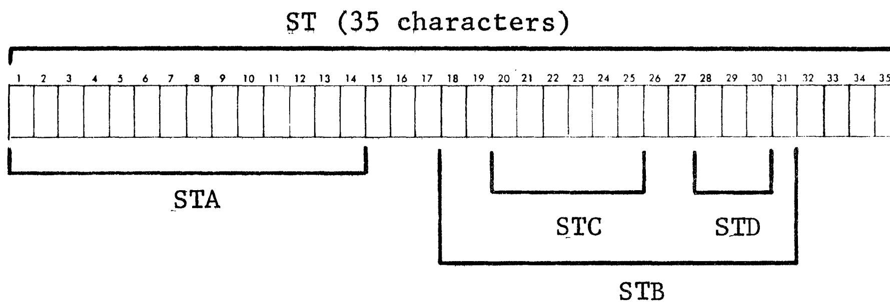
means that A is a string variable which can contain at most 30 characters, and B is a string variable which is a substring of A containing at most 14 characters. The sum of the string length specifications occurring before the substring variable identifier determines the number of character positions to be skipped in the main string before reaching the starting character position of the substring. The character positions of a string variable may be illustrated graphically by a set of contiguous "boxes", each box representing a single character position. Thus, the string variable A and its substring B, may be displayed graphically as follows:



Note that the definition of the size part allows the declaration of substring variables to be "nested"; i.e., a substring variable may contain a substring variable. For example,

```
STRING ST(STA(14),3,STB(2, STC(6),2, STD(3),1), 4)
```

may be displayed graphically as



When two or more string variables are associated with a size part which contains substring variables, the main string with which a substring identifier is to be associated in any particular instance must be given explicitly. For example, with the string declaration,

```
STRING A, B, C(72, SEQ(8))
```

one of the following forms must be used when referring to SEQ:

SEQ IN A

SEQ IN B

SEQ IN C

This form of the substring variable may be used like any other string variable. Ambiguously defined substrings of subscripted string variables are handled in the same way. For example, with the following string array declaration,

STRING ARRAY R, S [0:99](T(1),7)

if J represents a subscript expression for elements of the string arrays R and S, then one of the following forms must be used when referring to the substring T:

T IN R [J]

T IN S [J]

This form of the subscripted string variable may be used like any other string variable.

5.1.3 Formal String Variables

A formal string variable, i.e., a string variable which is a formal parameter of a procedure, is declared without a size part. The maximum number of characters that a formal string variable can contain will depend on the size of the corresponding actual parameter of the function designator. (See paragraphs 5.7.1 and 5.7.2.) In a procedure declaration which contains a formal string variable, the size of the string variable may be determined by the application of the GTL standard function LENGTH to the formal string

variable identifier, For example, if STR is a simple formal string variable and STRA is a formal string array, the

LENGTH(STR)

and

LENGTH(STRA)

gives the sizes of these formal string variables.

5.2 STRING DESIGNATOR

In GTL, the string designator is a construct which allows the programmer to refer to a string variable or any proper substring thereof. The definition of string designator includes the string variable, and has the three following forms:

SV

SV(ae1)

SV(ae2,ae2)

where SV represents a string variable, either simple or subscripted, and ae1 and ae2 represent arithmetic expressions. The first form of the string designator is simply the string variable itself. The second form of the string designator is the substring of SV obtained by skipping over the first ae1 characters in SV; the size of the substring is the number of remaining characters. The third form of the string designator is the substring of SV obtained by skipping over the first ae1 character positions in SV and its size is specified by ae2. (If the values of ae1 and ae2 are not non-negative integers, then they are converted into this form.)

For example, given the string declaration,

```
STRING CARD (72, SEQ(8))
```

the string designator

```
CARD(72)
```

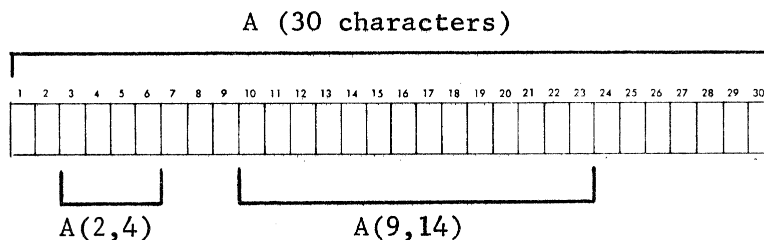
refers to the same substring of CARD as the substring variable SEQ. Given the string declaration,

```
STRING A(9, B(14), 7)
```

the string designator,

```
A(9, 14)
```

refers to the same substring of A as the substring variable B. The string designators A(2,4) and A(9,14) are illustrated graphically below.



5.3 STRING EXPRESSIONS

5.3.1 String Expression Forms

In ALGOL, an arithmetic expression may be considered as a set of rules which, when executed, generates a value which is a number; in GTL, a string expression is a set of rules which produces a value which is a string of characters.

A string expression is either a string primary, which has a string of characters as its value, or two or more string primaries separated by ampersands (& - the "concatenate operator"). The latter form has as its value the string produced by "joining together" the values of the constituent string primaries. The string primaries are described in paragraphs 5.3.2 through 5.3.19.

5.3.2 The Quoted String

The quoted string has the same syntactical form as a string in Burroughs Extended ALGOL, i.e., a string of characters enclosed in quotes ("). The quote mark itself may be quoted: """". The value of the quoted string is the string of characters appearing between the quote marks. The quoted string may not exceed 420 characters in length.

Examples:

```
"A"
"THIS IS A QUOTED STRING"
"*"
""""
```

5.3.3 String Designator

When used as a string primary, the value of the string designator is the string of characters in the string variable, or substring thereof, referenced by the string designator.

5.3.4 String Assignment Statement

A string assignment statement, besides being used as a statement, may be used in a string expression having as its value the value which is

assigned to the variables in the left part list. An extension of the syntax of a string assignment statement is described in Subsection 5.4.

5.3.5 String Function Designator

A string function designator is a call on a procedure which was declared with the type STRING, its value being the value to which the string procedure identifier is assigned in the string procedure declaration. The value of a string function designator may not exceed 7 characters in length. The formal parameters of the procedure, if any, may be of any type, including the string formal parameters discussed in Subsection 5.7. For example,

```
STRING PROCEDURE REVERSE(S); VALUE S; STRING S;
   BEGIN REAL L;
         REVERSE:=
         IF (L:= LENGTH(S)) = 1 THEN S ELSE
         S(L-1,1) & REVERSE(S(0,L-1));
   END OF REVERSE
```

defines a procedure which has as its value the string of characters in the string S in reverse order:

```
REVERSE("ABC") = CBA
```

5.3.6 SPACE function

The SPACE function is used to generate a specified number of spaces in a string expression. The SPACE function may be used in one of the following forms:

n SPACE

where n is an unsigned integer, and

SPACE(ae)

where ae represents an arithmetic expression. The SPACE function will generate the number of spaces specified by the value of n or ae. For example, if J is a real variable having a value of 40 then,

40 SPACE

SPACE(40)

SPACE(J)

all have the effect of generating 40 spaces.

5.3.7 The NIL Function

The NIL function is used to generate a specified number of zeros (the character "0") in a string expression. The NIL function may be used in one of the two following forms:

n NIL

where n is an unsigned integer, and

NIL(ae)

where ae represents an arithmetic expression. The NIL function will generate the number of zeros specified by the value of n or ae. For example, if J is a real variable having the value of 40, then

40 NIL

NIL(40)

NIL(J)

all have the effect of generating 40 zeros.

5.3.8 The String Skip Indicator

The string skip indicator has the effect of creating a gap in the string being generated by the string expression in which it appears. Its actual effect depends on the context of the string expression: If it is used in a string expression which is assigned to string variable in a string assignment statement (see Subsection 5.4), it causes the indicated number of character positions to be skipped over in the string variable during the execution of the assignment statement (see paragraph 5.4.4). If it appears in the string expression on the right hand side of a string relational expression (see Subsection 5.5), it causes the indicated number of characters in the string designator (on the left hand side) to be ignored in the process of determining the value of the string relational expression (see paragraph 5.5.4).

The string skip indicator may be used in one of the two following forms:

$\underline{n}$*

where $\underline{n}$ is an unsigned integer, and

*($\underline{ae}$)

where $\underline{ae}$ represents an arithmetic expression; for example, if J is a real variable having a value of 40, then

40 *

*(40)

*(J)

all have the effect of causing 40 characters to be skipped over.

5.3.9 The QMARK Function

The QMARK function, used in the form,

QMARK

in a string expression will generate one question mark (?) or invalid character. This function is provided since the question mark cannot be quoted in a GTL program, and there would be no other simple way of generating this character.

5.3.10 The Bit Expression

A bit expression is a string primary which generates a string of bits (not the characters "1" and "0", but the actual bit components of characters). If any characters are generated in the remainder of the string expression, the first of these characters will start at a position equal to the position of the last character generated before the bit expression plus the smallest multiple of six bits containing the bit string (since a character is six bits long). The syntax and semantics of the bit expression are explained in Subsection 5.6.

5.3.11 The Restricted Boolean Expression

A restricted Boolean expression, i.e., a Boolean expression which does not begin with any of the other preceding string primaries, may be used as

a string primary itself. The value of the Boolean expression in this context will be a string of letters, "TRUE" or "FALSE", depending on the value of the Boolean expression.

5.3.12 The Restricted Arithmetic Expression

A restricted arithmetic expression, i.e., an arithmetic expression which does not begin with any of the other preceding string primaries, may be used as a string primary itself. The value of the arithmetic expression in this context will be an unsigned string of digits up to 7 characters in length representing the value of the arithmetic expression. If the value of the arithmetic expression cannot be represented in this way, then the string value of the expression is undefined. For example, if R is a real variable with a value of 25, then

$$R \otimes 10$$

when used as a string primary, will generate the string "250".

5.3.13 The Restricted Symbol Expression

A restricted symbol expression (Section VI), i.e., a symbol expression which does not begin with any of the other preceding primaries, may be used as a string primary itself. The value of the symbol expression in this context must be an atomic symbol, which is converted into the string which the atomic symbol represents. For example, if S is a symbol variable having as its value the list (THIS IS A LIST), then

```
CAR(S)
```

will cause the string "THIS" to be generated.

5.3.14 The STRING Transfer Function

The STRING transfer function, when used in the form,

STRING(ae)

where ae represents an arithmetic expression, will generate the equivalent alpha representation of the value of the arithmetic expression (up to 7 characters in length), left-justified. For example, if R is a real or alpha variable, the value of which is the alpha string "AB", then

STRING(R)

will generate the string "AB". If the value of R is 250, then "3," will be generated.

A string which is right-justified in a field of n characters may be produced by the following variant of the STRING transfer function:

STRING(aexp,n)

where aexp is the arithmetic expression to be converted, and n is an unsigned integer, ranging from 1 to 7, which specifies the size of the resulting string. For example, if R is a real variable containing the ALPHA string "AB", then the value of

STRING(R,4)

would be the string "00AB". If R has a value of 250, then

STRING(R,1)

would generate the string "3,".

5.3.15 The SUBST Function

The string-valued function SUBST allows the programmer to make character-for-character substitutions in a string variable. The SUBST function may be used in two forms, the first of which is

SUBST(string designator, substitution pair list)

where string designator must be a string variable, or designated substring thereof, and substitution pair list is a list of one to 12 substitution pairs of the form

matching character : substitution character

where substitution character is the quoted character which is to replace the quoted special character matching character. The matching character cannot be the "blank" character. The length of the string designator must not be greater than 126 characters, and if longer than 63 characters it must be an even number; also, the string variable may not be a formal parameter. The value of the SUBST function is the contents of the string designator after the substitutions have been made. For example, if the string CARD contains "AR[INDEX]", then

```
CARD:= SUBST(CARD(0,72),"[":"(","]":")")
```

will change this string to "AR(INDEX)".

The second form of the SUBST function is

SUBST(string designator, substitution table)

where string designator has the same meaning as above, and substitution table

is a simple string variable containing substitution characters for all of the 64 characters in the B 5500 character set. This "table" is indexed by the REAL value equivalent of the character to be replaced; for example, the replacement for the Jth character of the string CARD from the string TABLE would be

```
TABLE(REAL(CARD(J,1)),1)
```

For example, if the first 10 characters of TABLE are "123456789A" and CARD contains the string "539648", then

```
CARD:=SUBST(CARD(0,72),TABLE)
```

will change the string to "64A759". The substitution table must be at least 64 characters long.

5.3.16 The FILL Function

The FILL Function allows access to the B 5500 operator TBN (Transfer Blanks for Non-numerics). It may be used in GTL string expressions in two forms, the first of which is

FILL(aexp, n)

where aexp is an arithmetic expression whose value should be an integer, and n is an unsigned integer ranging from 1 to 8. The value of the arithmetic expression is converted into a string which is right-justified in the field of characters, whose length is specified by n. Characters to the left of the left-most digit of the string are set to the blank character. For example,

```
FILL(215,5)
```

generates the string " 215".

The second form of the FILL function is

FILL n

where n is an unsigned integer ranging from 1 to 63. This string primary is intended to be used in a string assignment statement where it will replace each zero digit or non-digit character in the "destination string" by a blank until a non-zero-digit is encountered. The number of characters tested will be n minus one characters (so that if the last character is a zero, it will not be replaced by a blank). For example, if the first 5 characters in the string CARD are "00215", then

CARD:=FILL 5

changes the string to " 215".

5.3.17 The OCTAL Function

The OCTAL function transforms a 48-bit B 5500 word into a 16-character string containing the equivalent octal value. This function has the following forms:

OCTAL(aexp)

OCTAL(string designator)

where aexp is any arithmetic expression and string designator is a string variable, or substring thereof, beginning at a word boundary (multiple of 8 characters), which is 8 characters long.

5.3.18 The String Repeat Expression

The string repeat expression, which has the form

[ae : se]

where ae represents an arithmetic expression, and se represents a string expression, will cause the value of the string expression to be generated repeatedly the number of times specified by the value of the arithmetic expression. For example,

[3: "AB"]

will cause the string

ABABAB

to be generated. The value of the arithmetic expression must be an integer less than 64.

5.3.19 Parenthesized String Expression

A string expression enclosed in parentheses may be used as a string primary. For example, if A and B are string variables, then the following are string primaries:

(A & "-" & B)

(A:=B & "S")

5.4 THE STRING ASSIGNMENT STATEMENT

5.4.1 The Basic String Assignment Statement

The basic string assignment statement has the same form as an ordinary ALGOL assignment statement. However, in addition to string variables, string designators may also appear in the left part list of the assignment statement. For example, if A is a string variable 30 characters in length which contains only spaces at the time of execution of the assignment statement, and B is a string variable, the first 5 characters of which is the string "CARRY", then

```
A:=A(10):=B(0,4) & "IES"
```

will change the contents of A as indicated by the following graphic representation:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
C	A	R	R	I	E	S				C	A	R	R	I	E	S													

If this assignment statement is used as a primary in a string expression, then its value would be the string "CARRIES". If the length of the string generated by the string expression exceeds the number of character positions available in any of the string designators appearing in the left part list of the string assignment statement, then an error message will be generated, and the program will be terminated.

5.4.2 String Assignment with SPACE

The word SPACE (see paragraph 5.3.6) may be used at the end of a string expression to indicate that all of the remaining character positions in the string into which the string expression is assigned is to be filled with

spaces. For example, if A is a 30 character string, the assignment statements

```
A:=B(0,5)  & SPACE
A:=SPACE
```

are equivalent to the assignment statements

```
A:=B(0,5)  & SPACE(25)
A:=SPACE (30)
```

5.4.3 String Assignment with NIL

The word NIL (see paragraph 5.3.7) may be used in the same way as SPACE in paragraph 5.4.2; for example,

```
A:=B(0,5) &  NIL
```

is equivalent to

```
A:=B(0,5) &  NIL(25)
```

5.4.4 String Assignment with String Skip Indicator

The string skip indicator as defined in paragraph 5.3.8 may be used in the string expression in a string assignment statement as a means of effectively combining two or more assignment statements into one; for example, if LINE is a 120 character string variable, and CARD is an 80 character string variable, then

```
LINE(8):=CARD(0,72) & *(8) &  CARD(72,8)
```

is equivalent to the two assignment statements:

```
LINE(8):=CARD(0,72)
LINE(88):=CARD(72,8)
```

If there is more than one string designator in the left part list of the assignment statement a gap caused by a string skip indicator is filled in with characters from the right most string designator in the left part list; for example, if A and B are string variables, the latter containing the string "ABCDEFGH", then the assignment statement

```
A:=B:="123" & 2 * & "678"
```

will set the value of A(0,8) to the string "123DE678".

5.4.5 String Assignment Overlap: A Warning

A string generated by a string expression in a string assignment statement is not generated in its entirety before it is transferred into the string designators in the left part list. Instead, as each character of the string is generated from the string expression, it is transferred into the rightmost string designator in the left part list. Then each character in the string thus generated is transferred, one by one, into the preceding string designator, and this process continues until the string has been transferred into all of the string designators in the left part list. The means by which string assignment statement is effected introduces a side effect which may not be obvious. Whenever the string variable referenced by the rightmost string designator in the left part list also appears as a part of a string designator in the string expression of an assignment statement, the characters to be referenced in the string expression may have

already been changed to new characters generated earlier in string expression; for example, if A is a string variable containing the string "12345678", then

```
A(2):=A(1,3)
```

will change A(0,8) to "12222678"; on the other hand

```
A(2):=A(4,3)
```

will change A(0,8) to "12567678". This side effect may be used to advantage; for example, the most efficient way to fill the 120 character string variable LINE with asterisks is

```
LINE:=[2:"****"] & LINE(0,112)
```

5.4.6 String Assignment Statement Containing String Length Assignment

Sometimes it is not easy or convenient to determine the length of a string generated by a string expression. For this reason the following option is provided for the string assignment statement: if a real variable followed by a colon is inserted between the := and the string expression in the string assignment statement, then the length of the string generated by the string expression is assigned to this variable. For example, if A is a string variable, R is a real variable, and S is a symbol variable, the value of which is the atomic symbol "ATOMICSYMBOL", then the execution of the string assignment statement

```
A:=R: S
```

will set the first 12 characters of the string variable A to the string "ATOMICSYMBOL" and will set the value of R to 12.

5.4.7 The String FILL Statement

When filling a string variable with a very long literal string, the string FILL statement may be used:

FILL stringid WITH STRING quoted string

where stringid is the name of a simple string variable, and quoted string is a string of characters enclosed by quotes which may be as long as 1022 characters. For example,

```
STRING STR(216)
   .
   .
   .
FILL STR WITH STRING "WHEN FILLING A STRING VARIABLE WITH A
VERY LONG LITERAL STRING, THE STRING FILL STATEMENT MAY BE USED."
```

5.4.8 The String Addition Assignment Statement

The addition of a + between the := and the string expression of a string assignment statement will cause the string generated by the string assignment statement to be added to the contents of the rightmost string designator in the left part list. For example, if CARD is an 80 character string variable such that the last 8 characters contain the string "12345000" and INCR is an 8 character string variable containing the string "00001000", then the string assignment statement

```
CARD(72,8):= + INCR
```

will cause the contents of CARD(72,8) to be changed to "12346000".

The value of the string expression in this context should be a string of digits not exceeding 63 characters in length. If there is an overflow in the most significant character position, then this overflow will be lost.

(This type of string assignment statement has the same effect as the statement

DS:=N ADD

in a stream procedure of Burroughs Extended ALGOL[4].)

5.4.9 The String Subtraction Assignment Statement

The addition of a - between the := and the string expression of a string assignment statement will cause the string generated by the string assignment statement to be subtracted from the contents of the rightmost string designator in the left part list. For example, if CARD is an 80 character string variable such that the last 8 characters contain the string "12345000" and DCR is an 8 character string variable containing the string "00001000", then the string assignment statement

CARD(72,8):= - DCR

will cause the contents of CARD(72,8) to be changed to "12344000". The value of the string expression in this context should be a string of digits not exceeding 63 characters in length. (This type of string assignment statement has the same effect as the statement

DS:=N SUB

in a stream procedure of Burroughs Extended ALGOL[4].)

5.5 STRING COMPARISON

5.5.1 String Relational Expression

The string relational expression may consist of a string designator followed by a relational operator followed by any string expression. Any

of the relational operators, =, ≠, <, >, ≤, ≥, or their mnemonics, may be used; comparisons are made on the basis of the ordering of the B 5500 collating sequence.[4] For example, if A, B, CARD, and SEQ are string variables, then

```
A(0,J) = B(0,J-2) & "LY"
CARD(72,8) > SEQ(0,8)
```

are examples of string relational expressions.

If the length of the field of characters specified by the string designator is not equal to the length of the string generated by the string expression, then the result of the comparison will depend on the relational operator: if the string expression length is greater than the string designator length, then the relation will be TRUE if the operator is ≠, FALSE otherwise; if the string expression length is less than the string designator length, then

(1) If the operator is =, then the relation will be FALSE.

(2) If the operator is ≠, then the relation will be TRUE, and;

(3) For any other operators, the value of the relation will depend only on the result of the comparison of the string generated by the string expression.

5.5.2 String Relation with SPACE

The word SPACE (see paragraph 5.3.6) may be used at the end of a string expression to indicate that all of the remaining characters yet to be compared in the string designator on the lefthand side of the string relation are to be compared with the "blank" character. For example, if

A is a 30 character string, the string relations

A = B(0,5) & SPACE

A = SPACE

are equivalent to the relations

A = B(0,5) & SPACE(25)

A = SPACE(30)

5.5.3 String Relation with NIL

The word NIL (see paragraph 5.3.7) may be used in string relations in the same way as SPACE in paragraph 5.5.2 above. For example,

A = B(0,5) & NIL

is equivalent to

A = B(0,5) & NIL(25)

when A is a 30 character string variable.

5.5.4 String Relation with String Skip Indicator

The string skip indicator, as defined in paragraph 5.3.8, may be used in the string expression in a string relational expression as a means of effectively combining two or more string relations into one; for example, if LINE is a 120 character string variable, and CARD is an 80 character string variable, then

LINE(8,88) = CARD(0,72) & *(8) & CARD(72,8)

is equivalent to the expression

LINE(8,72) = CARD(0,72) AND LINE(88,8) = CARD(72,8)

An asterisk by itself may be used at the end of the string expression on the righthand side of a string relation containing the relational operator = to indicate that, in case the length of the string generated by the preceding part of the string expression is less than the string designator length, the value of the relation will depend only on the result of the string comparison and not on the difference in length (which would ordinarily make the relation FALSE; see paragraph 5.5.1). For example, if A and B are 30 character strings, and J is a real variable, having a value of between 1 and 30,

A = B(0,J) & *

is equivalent to

A(0,J) = B(0,J)

5.5.5 <u>String Pattern Matching</u>

A limited amount of string pattern matching is made possible in GTL by extending the number of primaries allowed in the string expression on the righthand side of the string relational expression when the operators are = and ≠. In addition to the string primaries described in Subsection 5.3, a number of pattern matching primaries are allowed. These pattern matching primaries have the following forms

<u>n</u> P

P(<u>ae</u>)

P

where P represents a "pattern element", <u>n</u> represents an unsigned integer, and <u>ae</u> represents an arithmetic expression. The number of characters in the string designator tested for a match is determined by <u>n</u> or <u>ae</u>, and

must be less than 64. If the pattern element is given with neither _ae_ nor _n_, then only one character from the string designator is tested. A table of the pattern elements and the set of characters they match is given below.

Pattern Element	Characters matched
ALF	any alphanumeric character
LTR	any letter
DGT	any digit
VWL	any vowel (A, E, I, O, or U)
AMONG _qs_	any character in the quoted string represented by _qs_ (which must be less than 8 characters in length)

For example, if the string variable A contains the string "CARRIES", then the following relation will be TRUE:

A(0,7) = 4 LTR & 2 VWL & "S"

Also, if the string variable A contains the string "AACBABB", then the following relation will be TRUE:

A(0,7) = 7 AMONG "ABC"

Note that the pattern element VWL is equivalent to AMONG "AEIOU".

5.5.6 The SEARCH Function

The SEARCH function converts a string relational expression into a pattern searching function. It has the basic form:

SEARCH(_string designator_ _relop_ _string expression_)

where the _string designator_ is the test string, _relop_ is the relational

operator, and string expression generates the string which will be compared to successive substrings of the string contained in the string designator. The sequence of comparisons will continue until the relation is satisfied, or until no more comparisons are possible. If the length of the string contained in the string designator is denoted by L_1 and the length of the string generated by the string expression is denoted by L_2, then the maximum number of comparisons which can be made is $(L_1 - L_2 + 1)$; this number is limited to 63. The value of the SEARCH function is the number of character positions skipped in the string designator before a successful match is made; if no substring of the string designator satisfies the relation, then the value of SEARCH is $(L_1 - L_2 + 1)$. For example, if STR is a 7 character string, then

Contents of STR	SEARCH Function	Value of SEARCH
"ABCDEFG"	SEARCH(STR = "C")	2
"ABCDEFG"	SEARCH(STR = "G")	6
"ABCDEFG"	SEARCH(STR = "X")	7
"ABCDEFG"	SEARCH(STR = "CD")	2
"ABCDEFG"	SEARCH(STR = "XYZ")	5
"AR(X); "	SEARCH(STR = AMONG"(),")	2
"ABC456."	SEARCH(STR ≥ "0")	3
" 1023"	SEARCH(STR ≠ " ")	3

There is also an additional form of the SEARCH function

SEARCH(strexp, var)

where strexp is a string relational expression as defined above, and var is a REAL or ALPHA variable. If the match of a single character

succeeds, then this character is transferred into the last character position of var. Otherwise, var retains its former value. For example, given the 7-character string variable STR, and the REAL variable R (initially zero),

Contents of STR	SEARCH Function, R	VALUE of SEARCH	VALUE of R
"ABX5Y "	SEARCH(STR = DGT,R)	3	"5"
" 1023"	SEARCH(STR ≠ " ",R)	3	"1"
"AR(X); "	SEARCH(STR = AMONG "(),",R)	2	"("
"ABCDEFG"	SEARCH(STR = ".",R)	7	"0"

5.6 BIT EXPRESSIONS

5.6.1 Bit Expression Form

A bit expression is an expression which generates a string of bits. A bit string so generated may contain gaps, such as might be used for masking purposes, analogous in effect to the string skip indicator (see paragraph 5.3.8). Since a single character is six bits long, a bit string n bits long, including gaps, is considered to contribute ((n + 5) DIV 6) characters to the string expression in which bit expressions appear. A bit expression may consist of one or more bit primaries. Two or more bit primaries may be separated by + symbols.

5.6.2 Bit Primary

A bit primary may have one of the following forms:

n BIT1	BIT1(ae2, ae1)
n BIT0	BIT0(ae2, ae1)
BIT1(ae1)	BIT1
BIT0(ae1)	BIT0

BIT1 represents the bit "1" and BITO represents the bit "0". The number of bits to be generated is determined by the unsigned integer n or the arithmetic expression ae1 if given, or is 1 otherwise. If the arithmetic expression ae2 is also given, it determines the number of bit positions to be skipped before any bits are generated. If, for example, C1, C2, C3, C4, and C5 are real variables which have values of 1 and 0 only, J is a real variable, and CODE is a string variable, then

CODE(J,1) = BIT1(1-C1,C1) + BIT1(1-C2,C2) +
BIT1(1-C3,C3) + BIT1(1-C4,C4) + BIT1(1-C5,C5)

will be TRUE only if the nth bit of CODE(J,1) is 1 for each Cn which is 1, for n = 1, 2, ..., 5. If, in the example given above, the string relational expression were changed into a string assignment statement by replacing the = with a :=, and CODE(J,1) initially contained the character 0, then the execution of this assignment statement would place into CODE(J,1) a bit pattern corresponding to the sequence of values of the Cn's.

5.7 STRING ACTUAL PARAMETERS

5.7.1 Call-by-Value

When a formal string parameter is called by value, as in the example given in paragraph 5.3.5, then the corresponding actual parameter may be any string expression. However, the string generated by the string expression may not be longer than 7 characters in length.

5.7.2 Call-by-Name

When the formal parameter is a string array then the corresponding actual parameter may only be a string array identifier or string array row.

However, when the formal parameter is a simple string variable (call-by-name) then the actual parameter may be either a string designator or string assignment statement. The latter is not equivalent to calling a string expression by name. Instead, the string assignment statement is executed when the function designator is called, not when referenced inside the procedure body. The name of the leftmost string designator in the left part list of the string assignment statement is given to the string formal parameter. The length of the string formal parameter will be the length of the string generated by the string expression in the assignment statement.

Even though the actual parameter may be a string designator which refers to a substring of a string variable, the corresponding formal string variable may be used in the procedure body as if it were the name of an entire string variable, and itself may be used in a string designator; for example, consider the following procedure declaration:

```
PROCEDURE P(R,S); VALUE S; STRING R,S;
R(1,4):=S & "X"
```

If A is a string variable containing the string "ABCDEFGH" then

```
P(A(1,6),"RS")
```

will change the value of A(0,8) to "ABRSXFGH".

5.8 USING STRINGS IN OTHER TYPES OF EXPRESSIONS

5.8.1 Arithmetic Expressions

If the contents of a string designator is a string of less than 8 digits, it may be used in an arithmetic expression in the same manner as

any other arithmetic primary. In this context, the digit string is automatically converted into the integer which the string represents. For example, if A is a real variable and S is a string variable containing the string "125", then the assignment statement

```
A:=S(0,3) + 25
```

will set A to 150.

If A is an alpha or real variable, and S is a string variable containing the string "ABC" in its first three character positions, then

```
A:=S(0,3) + 25
```

will set A to 148, since "ABC" = "123" when zone bits are stripped.

If the string referenced by a string designator is less than 8 characters in length, then the transfer function REAL applied to that string designator may be used in an arithmetic expression in the same manner as any other arithmetic primary. In this context, the string is automatically converted into the number which represents the string (in ALPHA format, right justified). For example, if A is an alpha or real variable, and S is a string variable containing the string "ABC" in its first three positions, then

```
A:=REAL(S(0,3))
```

has the same effect as

```
A:="ABC"
```

The transfer function REAL may not be applied to a Boolean expression starting with a string designator because of the ambiguity with the context described above.

5.8.2 Symbolic Expressions

If the string referenced by a string designator is less than 32 characters in length, then it may be used in a symbol expression in the same manner as any other symbol primary. In this context, the string is automatically converted into the atomic symbol which represents the string. For example, if L is a symbol variable, the value of which is the list (IS A LIST), and S is a string variable containing the string "THIS", then

```
L:=CONS(S(0,4),L)
```

will cause L to be set to the list (THIS IS A LIST).

5.9 USING AN ARRAY OR A STRING VARIABLE

A single-dimensional array identifier may be used in place of a string variable in the left-part list of a string assignment statement, in a string expression, or in a string relational expression. For example,

```
STRING CARD(80); ARRAY AR[0:9]
        .
        .
        .
CARD(72,8):=AR(72,8)
 AR(7,8):=CARD(32,6) & SPACE
  AR(3,5) > CARD(3,5)
```

Note that whenever an array identifier appears in the left-part list of a string assignment statement, it is possible that the flag bit of one or more array elements may be set. If an array element is accessed in an arithmetic expression context and the flag bit is set, then a FLAG BIT run time error will occur.

5.10 OPTIMALITY OF STRING EXPRESSIONS

In all string expressions for which the compiler can determine at compile time exactly what actions are to be performed (all skip parts and size parts in a string expression or assignment statement are unsigned integers), then the code emitted is almost always more efficient than an equivalent STREAM PROCEDURE. This is not necessarily true otherwise.

5.11 READING AND WRITING STRINGS

5.11.1 READ and WRITE Statements

A string variable not a formal parameter and longer than 8 characters in length may appear in ALGOL READ and WRITE statements at any place at which an array row is allowed. As with the array row forms, the number of words to be read or written, rather than the number of characters, must be given since only multiples of 8 characters may be read or written. For example, if CARD is an 80 character string, INFILE is an input file, LINE is a 120 character string, and OUTFILE is an output file, then the following READ and WRITE statements are allowed:

```
READ(INFILE,10,CARD)
WRITE(OUTFILE,15,LINE)
```

5.11.2 GTL Input-Output Functions

In addition to the ALGOL READ and WRITE statements described above, strings may be easily read from or written onto any file using the GTL Input-Output functions described in Section IX.

VI. LISP 2

6.1 INTRODUCTION

Among the facilities for processing symbolic data, GTL contains a non-standard version of LISP 2, a list processing language[1].

This section of the manual describes the list processing constructs provided by the GTL language. Although enough information is provided in this section to enable the user to write a LISP program, it is intended to supplement, but not replace, the standard references on LISP[2,3,8,6].

6.2 S-EXPRESSIONS AND LISP RECORDS

6.2.1 Record and Field Designator

The name S-expression (or "Symbolic expression") is given to the symbolic (or "external") representations of LISP data. In order to define the S-expression and relate it to the various LISP operations, the concept of a linked record[9] is introduced.

A record, like an array in ALGOL, is a set of values. In ALGOL, an array is an ordered set of values all of the same type, each of which may be referenced by an appropriate subscript. A record, on the other hand, contains a set of fields, and each field contains a value, the type of which corresponds to the type of field. A collection of records is said to belong to the same record class if each record in the collection contains the same corresponding fields. The value of a field in a record is referenced by the application of the field name to a reference expression, the value of which is a reference to the record. In GTL, this construct is called a field designator, and has the form

FIELDNAME(re)

where FIELDNAME is the name of a field and re is a reference expression.

Certain classes of LISP records contain two fields which are directly accessible to the programmer, the names of which are CAR and CDR. The value of a CAR or CDR field is a reference to a LISP record. A LISP reference expression; i.e., an expression the value of which is a reference to a LISP record, is called a symbol expression. Thus,

CDR(se)

is a reference to the LISP record which is referenced by the value of the CDR field of the LISP record referenced by the symbol expression se.

6.2.2 LISP Records

There are three primary classes of LISP records:

1) A record which represents a string of characters. This type of record is called an atomic symbol, and contains a CDR field, but not a CAR field.

2) A record which represents a number. This type of record is called an atomic number, or simply number, and contains neither the CAR nor the CDR field.

3) A record which contains only the CAR and CDR fields. It represents a symbolic expression called a dotted pair, which is defined below.

LISP records of types 1 and 2, atomic symbols and atomic numbers, are called atoms because they are the basic components from which the symbolic representations of LISP data are constructed. Atoms are represented symbolically simply by the strings and numbers which they represent.

A LISP record of type 3 is represented symbolically by a dotted pair

which has the form

(s1 . s2)

where s1 and s2 are the symbolic representations of the LISP records referred to by the values of the CAR and CDR fields, respectively, of the LISP record. For example,

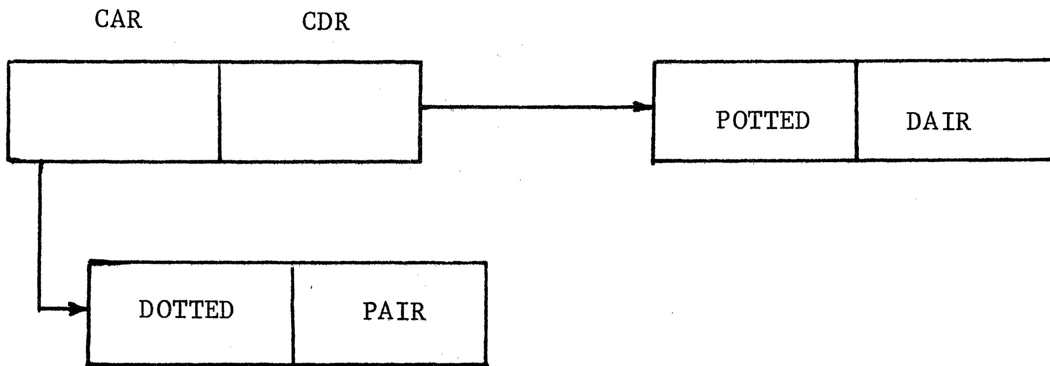
(DOTTED . PAIR)

is the symbolic representation of a LISP record for which the CAR field refers to the atomic symbol DOTTED, and the CDR field refers to the atomic symbol PAIR. Note that the definition of a dotted pair is recursive. If the values of the CAR and CDR fields of a LISP record are non-atomic, then the dotted pair representing the record will contain dotted pairs; for example,

((DOTTED . PAIR) . (POTTED . DAIR))

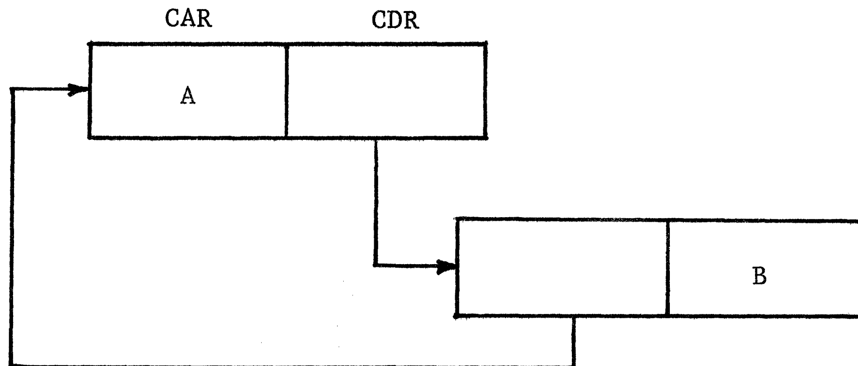
LISP records which represent dotted pairs can be represented graphically by a rectangle divided in half, the left half representing the CAR field and the right half representing the CDR field. Each half contains an atom if the corresponding field is atomic, or an arrow pointing to another rectangle if non-atomic. For example, the two dotted pairs given above can be represented as shown below.

corresponds to (DOTTED . PAIR)



corresponds to ((DOTTED . PAIR) . (POTTED . DAIR))

It is possible to create LISP records which have a graphic representation, but which have no corresponding symbolic representation as a dotted pair; for example,



6.2.3 LISP Lists

Atomic symbols, atomic numbers, and dotted pairs are all forms of symbolic representations of LISP data called S-expressions. There is an additional type of S-expression called a list. A LISP list can be defined recursively as

1) a sequence of one or more S-expressions enclosed in parentheses, or

2) the empty list, ()

The non-empty list has the form

(s1 s2 . . .)

where s1, s2, . . ., are S-expressions. Two or more list items may be separated by commas if desired.

Examples of LISP Lists:

```
(A 15 B)
(A,B,C)
(ONE)
((A . B) , (C . D) , (E . F))
(A (B C))
()
```

Lists are defined in terms of atoms and dotted pairs as follows. The empty list is equivalent to the atom 0 (zero)*. A list with one list item is equivalent to a dotted pair with the list item first and the atom 0 (zero)* second. A list with two or more list items is equivalent to a dotted pair with the first list item as the first

*In most LISP systems, the atomic symbol NIL is used instead of 0.

component and a list containing all list items except the first as the second component. Thus LISP lists may be defined in terms of dotted pairs (but not vice versa).

Examples:

() = 0

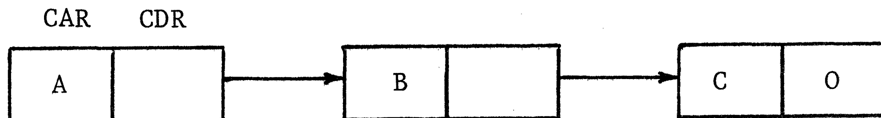
(A) = (A . 0)

(A B) = (A . (B . 0))

((C) D) = ((C . 0) . (D . 0))

((A B) (C) D) = ((A . (B . 0)) . ((C . 0) . (D . 0)))

LISP lists can be represented graphically in the same manner as dotted pairs; for example, (A B C) may be represented as



This kind of graphical representation is often useful for visualizing what occurs when S-expressions are manipulated by altering the contents of the CAR and CDR fields of the associated LISP records.

6.3 SYMBOL EXPRESSION

6.3.1 Definition

In ALGOL, an arithmetic expression may be considered as a set of rules which, when executed, generates a value which is a number. In GTL, a symbolic expression is a set of rules which produces a value which is a reference to a LISP record. Some of the components from which symbolic expressions are composed are described below.

6.3.2 Quoted S-expressions

The quoted S-expression is an S-expression enclosed in quotes which has as its value a LISP record, the symbolic representation of which is the S-expression. For example,

```
"(THIS IS A LIST)"
"ATOMICSYMBOL"
"(DOTTED . PAIR)"
"(3 5 7 11)"
```

6.3.3 Numbers and Arithmetic Expressions

Almost any class of arithmetic expression, including those composed of literal numbers, may be used in a symbol expression. Whenever an arithmetic expression is used where a LISP reference value is expected, its value is a reference to a LISP record which represents the number which is the value of the arithmetic expression. The arithmetic primary which cannot be used in this context is the string constant (of Burroughs Extended ALGOL, in which GTL is embedded), since it may be identical in form to a quoted atomic symbol representing the same string.

6.3.4 LISP Variables

Variables whose values are references to LISP records are declared and used in the same forms as REAL, INTEGER, and BOOLEAN variables in ALGOL. LISP variables are declared with the declarator SYMBOL; for example,

```
SYMBOL X, Y, Z
SYMBOL ARRAY SR[0:20]
```

A variable of type SYMBOL may be used in a symbol expression to produce its value, or, as in ALGOL, in the left part list of an assignment statement to change its value.

6.3.5 Assignment Statement

An assignment statement, besides being used as a statement, may be used in a symbol expression having as its value the value which is assigned to the variables in the left part list. An extension of the syntax of the LISP assignment statement is described in Subsection 6.7.

6.3.6 The Field Designators

The CDR field name may be applied to any symbol expression the value of which is a LISP record of type 1 or 3 (atomic symbol or dotted pair). The CAR field name may be applied to any symbol expression the value of which is a type 3 LISP record (dotted pair).

Examples

```
CAR("(A . B)")= A
CDR("(A . B)")= B
CAR("(A B C)")= A
CDR("(A B C)")= (B C)
```

Compositions of these field designators may be contracted to a form illustrated by the following examples:

```
CADR("(A B C)") = CAR(CDR("(A B C)")= B
CDAR("((A B) C)") = CDR(CAR("((A B)C)")) = (B)
CAAAR("(((A)))")= CAR(CAR(CAR("(((A)))")))= A
CADDR("(A B C)")= CAR(CDR(CDR("(A B C)")))= C
```

The length of a composite field designator may not exceed 13 characters.

6.3.7 Conditional Expression

The LISP 2 conditional expression has the same form as any ALGOL conditional expression. For example,

```
IF X = 1 THEN CAR(S) ELSE CADR(S)
```

6.3.8 LISP Function Designator

A LISP function designator is simply a call on a procedure which was declared with the type SYMBOL, its value being the reference to the LISP record defined by the SYMBOL procedure declaration. In GTL, the formal parameters of the procedure, if any, may be of type SYMBOL; the conventions of call-by-value and call-by-name of standard ALGOL also apply. For example, the declaration

```
SYMBOL PROCEDURE  ELN(N ,S); REAL N; SYMBOL S;
    BEGIN
       WHILE (N:=N - 1) > 0 DO S:=CDR(S);
       ELN:=CAR(S)
    END OF ELN
```

defines a procedure which has as its value the Nth element of the list S;

```
ELN(1, "(A B C D)")= A
ELN(3, "(A B C D)")= C
```

Note that the preceding definition is not intended to imply that procedures of any other type (or untyped) may not have formal parameters of type SYMBOL; see, for example, the procedure declaration given in paragraph 6.8.2.

6.4 LISP STANDARD FUNCTIONS

6.4.1 CONS

The LISP function CONS is a standard (or "intrinsic") function of two arguments, both symbolic expressions. The value of CONS is a freshly-created LISP record whose CAR field is set to the value of the first argument, and whose CDR field is set to the value of its second argument. For example,

```
CONS("A", "B") = (A . B)
CONS("A", CONS("LIST",0)) = (A LIST)
```

Note that the difference between CONS("A", "B") and "(A . B)" is that each time the latter is evaluated, its value is a reference to the same LISP record.

6.4.2 LIST

The LIST function is a standard function of one or more arguments, the value of which is a set of link LISP records the symbolic representation of which is a LISP list. For example,

```
LIST("THIS", "IS", "(A LIST)") = (THIS IS (A LIST))
```

The use of the word LIST in this context does not interfere with its use in the LIST declaration of Burroughs Extended ALGOL.

6.4.3 RANDOM

The standard function RANDOM is a function of one symbol expression argument the value of which should be a list. The value of RANDOM is one item chosen at random from the list.

Example:

```
RANDOM("(CHOOSE ONE OF THESE AT RANDOM)")
```

6.4.4 APPEND

The standard function APPEND is a function of two symbol expression arguments whose values should be lists. Its value is a copy of the first list with the CDR field of its last record modified to point to the second list. Its effect is illustrated by the following procedure declaration:

```
SYMBOL PROCEDURE APPEND(X, Y); VALUE X, Y; SYMBOL X, Y;
  APPEND:=IF NULL (X) THEN Y ELSE
    CONS(CAR(X), APPEND(CDR(X), Y))
```

(The NULL function is TRUE if its argument is the empty list - see paragraph 6.5.5).

Example:

```
APPEND("(A B C)","(D E F)") = (A B C D E F)
```

6.4.5 NCONC

The standard function NCONC is a function of two symbol expression arguments whose values should be lists. Its value is its first argument (if a non-empty list) with the CDR field of its last record altered to point to the value of NCONC's second argument. If the first argument is the empty list, then the value of NCONC is the value of its second argument. The effect of NCONC, illustrated by the procedure declaration given below, is similar to that of APPEND except that its first argument is not copied.

```
SYMBOL PROCEDURE NCONC(X,Y); VALUE X, Y; SYMBOL X, Y;
  IF NULL(X) THEN NCONC:=Y ELSE
  BEGIN
    NCONC:=X;
    WHILE NOT NULL(CDR(X)) DO X:=CDR(X);
    CDR(X):=Y;
  END
```

The last assignment statement shown above means that the CDR field of the record to which the value of X refers is changed to the value of Y. This construct is further explicated in Subsection 6.7.

6.4.6 SPACE and QMARK

The standard functions SPACE and QMARK are functions of no arguments whose values are atomic symbols which represent the blank and question mark characters, respectively. These functions are provided since these characters are not part of the syntax of S-expressions which may be read or quoted (see Subsection 6.10).

6.5 BOOLEAN STANDARD FUNCTIONS

6.5.1 ATOM

The Boolean standard function ATOM, when applied to a symbolic expression argument, yields a value of TRUE if the value of the symbolic expression is an atom, i.e., an atomic symbol or number, and is FALSE otherwise.

Examples:

```
ATOM("ATOMICSYMBOL") = TRUE
ATOM(X + 5) = TRUE
ATOM("(A . B)") = FALSE
```

6.5.2 ATSYM

The Boolean standard function ATSYM, when applied to a symbolic expression argument, yields a value of TRUE if the value of the symbolic expression is an atomic symbol, and is FALSE otherwise.

Examples:

```
ATSYM("ATOMICSYMBOL") = TRUE
ATSYM(125) = FALSE
ATSYM("(A . B)") = FALSE
```

6.5.3 NUMBERP

The Boolean standard function NUMBERP, when applied to a symbolic expression argument, yields a value of TRUE if the value of the symbolic expression is a reference to a LISP record which represents a number, and is FALSE otherwise.

Examples:

NUMBERP("ATOMICSYMBOL") = FALSE

NUMBERP(125) = TRUE

NUMBERP("(A . B)") = FALSE

6.5.4 ALF

The Boolean standard function ALF, when applied to a symbolic expression argument having an atomic symbol as its value, yields a value of TRUE if the atomic symbol is an identifier, and is FALSE otherwise. If the value of the symbolic expression argument is not an atomic symbol, the value of ALF is undefined.

Examples:

ALF("ATOMICSYMBOL") = TRUE

ALF(":") = FALSE

ALF("A") = TRUE

In other words, ALF distinquishes between atomic symbols of types 1 and 2 as defined in paragraph 6.12.1.

6.5.5 NULL

The Boolean standard function NULL, when applied to a symbolic expression argument, yields a value of TRUE if the value of the symbolic

expression is the atom 0 (zero), and is FALSE otherwise.

Examples:

NULL(0) = TRUE

NULL("()") = TRUE

NULL("(ONE)") = FALSE

NULL(CDR("(ONE)")) = TRUE

6.5.6 <u>MEMBER</u>

The Boolean standard function MEMBER, when applied to two symbolic expression arguments, yields a value of TRUE if the value of the first argument is a member of the list which is the value of the second argument.

Examples:

MEMBER("B", "(A B C)") = TRUE

MEMBER("X", "(A B C)") = FALSE

MEMBER("(C D)", "(A B (C D))") = TRUE

6.6 LISP RELATIONAL EXPRESSIONS

The definition of the relational expression of ALGOL is extended by the includion of the LISP relational expression which has the form:

<u>sv</u> EQ <u>se</u>

<u>sv</u> NEQL <u>se</u>

<u>sv</u> = <u>se</u>

<u>sv</u> ≠ <u>se</u>

where <u>sv</u> is a SYMBOL variable, LISP assignment statement, LISP function designator, or a field designator; EQ, NEQL, =, and ≠ are the LISP relational operators; and <u>se</u> is a symbol expression. The relational

expression containing EQ is TRUE if, and only if, the values of _sv_ and _se_ are references to the same LISP record. The relational expression containing = is TRUE if and only if the values of _sv_ and _se_ are references to LISP records (not necessarily the same) which represent the same S-expression. The relational expressions containing NEQL and ≠ are the negations of the relational expression containing EQ and =, respectively.

Note that, according to the definitions given above, a relational expression of the form

    CAR(S) = 15

is syntactically correct, whereas

    CAR(S) > 15

is not, since > is not a LISP relational operator. The above expression could be written correctly as

    15 < CAR(S)

in which case a run-time error would result if CAR(S) is not a number.

6.7 THE LISP ASSIGNMENT STATEMENT

The LISP assignment statement has the same form and operational meaning as an ordinary ALGOL assignment statement. In addition to SYMBOL variables, however, field designators (including the composite forms shown in paragraph 6.3.6) may be used in place of any variable in a left part list. For example, if X is a SYMBOL variable having the value (A B C), then

    CAR(X):="D"

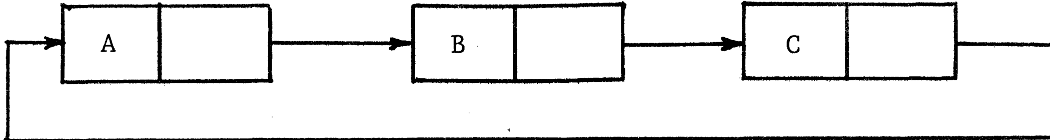
changes the CAR field of the LISP record referenced by the value of X to the atomic symbol D, so that, after the execution of this assignment statement, the value of X is (D B C).

Examples:

If the value of X before the statement is (A B C), then the following occurs:

assignment statement	value of X after assignment statement
CADR(X):="D"	(A D C)
CDR(X):="(D E)"	(A D E)
CAR(X):=CDR(X):="C"	(C . C)
CDDDR(X):=X	no S-expression representation

The result of the last assignment statement given above is a "circular" list and may be illustrated graphically as the following:



Note that NCONC (X, X) would have the same effect. It also changes X into a circular list. Circular lists may not be read or printed by the normal GTL Input-Output mechanism.

The assignment statement with the field designator in the left part list is the only means of changing the CDR and CAR fields of LISP records in LISP 2.

6.8 THE LISP ITERATIVE STATEMENT

6.8.1 The ON Statement

There are two types of LISP iterative statements which are similar in form to the FOR-statement of ALGOL. The first type has the form:

FOR s ON e DO st

where

s is a simple SYMBOL variable

e is a symbol expression, and

st is any statement

It is equivalent to the following compound statement:

```
BEGIN
s:=e;
WHILE NOT NULL(s) DO BEGIN st; s:=CDR(s) END
```

For example, consider the following procedure declaration:

```
SYMBOL PROCEDURE SUBST(X, Y, Z); VALUE X, Y, Z; SYMBOL X, Y, Z;
  BEGIN
    SYMBOL S;
    SUBST:=Z;
    FOR S ON Z DO IF X = CAR(S) THEN CAR(S):=Y;
  END
```

which has the effect of substituting Y for every occurrence of X in the list Z.

SUBST("A", "R", "(A B A C)") = (R B R C)

6.8.2 The IN Statement

The second type of LISP iterative statement has the following form:

FOR s IN e DO st

where s, e, and st have the same meanings as in paragraph 6.8.1 above. It is equivalent to the block:

```
BEGIN
  SYMBOL X;
  FOR X ON e DO BEGIN s:=CAR(X); st END
END
```

For example, a definition of the MEMBER standard function could be given by the following procedure declaration:

```
BOOLEAN PROCEDURE MEMBER(X,Y); VALUE X, Y; SYMBOL X, Y;
BEGIN LABEL EXIT; SYMBOL S;
  FOR S IN Y DO IF X = S THEN BEGIN MEMBER:=TRUE;
                       GO TO EXIT END;
  MEMBER:=FALSE;
EXIT: END OF MEMBER
```

6.8.3 The WHILE Part

Both forms of the iterative statement may contain a WHILE part in the form:

FOR s { IN / ON } e WHILE be DO st

where be is a Boolean expression. This means that the loop will continue until be is FALSE or until the empty list is reached.

6.9 EXTENSIONS OF ARITHMETIC EXPRESSIONS

6.9.1 Arithmetic Expression Syntax Extension

An arithmetic expression may contain any of the symbol expressions described in Subsection 6.3; the value of the symbolic expression should be a reference to a LISP record which represents a number. The value of the symbol expression in this context will be this number. For example, if X is a REAL variable and S is a SYMBOL variable,

```
X:=CDR(S) + X
```

is permitted if it is known that the value of CDR(S) is a number.

6.9.2 The LENGTH Function

There is a useful integer-valued standard function called LENGTH which has a symbolic expression as an argument. If the value of the symbolic expression is a list, then the value of LENGTH is the number of items on the list. If the value of the symbolic expression is an atomic symbol, then the value of LENGTH is the number of characters in the string represented by the atomic symbol; otherwise, the value of LENGTH is undefined. For example,

```
LENGTH("(A B)") = 2
LENGTH("((A B) (B C) (D E))") = 3
LENGTH("()") = 0
LENGTH("ONE") = 3
LENGTH("ATOMICSYMBOL") = 12
```

6.10 READING AND WRITING S-EXPRESSIONS

6.10.1 Output Functions

The value of any GTL variable or function designator, including LISP variables, procedures, and assignment statements--i.e., those declared with type SYMBOL, and LISP field designators--may be printed by the PRINT statement. The PRINT statement consists of the word PRINT followed by one or more "printable items". (See paragraphs 9.2.2 through 9.2.17 on the use of the PRINT statement.) For example, if S is a SYMBOL variable having the atomic symbol X as its value, and Y is a REAL variable, the value of which is 15, then

```
PRINT # THE VALUE OF # S # IS # Y
```

causes

```
THE VALUE OF X IS 15
```

to be printed on the output file. The output file is specified by the OUTPUT statement. (See Subsection 9.3.)

If S and R are SYMBOL variables with values (A B C) and (D E F), respectively, then

```
PRINT S,R
```

causes

```
(A B C) (D E F)
```

to be printed.

If S is a SYMBOL PROCEDURE which returns as its value the S-expression (THIS IS A LIST), then

```
PRINT CDR(S)
```

causes

```
(IS A LIST)
```

to be printed.

The user need not be concerned about printing items for which the character representation exceeds the size of a logical record of the output file. The output system automatically edits the output string so that it can be written on one or more logical records as needed.

If the GTL Output mechanism is used, then the following file and output string declarations are suggested:

1) for the line printer:

```
FILE OUT OUTFILE 16(2,15)
          and
STRING LINE (120)
```

2) for the remote terminal:

```
FILE OUT OUTFILE REMOTE (2,9)
          and
STRING LINE (72)
```

These declarations must appear in the outermost block of the program. With these declarations, the following output statement should be executed

before using PRINT:

OUTPUT(OUTFILE,LINE,<u>file length in characters</u>)

where <u>file length in characters</u> would be an unsigned integer specifying the file length in characters of the file. For the line printer and remote terminal, this would be 120 and 72, respectively. For remote terminals, the FILE REMOTE construct can be used (Subsection 9.6). (See Section IX for complete details on GTL Input-Output.)

6.10.2 Input Functions

The functions READ and READ1 may be used in symbol expressions for reading S-expressions from the input files. The function READ1 reads single atomic symbols and numbers only, and READ reads S-expressions. Dotted pairs and lists read by READ must be followed by $ (which serves to indicate the end of an S-expression in case of a parenthesis mismatch); the $ functions as a delimiter only and will not be read by a subsequent READ or READ1. For example, if

(NUMBER . 125)$

appears in the input string, then the value of READ will be the dotted pair (NUMBER . 125); if READ1 is executed six times (without an intervening READ), then the values of READ1 will be six atoms:

(

NUMBER

.

125

)

$

Although the spacing between items read from an input file by READ or READ1 is not important, an identifier or a digit string in a number cannot be broken across the boundaries of an input record (for example, the characters of an atomic symbol cannot begin on one card and continue on the following card); S-expressions read by READ can otherwise be spread across more than one input record. The value of READ or READ1 at end-of-file is the QMARK atomic symbol.

If the GTL Input mechanism is used, the following file and input string declarations are suggested for a card file:

```
FILE IN INFILE (2,10);
        and
STRING CARD (80)
```

For a remote terminal file, the following might be used:

```
FILE IN INFILE REMOTE (2,9);
        and
STRING CARD (72)
```

These declarations must appear in the outermost block of the program. With these declarations, the following INPUT statement should be executed before using READ or READ1:

INPUT(INFILE,CARD,file length in characters)

where file length in characters would be 80 and 72 for the card reader and remote terminal, respectively. (See Subsection 9.5.) If the remote terminal is being used for both Input and Output, then only one REMOTE file should be declared and the file identifier associated with that file should be used in both the INPUT and OUTPUT statements. If a listing of the input string from the card file is desired, then the additional declarations given in paragraph 9.8.4 can be used. For remote terminals

the FILE REMOTE construct can be used. (See Section IX for details of GTL Input-Output.) Warning: If the FILE REMOTE declaration (Subsection 9.6) is used in lieu of the above, then care should be exercised since a psuedo end-of-file is normally returned after every other READ or READ1; i.e., READ or READ1 will be equal to the QMARK (question mark) atomic symbol on every other read. (See paragraph 9.6.2, especially part 4.)

6.11 THE SYMBOL MONITOR

The values of SYMBOL variables and procedures can be monitored by the LISP monitor system. The variables and procedures to be monitored are specified by the declaration SYMBOL MONITOR followed by a list of SYMBOL variables or procedure identifiers. For example, if the variables X and SR and the procedure ELN (declared in paragraphs 6.3.4 and 6.3.8) are to be monitored, then the delcaration

```
SYMBOL MONITOR X, SR, ELN
```

should be used (after the declarations of these items). The name of the SYMBOL variable (plus values of subscripts if a subscripted variable) is printed with its value whenever it appears in the left part list of an assignment statement which is executed (a subscripted variable must be the leftmost variable for it to be monitored). Whenever a monitored SYMBOL procedure is evaluated, the procedure name, its arguments (if call by value and type REAL, INTEGER, BOOLEAN, or SYMBOL) and its value are printed. If a call on a SYMBOL procedure to be monitored appears before the SYMBOL MONITOR declaration (such as would normally occur with recursive procedure declarations), then that procedure call would not be monitored. This restriction can be circumvented by either declaring the procedure

FORWARD and making the actual procedure declaration after the SYMBOL MONITOR declaration or by making the SYMBOL MONITOR declaration inside the procedure declaration itself (which would cause only recursive calls to be monitored). The monitor file is specified by the output statement (paragraph 6.10.1). A monitor declaration is effective only in the block in which it appears.

For example, with the declarations

```
SYMBOL R;
SYMBOL PROCEDURE LISTOFATOMS(S);VALUE S; SYMBOL S; FORWARD;
SYMBOL MONITOR LISTOFATOMS, R;
SYMBOL PROCEDURE LISTOFATOMS(S); VALUE S; SYMBOL S;
    LISTOFATOMS:=IF NULL(S) THEN 0 ELSE
                 IF ATOM(CAR(S))THEN
                    CONS(CAR(S),LISTOFATOMS(CDR(S)))ELSE
                 NCONC(LISTOFATOMS(CAR(S)),LISTOFATOMS(CDR(S)))
```

the execution of the assignment statement

```
R:=LISTOFATOMS("(A ((B) C))")
```

will cause the following to be printed (names of variables and procedures are truncated to 7 characters when necessary):

```
CALL LISTOFA
(A ((B) C))
    CALL LISTOFA
    (((B) C))
         CALL LISTOFA
         ((B) C)
```

```
                    CALL LISTOFA
                    (B)
                         CALL LISTOFA
                         0
                         LISTOFA = 0
                    LISTOFA = (B)
                    CALL LISTOFA
                    (C)
                         CALL LISTOFA
                         0
                         LISTOFA = 0
                    LISTOFA = (C)
               LISTOFA = (B C)
               CALL LISTOFA
               0
               LISTOFA = 0
          LISTOFA = (B C )
        LISTOFA = (A B C)
    R = (A B C)
```

6.12 ATOMIC SYMBOLS

6.12.1 Types of Atomic Symbols

In the GTL LISP 2 system, there are three types of atomic symbols (classified by the kinds of strings the atomic symbol represents):

1) Identifier, which is an ordinary ALGOL identifier (i.e., a letter, which may be followed by one or more letters or digits),

2) Special character, which is any non-alphanumeric character in the B 5500 character set except the blank character and the question mark, and

3) Non-standard atomic symbol, which is any string of characters (which may include the blank character and question mark which is neither an identifier nor a special character.

No atomic symbol of any type may exceed 31 characters in length.

Quoted atomic symbols appearing in symbol expressions (paragraph 6.3.2) and atomic symbols read by READ1 may be any identifier or special character. The atomic symbols appearing in quoted dotted pairs or lists, or dotted pairs or lists read by READ, may be any identifier or special character except the following special characters:

(

)

.

,

"

$

These special characters cannot be recognized as atomic symbols in this context since they serve as delimiters of dotted pairs and lists (for S-expressions which are quoted or read by READ).

6.12.2 Nonstandard Atomic Symbols

Any nonstandard atomic symbol may be created by the MKATOM function which is described in paragraph 6.14.3. Also, the blank and question mark atomic symbols may be created by using the SPACE and QMARK functions, respectively, in symbolic expressions (paragraph 6.4.6).

6.12.3 Uniqueness of Atomic Symbols

Every atomic symbol created by the constructs described in this section (i.e., those appearing in quoted S-expressions, or read by READ or READ1) is unique. A single type 1 LISP record represents all occurrences of identical character strings in S-expressions which are read or quoted. This uniqueness has an important consequence: Information contained in a set of linked LISP records may be associated with the character string represented by an atomic symbol via the CDR field of the atomic symbol. For example, if one describes the syntax of simple arithmetic expressions by the following BNF equations,

```
<e> :: = <p>|<p><op><p>|<p><op><e>
<p> :: = <v>| (<e>)
<v> :: = A | B | C
<op> :: = + | - | ⊗ | /
```

their representation may be effected through the following assignment statements:

```
CDR("E"):="((P) (P OP P) ( P OP E))"
CDR("P"):=CONS(LIST("(", "E", ")"), "((V))")
CDR("V"):="((A) (B) (C))"
CDR("OP"):="((+) (-) (⊗) (/))"
```

so that, when using the procedure GEN, as defined below,

```
PROCEDURE GEN(X); VALUE X; SYMBOL X;
     IF NULL(CDR(X)) THEN PRIN X SPACE ELSE
     FOR X IN RANDOM(CDR(X)) DO GEN(X)
```

calls on the pair of statements

GEN ("E"); TERPRI

would cause to be printed randomly generated expressions which may have forms like those shown below:

A

A + B

A / (B - C)

(A + B ⊗ C) / A

6.13 THE LISP OBJECT LIST

6.13.1 The LISP Symbol Table

The uniqueness of atomic symbols described in paragraph 6.12.3 is assured through the use of a symbol table created and maintained by the GTL system. All single character atomic symbols are necessarily unique. The numeric value of a character is internally converted directly into the reference to the LISP record representing the character. However, all unique multi-character atomic symbols are contained on a list called the object list. Whenever a unique atomic symbol representing a string of characters is to be created, the object list is consulted first to determine whether or not an atomic symbol already exists which represents the string in question. If the atomic symbol already exists, a reference to this atomic symbol is returned. If the atomic symbol is not on the object list, it is created, and placed on the object list. The object list itself is actually not a single list but a collection of 125 lists. An arithmetic operation (MOD) is performed on a part of the string to be tested, yielding a value between

0 and 124. This value is then used as an index to an implicitly declared SYMBOL array, each element of which references a possibly empty list of atomic symbols. This procedure, called hashing, greatly reduces the amount of time required to determine the existence of an atomic symbol representing a multi-character identifier. The atomic symbols on an object list are not actually members of a list but are linked together through the CTR field of the atomic symbol. (The CTR field is described in Subsection 6.16.) In this context, 1 is used as an end of list indicator instead of 0. Therefore, every atomic symbol on the object list has a non-NULL CTR field.

6.13.2 The OBLIST Function

The OBLIST function may be used to access all of the multi-character atomic symbols on the LISP symbol table as described above. The OBLIST function is used in the form:

OBLIST(aexp)

where aexp represents an arithmetic expression, the value of which must be an integer between 0 and 124 (as explained above). For example, if S is a SYMBOL variable and X is a REAL variable, then the following statements could be used to print the contents of the object list:

```
FOR X:=0 STEP 1 UNTIL 124 DO
    IF NOT NULL(S:=OBLIST(X)) THEN
        DO PRIN S SPACE UNTIL S:=CTR(S) = 1;
TERPRI
```

6.13.3 The REMOB Statement

One or more atomic symbols may be removed from the object list by the REMOB function for the purpose of reclaiming storage used for atomic symbols and/or making an atomic symbol unrecognizable. The REMOB statement may be used in two forms:

REMOB

REMOB(se)

where se represents a symbol expression. The first form of the REMOB statement will remove the entire object list. The second form will remove from the object list the atomic symbol referenced by the value of the symbol expression se.

6.14 STRINGS AND ATOMIC SYMBOLS

6.14.1 Creation of Atomic Symbols

Any string of characters less than 32 characters in length can be converted into an atomic symbol, and vice versa. Conversion of an atomic symbol into the string of characters which it represents was discussed in paragraph 5.3.13. Every multi-character atomic symbol created by the GTL system is placed on the object list with the exception of those created by the GENSYM function (paragraph 6.14.4) and the asterisk forms of the MKATOM function (paragraph 6.14.3). The following two paragraphs describe functions of string expressions (See Section V) which are useful in the LISP portion of GTL. These functions are used implicitly by the READCON function (Section IX).

6.14.2 The ATCON Function

The ATCON function is a Boolean standard function which indicates whether or not a string is represented by an atomic symbol on the object list. The ATCON function is used in the form:

ATCON(se)

where se is a string expression. The value of the string expression should be less than 32 characters in length. The value of ATCON will be TRUE if there is an atomic symbol on the object list which represents the string, and FALSE otherwise. If ATCON is TRUE, then the atomic symbol which was found may be accessed by the standard variable INSYM (see paragraph 9.7.1). If the function ATCON is used by itself, without the string expression, the string contained in the string designator

INSTR(0, LENGTH(INSTR))

will be tested (see paragraphs 9.4.3, 9.4.4, and 9.7.1).

6.14.3 The MKATOM Function

The MKATOM function is used to create an atomic symbol from a string. The value of MKATOM is the atomic symbol which is created. The MKATOM function may be used in the following forms:

MKATOM(se)

MKATOM(se)*

MKATOM

MKATOM*

where se represents a string expression. The value of the string expression

must be less than 32 characters in length. The first two forms of the MKATOM function will return a reference to an atomic symbol which represents the string generated by the string expression _se_. The third and fourth forms of the MKATOM function will return a reference to an atomic symbol which represents the string contained in the string designator

INSTR(0,LENGTH(INSTR))

(see paragraphs 9.4.3, 9.4.4, and 9.7.1). The first and third forms of the MKATOM function will check the object list first to see if the atomic symbol already exists (see Subsection 6.13); if so, a reference to this atomic symbol is returned. If an atomic symbol does not already exist, then a new one is created and placed on the object list. If the second and fourth forms of the MKATOM function, the asterisk forms, are used, with multi-character atomic symbols, they will create a new atomic symbol which is not placed on the object list, regardless of whether or not there is an atomic symbol on the object list representing the string. The asterisk has no effect if the value of _se_ is a single-character atomic symbol.

6.14.4 The GENSYM Function

The GENSYM function is a SYMBOL standard function of no arguments. Each call on the GENSYM function will create a new atomic symbol which is not placed on the object list (and will not be recognized if read or quoted, or tested by ATCON). Atomic symbols created by GENSYM represent strings consisting of the letter "G" followed by a 3 digit number. For example, the first 3 calls on GENSYM will create the atomic symbols

G001

G002

G003

6.15 LISP REFERENCE VALUE TRANSFER FUNCTIONS

6.15.1 The CTSM Function

The CTSM function is a real-valued function used in the form:

CTSM(se)

where se represents a symbol expression. The value of CTSM is the contents of the word referenced by the value of se. For example, the value of the ID field of a LISP record (see paragraph 6.22.1) is given by

CTSM(se).[1:2]

In many cases, a REAL FIELD designator is more convenient than the CTSM function (see paragraph 7.2.2).

6.15.2 The SMTA Function

The SMTA (SyMbol To Arithmetic) function is a real-valued function used in the form:

SMTA(se)

where se represents a symbol expression. The value of SMTA is the arithmetic equivalent of the LISP reference value. For example, if

SMTA("THING") = 167

then 167 is the actual (relative) address of the LISP record which is the atomic symbol THING (see paragraph 6.22.1). Note that the relation

SMTA(se) $\leq$ 63

is true when the value of se is a single character atomic symbol or a single digit number (see paragraphs 6.22.2 and 6.22.3).

6.15.3 <u>The ATSM Function</u>

The ATSM (Arithmetic To SyMbol) function is a LISP reference-valued function used in the form

ATSM(*ae*)

where *ae* represents an arithmetic expression. The ATSM function converts the value of *ae* into the equivalent LISP reference value. For example, the following relations are always true:

SMTA(ATSM(*ae*)) = *ae*

ATSM(SMTA(*se*)) = *se*

where *ae* and *se* represent arithmetic and symbol expressions, respectively. Since any arithmetic expression may be used as an argument of ATSM, the user should be very careful to make certain that the value of ATSM is a legitimate LISP reference value. This is especially important when automatic reclamation is used, since the garbage collector will expect that all SYMBOL valued items will be an address of a legitimate LISP record.

The ATSM transfer function may also be used to modify the address of a LISP record when used in the following form

ATSM(*aexp*, *sexp*)

where *aexp* is an arithmetic expression, the value of which is added to the value of the SYMBOL expression *sexp*. The value of *aexp* must be a non-negative integer (see also paragraph 7.4.5). This expression is equivalent to

ATSM(*aexp* + SMTA(*sexp*))

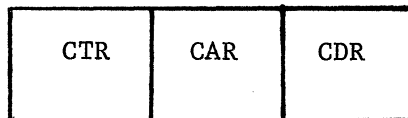
6.16 THE CTR FIELD

In addition to the CAR and CDR fields contained in LISP records which represent dotted pairs, there is an additional field, in GTL, called the CTR field. This additional field is provided since the internal machine representation of a LISP record, a B 5500 word, is large enough to accommodate an additional reference-valued field. The CTR field is not a standard LISP field, and it is not found in most LISP systems. There is also no corresponding symbolic representation of this field in LISP S-expressions. A CTR field designator may be used in the same forms as the CAR and CDR field designators, and may be used in composite field designators (see paragraph 6.3.6). For example,

CTDR(X) = CTR(CDR(X))

The CTR field is useful for a variety of applications such as predecessor links and for multi-linked list structures (see also Subsection 6.18).

A "dotted-pair" type LISP record with CTR field may be represented graphically by



6.17 PREFIX AND DOT OPERATORS

6.17.1 Prefix Field Designators

All of the LISP field designators described in paragraph 6.3.6 and the CTR field designator described above, may be used in a prefix form. The prefix form consists of the field name followed by a SYMBOL variable, either simple or subscripted. For example, if S is a SYMBOL variable,

then,

CADR S := CAR S

is equivalent to

CADR(S) := CAR(S)

6.17.2 Boolean Prefix Operators

If an argument of any of the Boolean standard functions ATOM, ATSYM, NUMBERP, ALF, or NULL (described in paragraphs 6.5.1 through 6.5.5) is a SYMBOL variable, then the Boolean function may be used as a prefix operator (without parentheses). For example, if S is a SYMBOL variable,

ATOM S

NULL S

ATSYM S

NUMBERP S

ALF S

are valid GTL constructs.

6.17.3 The Dot Operator

The definition of symbolic expression given in Subsection 6.3 is extended by the inclusion of the following construct:

se1 . se2

where se1 and se2 represent symbol expressions. It is equivalent to

CONS(se1, se2)

The period used in this context is called the dot operator. For example, the value of

```
"A" . "B"
```

is the dotted pair (A . B). Since sel and se2, as defined above, may also contain dot operators, symbol expressions may be parenthesized to limit the scope of a dot operator. When two or more dot operators appear in a symbol expression, the association is from the right; for example

```
"A" . "B" . "C" . "D"
```

is equivalent to

```
"A" . ("B" . ("C" . "D"))
```

the value of which is (A . (B . (C . D))). In the following additional examples, it is assumed that S is a SYMBOL variable with a value of (B C D).

symbol expressions	value
"A" . S	(A B C D)
CAR S . "(A D)"	(B A D)
"A" . "B" . "C" . 0	(A B C)
("A" . "B") . ("C" . "D") . 0	((A . B) (C . D))

When the symbol expression contains arithmetic operators, the dot operator has the lowest precedence; for example, if the value of the SYMBOL variable S is (3 4 5), then the value of

```
CAR S + CADR S . "(8 9)"
```

is the list (7 8 9).

6.18 PROPERTY LIST OPERATORS

6.18.1 The Property List

Most LISP systems use the CDR field of atomic symbols to reference linked lists of some kind containing attribute-value pairs. Such lists are called property lists of atomic symbols. Thus, with each atomic symbol there may be associated one or more attributes (atomic symbols) and each attribute of an atomic symbol has a corresponding value (an S-expression). In GTL, an economy of representation is achieved by using the CTR field for the attribute, the CAR field for the value, and the CDR field to reference the following attribute-value pairs (if any). The GTL property list operations are described in the following paragraphs.

6.18.2 ADDPROP

ADDPROP is a statement which is used to add an attribute-value pair to the property list of an atomic symbol. It is used in the form

ADDPROP(sym, attribute, value)

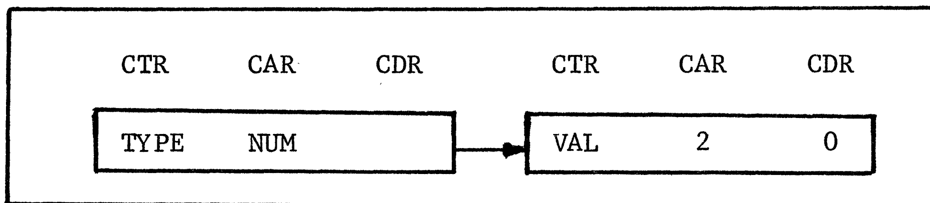
where sym, attribute, and value represent symbol expressions. The values of sym and attribute should be atomic symbols. The effect of ADDPROP is illustrated by the following procedure declaration:

```
PROCEDURE ADDPROP(S,A,V); VALUE S,A,V; SYMBOL S,A,V;
    IF ATSYM(S) THEN
        BEGIN CDR S := V. CDR S;
            CTDR S := A
        END
```

For example, if the CDR field of the atomic symbol "TWO" is initially empty, then the two statements

```
ADDPROP("TWO", "VAL", 2)
ADDPROP("TWO", "TYPE", "NUM")
```

have the effect of changing the CDR field of "TWO" as illustrated graphically below.



6.18.3 PROP

PROP is a symbol-valued function which may be used in any symbol expression. It is used in the form

PROP(<u>sym</u>, <u>attribute</u>)

where <u>sym</u> and <u>attribute</u> represent symbol expressions whose values should be atomic symbols. If the CDR field of <u>sym</u> is a property list containing <u>attribute</u>, then the value of PROP is the LISP record containing <u>attribute</u> in its CTR field and the value associated with the attribute in its CAR field. Otherwise, the value of PROP is 0. The effect of PROP is illustrated by the following procedure declaration:

```
SYMBOL PROCEDURE PROP(S, A); VALUE S, A; SYMBOL S, A;
    IF ATSYM S THEN
    IF NULL(S := CDR S) THEN PROP := 0 ELSE
    FOR S ON S DO
     IF CTR S EQ A THEN RETURN S
```

(Note that this declaration makes use of the RETURN statement described in Subsection 2.5. Referring to the example given in paragraph 6.18.2 above,

```
CAR(PROP("TWO", "TYPE")) = NUM
CAR(PROP("TWO","VAL")) = 2
PROP("TWO", "*") = 0
```

A list associated with a given attribute could be extended by a statement like that given below.

```
IF NULL(S := PROP(R,"*")) THEN ADDPROP(R, "*", L)
ELSE CAR S := APPEND (L, CAR S)
```

6.18.4 <u>REMPROP</u>

REMPROP is a statement which is used to remove an attribute-value pair from the property list of an atomic symbol. It is used in the form

REMPROP(<u>sym</u>, <u>attribute</u>)

where <u>sym</u> and <u>attribute</u> represent symbol expressions. The effect of REMPROP is illustrated by the following procedure declaration:

```
PROCEDURE REMPROP(S,A); VALUE S, A; SYMBOL S, A;
    IF ATSYM S THEN
        BEGIN SYMBOL R;
            WHILE NOT (NULL(R := CDR S) OR
                (CTR R)EQ A) DO S := R;
            CDR S := CDR R
        END
```

6.18.5 The Numeric Property Record

The property list of an atomic symbol may also contain a "numeric" property record which contains a CDR field but neither the CTR nor CAR fields. Instead of containing LISP reference values, the CTR and CAR fields are combined into a single field which can contain an unsigned integer. The length of this field is 29 bits (its value may lie between 0 and (2^{29}-1), inclusive). The numeric property record is added to and removed from property lists of atomic symbols by the statements

ADDPROP(_sym_, *, _aexp_)

REMPROP(_sym_, *)

where _sym_ represents a symbol expression and _aexp_ represents an arithmetic expression. The value of _sym_ should be an atomic symbol and the value of _aexp_ should be an unsigned integer. A number placed on a property list in this manner may be accessed by the arithmetic standard function NPROP used in the form

NPROP(_sym_)

where _sym_ has the same meaning as above. NPROP may be used in any arithmetic expression. For example, after the execution of

ADDPROP("VAL",*,215)

then

NPROP("VAL") = 215

6.18.6 Reference Property Records

References to records other than LISP records may be placed on property lists of atomic symbols in LISP records called "reference property records." Like the numeric property records described above, these records have neither CAR nor CTR fields. These records and the property list operations associated with them are described in Section 7.

6.19 THE SYMBOL DEFINE DECLARATION

6.19.1 The Standard Declaration

A SYMBOL DEFINE declaration is used to define an identifier which represents an S-expression. It has the same form as an ordinary DEFINE declaration of B 5500 Extended ALGOL except that the definition must be a quoted S-expression without a # at the end. For example,

```
SYMBOL DEFINE DF = "(A B C)"
```

In this case, every occurrence of DF in symbol expressions, including quoted S-expressions, is replaced by (A B C). Thus, with the declarations,

```
SYMBOL DEFINE A1 = "(A B C)",
              A2 = "(D E F)",
              A3 = "(G H I)",
              B1 = "(A1 A2 A3)"
```

every occurrence of B1 is replaced with the list

```
((A B C) (D E F) (G H I))
```

Each occurrence of the SYMBOL DEFINE identifier is replaced with the same set of records representing the quoted S-expression. If an ordinary DEFINE declaration were used; e.g.,

```
DEFINE D = "(A B C)"#
```

it would be replaced by a different set of records representing the same S-expression(except in the case of atomic symbols, which are unique). In addition, an ordinary defined identifier would not be replaced by its definition in a quoted S-expression. Also, a SYMBOL defined identifier may not be used in its own definition.

6.19.2 CDR Field Initialization

If the quoted S-expression appearing in a SYMBOL DEFINE declaration is a quoted atomic symbol, then the CDR field of the atomic symbol can be initialized at the time the declaration is made by the inclusion of a "field initialization part" in the SYMBOL DEFINE declaration. There are four forms of the "field initialization part," each of which must immediately follow the quoted atomic symbol.

The first form of the field initialization part consists of a colon followed by any quoted S-expression. The CDR field of the quoted atomic symbol will reference the records representing the quoted S-expression. For example, with

```
SYMBOL DEFINE DF = "ABC" : "(A B C)"
```

the following relations will be true:

```
DF = "ABC"
CDR(DF) = "(A B C)"
CDR("ABC") = "(A B C)"
```

The second form of the field initialization part is a colon followed by an unsigned integer enclosed in brackets. The CDR field of the quoted atomic symbol will reference a numeric property record representing the unsigned integer (see paragraph 6.18.5). For example, with the declaration

```
SYMBOL DEFINE A1 = "VAL" : [251]
```

the following relation will be true:

```
NPROP(A1) = 251
```

The third form of the field initialization part is a colon followed by a parenthesized list of attribute-value pairs and/or bracketed unsigned integers. Two or more list items are separated by commas, and each attribute-value pair consists of an atomic symbol followed by a colon followed by an S-expression. The CDR field of the atomic symbol is initialized to a property list (see paragraph 6.18.1) consisting of attribute-value pairs and/or numeric property records. For example,

```
SYMBOL DEFINE TW = "TWO" : (TYPE:NUM, VAL:2)
```

has the same effect as the two examples of ADDPROP statements given in paragraph 6.18.2. Also, with the declaration

```
SYMBOL DEFINE DV = "/" : (TYPE:OP, [125])
```

the following relations will be true:

```
CAR(PROP(DV,"TYPE")) = "OP"
NPROP(DV) = 125
```

The fourth form of the field initialization part consists of a colon followed by an unsigned integer. The CDR field of the atomic symbol will be initialized to the integer itself and not to a reference to a LISP record. The value of the integer must be less than 32768 and must not exceed 63 when automatic storage reclamation is used (see Subsection 6.20). The CDR field of the quoted symbol appearing in this type of SYMBOL DEFINE declaration must never be referenced in a symbol expression.

The CDR field of such an atomic symbol may be used in an arithmetic expression when the CTSM transfer function is used (see paragraph 6.15.1).

For example, with the declaration

```
SYMBOL DEFINE D523 = "START" : 523
```

the following relation will be true:

```
CTSM(D523).[33:15] = 523
```

6.19.3 The Asterisk Form

When a SYMBOL DEFINE declaration is used for the sole purpose of initializing the CDR field of an atomic symbol, the following form of the definition part may be used: the defined identifier and the = may be replaced with an asterisk. For example,

```
SYMBOL DEFINE * "THE" : "ARTICLE"
```

will initialize the CDR field of the atomic symbol THE to the atomic symbol ARTICLE. With this form of SYMBOL DEFINE declaration, the fourth form of the CDR field initialization part (as described above) is particularly useful for associating numbers with classes of key words; for example,

```
SYMBOL DEFINE * "SIN" : 1,
              * "COS" : 2,
              * "EXP" : 3,
              * "LN"  : 4
```

might be used in conjunction with the CASE statement:

```
CASE CTSM(S:=READ1).[33:15] OF
BEGIN
    PRINT #UNDEFINED OPERATION#;
    PRINT X:=SIN(X);
    PRINT X:=COS(X);
    PRINT X:=EXP(X);
    PRINT X:=LN(X);
END
```

6.20 STORAGE RECLAMATION

6.20.1 Automatic Versus Programmed Storage Reclamation

In GTL, the user is given a choice between automatic and programmed storage reclamation. When a relatively large amount of storage is used and when keeping track of discarded list structure is difficult or impossible, automatic storage reclamation should be used. On the other hand, if it is relatively easy for the programmer to keep track of the list structure which is to be discarded, then it would be more efficient to use the RECLAIM statement described below. Also, if the amount of storage used is relatively small, or if the amount of list structure in use does not decrease, then the user may elect to use no storage reclamation at all. In GTL, storage is allocated for LISP programs in 512 word blocks. Each time four of the blocks have been used (2048 words), the GTL system will check the available storage list, called the freelist, to see if any words have been reclaimed. No new blocks of storage will be allocated as long as there are a sufficient number of words remaining on the freelist. Words are linked into the freelist either automatically, by the automatic storage reclamation system, or programmatically by the RECLAIM function.

6.20.2 Automatic Storage Reclamation

When automatic storage reclamation is desired in a LISP program, the words SYMBOL RECLAIM, followed by a semicolon, must appear directly after the first BEGIN in the program; i.e.,

```
BEGIN SYMBOL RECLAIM;
```

This pseudo-declaration tells the compiler that the automatic storage reclamation is to be used. The internal function used to perform the storage reclamation is usually called the garbage collector. The garbage collector goes to work when a block of allocated storage is exhausted and the freelist is empty (see paragraph 6.20.1). The garbage collector can also be forced into action by the RECLAIM function described below. The GTL garbage collector uses an algorithm used by most other LISP systems: a marking phase followed by a collection phase. In the marking phase, every LISP record which can be accessed by a SYMBOL variable or through the CDR field of an atomic symbol on the object list is marked. In other words, all list structure in use by the program at the time the garbage collector is called is marked. In the collection phase, a linear scan of the blocks of storage allocated at that point is made, unmarking the LISP records which are marked, and reclaiming the initially unmarked records. The operation of the garbage collector can be monitored through various GTL system control parameters which are made available to users (see Appendix C). There are two restrictions which must be observed when automatic reclamation is used. Non-local jumps—i.e., jumps to labels outside a procedure or block—are not permitted, and the values of SYMBOL variables and procedures and the contents of all CAR and CDR fields must be legitimate LISP reference values (see paragraphs 6.19.3 and 6.15.3).

6.20.3 Programmed Storage Reclamation

When automatic storage reclamation is not used, LISP records to be discarded may be linked into the freelist RECLAIM statement of the form

RECLAIM(*se*)

where *se* represents a symbol expression the value of which should be a reference to the LISP record which is to be reclaimed. The RECLAIM function will reclaim single LISP records only. A collection of procedures for reclaiming lists and atomic symbols is given in Subsection 6.24. If automatic storage reclamation is used, the statement RECLAIM may be used to force the garbage collector to go into action. (Also see paragraph 7.4.4.)

6.21 AUTOMATIC STORAGE AND RETRIEVAL OF LISP LIST STRUCTURE

6.21.1 The LISP "Memory"

The GTL LISP system provides a mechanism by which all of the atomic symbols on the object list, and all of the list structure referenced by the CDR fields of these atomic symbols, can be dumped at some point in a program, and later loaded at another point in the same or another program. This is done by actually saving and retrieving the internal representation of the LISP records, rather than by attempting to read and write the symbolic representations of these records. If a program's "experiences" are encoded, for example, in property lists of atomic symbols, then these "experiences" could be saved and later recalled by the same program or by a different program, giving the program a "memory". The file upon which the LISP records are stored must be specified by the user, and must have the following specifications: the file must be declared in the outermost block of the program, each logical record in the file must be at least 512 words long, and the file should be large enough to contain 80 logical records.

For example,

```
FILE REMEM DISK SERIAL [20:4] (1,540,SAVE 10); COMMENT DISK;
```

or

```
FILE REMEM 2 (1,512,SAVE 10); COMMENT TAPE;
```

Two functions which operate on the file, RECALL and REMEMBER, are described below.

6.21.2 The REMEMBER Statement

The REMEMBER statement is used to store the contents of LISP records. It is used in the form

REMEMBER(_fileid_)

where _fileid_ is the name of the file described in paragraph 6.21.1 above. To save the pointers of various SYMBOL variables and SYMBOL arrays, as well as the LISP memory, the following extension of the REMEMBER statement may be used:

REMEMBER(_fileid_,*,_list_)

where _list_ is any explicit list of SYMBOL variables and SYMBOL arrays. The syntax of this _list_ is identical to an explicit ALGOL LIST used in an ALGOL WRITE statement. For example,

REMEMBER(_fileid_,*,L,L1,L2,FOR I:=1 STEP 1 UNTIL N DO S[I])

In fact, the REMEMBER statement may be considered identical to an ALGOL WRITE statement with an explicit LIST, with the additional attribute of writing out the LISP memory. The REMEMBER statement does not REWIND or

LOCK the file. Thus multiple REMEMBERs may be made to the same file. If the SYMBOL RECLAIM option is being used (see Subsection 6.20), then the garbage collector is called before the REMEMBER statement is executed. The garbage collector collects all LISP records which are not on or referenced through the object list, and places these records on the free list. Therefore, in this case no LISP records may be referenced after the REMEMBER statement is used.

6.21.3 The RECALL Statement

The RECALL statement will recall a LISP memory which was generated by a program in which a REMEMBER statement was executed. It is used in the form

RECALL(fileid)

where fileid is the name of the file described in paragraph 6.21.1 above. To retain the pointers saved during a REMEMBER statement (see paragraph 6.21.2) as well as the LISP memory, the following extension of the RECALL statement may be used:

RECALL(fileid,*,list)

where list is any explicit list of SYMBOL variables and SYMBOL arrays. The syntax of this list is identical to an explicit ALGOL LIST used in an ALGOL READ statement. For example,

RECALL(fileid,*,FOR I:=1 STEP 1 UNTIL 3 DO S[I],L)

The RECALL may be considered analogous to an ALGOL READ statement with an explicit list, with the additional attribute of reading in the LISP memory from the file specified. The RECALL statement does not REWIND the file.

Thus multiple RECALLs may be made from the same file. Since a RECALL statement initializes the LISP symbol table and all LISP records referenced through the CDR fields of atomic symbols in the symbol table, all references created by the compiler into the LISP memory may be invalid after executing a RECALL statement. To avoid this problem, every quoted S-expression appearing in the program must be a single character atom (a single character atomic symbol or a digit). If the SYMBOL RECLAIM option is used, then no operation which causes a LISP record to be generated may be performed before the execution of the RECALL statement. These operations include the creation of lists, dotted pairs, atomic numbers (other than single digits) and multi-character atomic symbols. Also every quoted S-expression appearing in the program must be a single character atom (a single-character atomic symbol, or a digit).

6.22 THE INTERNAL REPRESENTATION OF LISP RECORDS

6.22.1 LISP Reference Values

All LISP reference values in GTL are actually pointers or (relative) addresses of words in core memory. A maximum of 32768 words are available, addressed from 0 to 32767. These words are, in effect, elements of an array like that specified by the following declaration

```
ARRAY LINK[0:63,0:512]
```

If R represents a LISP reference value, then the contents of the word referenced by R would be

```
LINK[R.[33:6],R.[39:9]]
```

A field which is common to all types of LISP records is the ID field.

The ID field is a 2 bit field, the value of which indicates the type of LISP record. ID is not a GTL field name, but the contents of the ID field can be referenced indirectly (see paragraph 6.15.1). The locations of the ID, CTR, CAR, and CDR fields are specified by the following partial word field descriptions:

field name	partial word field description
ID	[1:2]
CTR	[3:15]
CAR	[18:15]
CDR	[33:15]

A description of the contents of these and other fields in LISP records is given below.

6.22.2 Atomic Symbols

A LISP record is identified as an atomic symbol by an ID field value of 2. The CTR field is used to link together atomic symbols which are on the object list (see Subsection 6.13), and should never be changed by the programmer. The CAR field of a single character will contain a 1, if a letter, or a 2 otherwise. The CAR field of a multi-character atomic symbol contains a pointer to a set of linked words containing the string of characters which the atomic symbol represents. A word representing a single character atomic symbol does not contain a reference to its symbolic representation; the address of the word will always be equal to the numeric value of the character. The CAR field of an atomic symbol should not be referenced.

6.22.3 Atomic Number

A LISP record is identified as an atomic number by an ID field value of 3. A number which is a single digit is uniquely represented by one word; the address of the word is the value of the digit. All other numbers are represented by two words.

6.22.4 Dotted Pairs

A LISP record which represents a dotted pair is identified by an ID field value of 0. All three of the fields, CTR, CAR, and CDR, may be referenced and changed by the programmer. Also, all three fields are considered to be valid LISP reference fields by the garbage collector.

6.22.5 Other Types of Records

Numeric property records and reference property records are identified by an ID field value of 1. Only the CDR field of these words is considered to be a valid LISP reference field.

6.23 LISP SYSTEM CONTROL PARAMETERS

The values of various control parameters used by the GTL system may be accessed by a standard function called CONVAL. The CONVAL function is used in the form

CONVAL(n)

where n represents an unsigned integer whose value designates the desired control parameter. Some of the values of n which may be used and the corresponding values of CONVAL(n) are listed in the table given below. (See also APPENDIX C.)

n	value of CONVAL(n)
0	a newly-generated random number between 0 and 1 (used by the LISP RANDOM function)
1	value of current random number produced by CONVAL(0)
5	total number of words collected by garbage collector
6	number of times garbage collector is called
7	time (in seconds) required by last call on garbage collector
8	arithmetic value of the address of the first word in the freelist (0 if empty)
9	first subscript of array described in paragraph 6.22.1
10	second subscript of the array described in paragraph 6.22.1
29	normally 0; will be set to 1 after REMEMBER is executed, meaning that no LISP operation may be performed that causes a new LISP record to be generated when using automatic garbage collection
30	initially 0; will be set to 1 after the first LISP record is created by the program; when set to 1, the RECALL statement cannot be used when using automatic garbage collection
31	number of atomic symbols created by GENSYM
36	current index of table of LISP reference values maintained by the garbage collector; it is initially 125
37	two less than the number of initial blocks of storage allocated before the garbage collector is called (see paragraph 6.20.1)

Three expressions involving CONVAL which might be useful to the GTL programmer are listed below.

expression	meaning
CONVAL(9) ⊗ 512 + CONVAL(10)	number of words in use by the GTL system
LENGTH(ATSM(CONVAL(8)))	length of freelist
ENTIER(CONVAL(0) ⊗ N)	random integer between 0 and N-1

The value of the first expression minus the second is the number of words in use by the program.

6.24 PROGRAMMED STORAGE RECLAMATION

The following set of procedures may be used to reclaim storage when the automatic storage system is not used.

```
PROCEDURE RECLAIMLIST(L); VALUE L; SYMBOL L;
     BEGIN SYMBOL S;
          IF NOT ATOM(L) THEN
          DO BEGIN S := CDR L;
                   RECLAIM(L)
             END UNTIL ATOM(L := S)
     END OF RECLAIMLIST

PROCEDURE RECLAIMATOM(L); VALUE L; SYMBOL L;
     BEGIN REAL R, N;
          IF SMTA(S) > 63 THEN
          IF R := CTSM(L) < 0 THEN    COMMENT   ATOM;
```

```
            IF R.[1:2] = 3 THEN BEGIN COMMENT NUMBER;
                                      RECLAIM(L);
                                      RECLAIM(ATSM(R))
                                END ELSE
                BEGIN COMMENT ATOMIC SYMBOL;
                    IF R.[3:15] ≠ 0 THEN REMOB(L);
                    RECLAIM(L);
                    N := (R := CTSM(L := ATSM(R.[18:15]))).[1:5];
                    WHILE N > 7 DO BEGIN
                                      RECLAIM(L);
                                      N := N - 4;
                                      R := CTSM(L := ATSM(R));
                                    END;
                    RECLAIM(L)
                END
        END OF RECLAIMATOM

PROCEDURE RECLAIMALL(S); VALUE S; SYMBOL S;
        BEGIN LABEL START; REAL R;
        START: IF R := CTSM(S) < 0 THEN RECLAIMATOM(S) ELSE
            BEGIN RECLAIM(S)
                IF R.[1:2] = 0 THEN BEGIN COMMENT DOTTED PAIR;
                                          RECLAIMALL(ATSM(R.[3:15]));
                                          RECLAIMALL(ATSM(R.[18:15]))
                                     END;
                S := ATSM(R);
                GO TO START
            END
        END OF RECLAIMALL
```

The procedure RECLAIMLIST will reclaim a dotted pair on the top level of a list; i.e., the records referenced by the CAR and CTR fields of the top level records will not be reclaimed.

The procedure RECLAIMATOM will reclaim atomic symbols and atomic numbers. If an atomic symbol to be reclaimed by RECLAIMATOM is on the object list, it will first be removed from the object list.

The procedure RECLAIMALL, which uses RECLAIMATOM, will reclaim atoms, lists and dotted pairs. If RECLAIMALL is applied to a list or dotted pair, it will reclaim everything in the list or dotted pair. If the user wants to reclaim everything except atomic symbols, then RECLAIMATOM(S) may be replaced by

```
BEGIN IF R.[1:2] = 3 THEN RECLAIMATOM(S) END
```

in the procedure RECLAIMALL.

Under no circumstances should RECLAIMLIST and RECLAIMALL be applied to circular lists. This would generate an infinite loop in the program.

6.25 LISP EXAMPLE PROGRAM

The following example LISP program is not intended to represent a practical program, but merely serves to illustrate some of the GTL LISP 2 constructs. The GTL Input-Output system, which is described in 6.10 and in Section IX, is also included in the example.

```
BEGIN COMMENT THE SYMBOL PROCEDURE LCS, DEFINED BELOW, FINDS THE
                LONGEST COMMON SEGMENT OF THE TWO LISTS L1 AND L2;
FILE IN INFILE (2,10);
FILE OUT PRINTER 16(2,15);
STRING LINE(120), CARD(80);
BOOLEAN PROCEDURE INPRO;
      BEGIN
      LABEL EOF,EXIT;
      READ(INFILE,10,CARD)[EOF];
      WRITE(PRINTER,10,CARD);
      GO TO EXIT;
EOF:  INPRO := TRUE;
EXIT: END OF INPRO;
SYMBOL L1,L2;
LABEL START;
      COMMENT
      COMSEGL FINDS THE LENGTH OF THE LONGEST INITIAL COMMON SEGMENT
      OF TWO LISTS, X AND Y;
INTEGER PROCEDURE CCMSEGL(X,Y);
      VALUE X,Y;
      SYMBOL X,Y;
      COMSEGL := IF NULL(X) OR NULL(Y) OR CAR(X) NEQ CAR(Y) THEN 0
            ELSE COMSEGL(CDR(X),CDR(Y)) + 1;
      COMMENT
      COMSEG FINDS THE LONGEST INITIAL COMMON SEGMENT OF TWO LISTS
      X AND Y;
SYMBOL PROCEDURE COMSEG(X,Y);
      VALUE X,Y;
      SYMBOL X,Y;
      COMSEG := IF NULL(X) OR NULL(Y) OR CAR(X) NEQ CAR(Y) THEN 0
             ELSE CONS(CAR(X),COMSEG(CDR(X),CDR(X)));
SYMBOL PROCEDURE LCS(L1,L2);
      VALUE L1,L2;
      SYMBOL L1,L2;
      BEGIN
      LABEL A;
      REAL K,N,LX,LY;
      SYMBOL X,Y,BEST;
      LX := LENGTH(L1);
      FOR X ON L1 WHILE LX GTR K DO
            BEGIN
            LY := LENGTH(L2);
            FOR Y ON L2 WHILE LY GTR K DO
                  BEGIN
                  N := COMSEGL(X,Y);
                  IF N LEQ K THEN GO TO A;
                  BEST := COMSEG(X,Y); K := N;
A:                LY := LY - 1;
                  END;
            END;
```

```
        LCS := BEST;
        END OF LCS;
        COMMENT
        START OF EXECUTABLE CODE;
        OUTPUT(PRINTER,LINE,120);
        INPUT(INPRO,CARD,80);
        PRINT #THE FOLLOWING IS A TEST OF THE LCS FUNCTION#;
START:IF L1:=READ EQ QMARK OR L1 EQ "STOP" THEN EXIT;
        L2 := READ;
        PRINT LCS(L1,L2);
        GO TO START;
END.
```

THE CARD INPUT TO THE PROGRAM IS AS FOLLOWS:

```
(A B C B C D E)$
(B C D A B C D E)$
STOP
```

THE OUTPUT AS LISTED ON THE PRINTER IS:

```
THE FOLLOWING IS A TEST OF THE LCS FUNCTION
(A B C B C D E)$
(B C D A B C D E)$
(B C D E)
STOP
```

VII. RECORD PROCESSING

7.1 INTRODUCTION

Among the facilities in GTL, there is a collection of interrelated systems for creating and manipulating complex data structures. One of these systems, the GTL version of LISP 2, is described in Section VI. The purpose of this section is to describe the GTL record processing system, which consists of two separate systems: a disk-storage-oriented system and a core-storage-oriented system. The disk system is designed for manipulating fixed length linked records on a random disk file. The core system is an extension of the GTL LISP system for variable length plex processing. (The term "plex", first used by D. T. Ross, refers to a node, or linked record, which contains a variety of data types. In this section the term "plex processing" will be used primarily to refer to the core-storage record processing system (Subsection 7.4)) . Both of these systems use constructs which are based on the record processing system described in Wirth and Hoare's "A Contribution to the Development of ALGOL".[9]

Familiarity with the GTL LISP system is required for understanding the core-oriented plex processing system.

The remainder of this section is divided into three subsections: a description of the constructs common to both record processing systems (Subsection 7.2), and complete definitions of the disk and core systems (Subsections 7.3 and 7.4).

7.2 BASIC CONCEPTS OF GTL RECORD PROCESSING

7.2.1. Reference Expressions

A reference expression is simply an expression whose value is a reference to, or address of, a record (see Section 6.7, "Reference Expressions", p. 426, Reference 9). In GTL, reference expressions include the following:

1) reference variable,
2) reference function designator,
3) reference-valued field designator,
4) reference assignment statement,
5) conditional reference expression,
6) record designator,
7) null reference, and
8) parenthesized reference expression

A reference variable is a reference-valued simple variable or array element. In GTL, it is declared with a special class of declarators called record class identifiers (p. 423, Reference 9). As the name implies, a reference variable may only reference records contained in the class of records associated with the record class identifier. Reference variables are declared in the same form as variable declarations of type REAL; for example,

```
rci A, B, C
rci ARRAY RCA[0:99]
```

where rci represents a record class identifier. Record class identifiers are discussed further in Subsections 7.3 and 7.4.

Reference-valued procedures and formal paramenters (both name and value) are declared in the same manner:

rci PROCEDURE P(X,Y); VALUE X; rci X,Y; etc.

A reference-valued field designator is a construct which refers to the value of a particular field within a record. The type of field is determined by the declarator used to declare the field. Field designators are discussed in paragraphs 7.2.2 and 7.2.4, below.

A reference assignment statement has the same form and operational meaning as an ordinary REAL-valued assignment statement. All of the variables, function designators, and field designators appearing in a reference-valued assignment statement must be of the same type; i.e., they must have been declared with the same record class identifier.

Conditional reference expressions have the same form and operational meaning as other types of conditional expressions:

IF bexp THEN rexp1 ELSE rexp2

Where bexp represents a Boolean expression, and rexp1 and rexp2 represent reference expressions. If the value of the Boolean expression is TRUE, then the value of the conditional expression is the value of rexp1; otherwise, its value is the value or rexp2. Of course, rexp1 and rexp2 must have the same reference type; i.e., they must both be associated with the same record class.

Record designator is the name given to the construct which is used to generate new records in a given record class. This construct is described in Subsection 7.3 and 7.4.

The null reference is represented by the word NIL, and is used to indicate the absence of a reference to a record. It may be used, for example, to indicate the end of a list of linked records. (Internally, the value of NIL is zero - the zeroth record is never accessed.) NIL is the only reference expression which is associated with all record classes.

7.2.2. Field Designators

As mentioned above, the field designator is a construct used to access the value of a field within a record. It has the form

<u>fieldid</u>(<u>rexp</u>)

where <u>fieldid</u> represents a field identifier and <u>rexp</u> represents a reference expression. The type of the field and its relative location within the record referenced by <u>rexp</u> is determined by a field declaration, which is described below. The programmer should be careful to ensure that the value of the reference expression is never the null reference.

A field designator of any type may take the place of a variable of the same type in the left-part list of an assignment statement. For example,

```
AGE(JACK):=28
```

where AGE is a REAL-type field identifier and JACK is a reference variable. In addition, a string field designator may take the place of a string variable in a string designator; for example,

```
STRING CARD (80);
STRING FIELD CARDF [0:80];
rci X
   .
   .
   .
CARDF(X)(0,72):=CARD(0,72);
CARD(72,8):=CARDF(X)(72,8)
```

where rci represents a record class identifier.

7.2.3. The Reference Assignment Statement

The reference assignment statement (when used as a statement) has the same form and is subject to the same restrictions as the reference-valued assignment statement described in paragraph 7.2.1. For example,

```
X:=NEXT(X);
NEXT(X):=NIL
```

where X is a reference variable and NEXT is a field identifier, both of the same type(i.e., both declared by the same record class identifier).

7.2.4. The Field Declaration

The field declaration is used to declare the type of a field identifier and its relative location within a record. It has the following forms

type FIELD fieldid (loc) [skip:length]

type FIELD fieldid (loc)

where type represents a declarator indicating the type of field, fieldid represents the field identifier being declared, and loc, skip and length represent unsigned integers. The relative position of the field within a record is given by loc, which may range in value from 0 to 127. A loc of 0 refers to the first word, 1 to the second word, etc. For a non-STRING field, skip is the number of bits to be skipped from the beginning of the word and length is the length of the field in bits, so that this part of the field declaration has the same meaning as the field description of the partial word designator of Extended ALGOL (paragraph 3-10, Reference 4). If the field identifier is to refer to the entire word, the partial word part of the declaration must be omitted, as a partial word part of [0:48] is not permitted. In the

case of a STRING field, _skip_ is the number of characters to be skipped from the beginning of the word (from 0 to 7), and _length_ is the length of the field in characters; _skip_ and _length_ must be included in all STRING field declarations.

The permitted field types are REAL, INTEGER, ALPHA, BOOLEAN, STRING and the record class identifiers (which includes SYMBOL).

Examples:

REAL FIELD AGE (0) [41:7]

SYMBOL FIELD SYMF (3)

STRING FIELD NAME (4) [0:32]

rci FIELD NEXT (2) [33:15]

where _rci_ represents a record class identifier, a reference-type field. SYMBOL and other reference-type fields must be at least 15 bits long. The GTL compiler makes no distinction between REAL and INTEGER FIELDs; a full word INTEGER field may be assigned a REAL value.

Several simplifications of the field declaration are permitted: a _loc_ part of 0 may be omitted; if the _type_ part is omitted, a REAL field is implied; and a collection of field declarations of the same type may be combined into one declaration. For example,

Sample	Equivalent
STRING FIELD SF [0:64]	STRING FIELD SF (0) [0:64]
FIELD RF	REAL FIELD RF (0)
SYMBOL FIELD CARF[18:15],	SYMBOL FIELD CARF(0)[18:15]
CDRF[33:15]	SYMBOL FIELD CDRF(0)[33:15]

7.2.5. Indexed Fields

Another form of field designator is the indexed field designator, with the form

fieldid[index](rexp)

where rexp and fieldid represent a record expression and field identifier, respectively, and index represents an arithmetic expression, the value of which designates the relative location of the field in the record referenced by rexp. The value of index must be within the bounds specified in the indexed field declaration, which has the two forms

type FIELD fieldid (n, m) [skip:length]

type FIELD fieldid (n, m)

where type, skip and length have the same meanings as above (except an indexed STRING field is not allowed). The constants n and m specify the first and last words in the record which may be referenced by the indexed field designator. For example,

REAL FIELD RFX (0,9)

may be used to reference the first 10 words of a record. For example, the sum of the first 10 words of the record referenced by the reference variable X may be computed as follows:

```
FOR I:=0 STEP 1 UNTIL 9 DO SUM:=RFX[I](X) + SUM
```

7.3. THE DISK SYSTEM

7.3.1. The Record Class Declaration

A GTL program may contain up to 31 record class identifiers associated with linked-record random disk files. A record class identifier is declared by a record class declaration (Section 5.4, "Record Class Declarations", p. 423, Reference 9); it has the form

RECORD rci fileid (fieldlist)

where rci represents the record class identifier, fileid is the name of the random disk file which is to contain records of the rci class, and fieldlist is a list of one or more field identifiers. The size of a logical record of the file fileid must be large enough to accommodate all of the fields in the fieldlist. The field identifiers in the fieldlist may be declared either before or after the record class declaration, except for the rci-type fields which must be declared afterwards. The compiler allows the specification of overlapping fields in the record class declaration. In general, if the first character of a word is part of a STRING field, then that word should not also contain a non-STRING field; to do otherwise may result in a FLAG BIT error termination.

Two or more rci's may be associated with a given disk file, and a given field may be contained in two or more record classes; for example,

```
RECORD DEALER RANFILE (NAME, ORDER, NEXT);
RECORD STOCK RANFILE (STOCKNO, PRICE, QUANTITY, DATE, NEXT,
                      NXT);
STRING FIELD NAME (1) [0:32];
STOCK FIELD ORDER (5), NXT (5);
DEALER FIELD NEXT;
REAL FIELD STOCKNO (1),
           PRICE (2),
           QUANTITY (3),
           DATE (4)
```

The following is an example of the constructs described in Subsection 7.2 using the declarations given above, and

```
DEALER DLR;
STOCK STK;
REAL SUM
    .
    .
    .
WHILE DLR ≠ NIL DO
  BEGIN
  STK:=ORDER(DLR);
  SUM:=0;
  WHILE STK ≠ NIL DO
     BEGIN
     SUM:=PRICE(STK) ⊗ QUANTITY(STK) + SUM;
     STK:=NXT(STK)
     END;
PRINT NAME(DLR) SKIP(40) SUM;
DLR:=NEXT(DLR)
END
```

The file fileid in a record class declaration must be declared by a special random disk file declaration, which is described below.

7.3.2. The RECORD File Declaration

The RECORD file declaration has the same general form as ordinary random disk file declarations (paragraph 9-39, Reference 4), with the following exceptions:

1) "FILE" is replaced by "RECORD FILE",

2) the disk access technique ("RANDOM") is replaced by the disk type ("LOCAL", "NEW", or "OLD"), optionally followed by a constant, adr, and

3) the logical record size must be a constant.

A disk type of LOCAL means that the file is (a non-SAVE file) to be created by the program in which it appears and will not exist after the execution of the program; a non-LOCAL disk type indicates a new file to be created (NEW), or a previously created file (OLD). The optional constant, adr, indicates the (relative) address of the first record to be created by the record processing system (for a LOCAL or NEW file); if it does not appear, the starting address will be one. This allows the programmer to use the disk records with smaller addresses for other purposes (such as storing the heads of lists of linked records in non-LOCAL files).

Examples:

```
RECORD FILE DISC DISK LOCAL [20:300] (1,10,30)
SAVE RECORD FILE NEWF DISK NEW 2 [5:300] (1,15,30,SAVE 30)
RECORD FILE RANFILE DISK OLD "DEALERS" (1,10,30)
```

7.3.3. The Record Designator

The record designator is the constuct used to generate records. It has the two forms

rci(expression list)

rci

were rci is a record class identifier, and expression list is a list of expressions corresponding in type and position to the fields given in the record class declaration (p.426, Reference 9). If the field

is an indexed field, the corresponding expression should be a list of expressions (corresponding in type to the type of field) enclosed in brackets. If any of the fields in the record are not to be assigned a value in the record designator, an asterisk may replace the corresponding expression. If the <u>rci</u> is given without the <u>expression list</u>, a record is generated with all of its words set to zero. This means that a REAL field is set to zero, a BOOLEAN field is set to FALSE, a STRING field is set to all zero characters and reference fields are set to the null reference, NIL.

<u>Examples</u>:

```
RECORD PARTNO DF (STF,TYPE,NBR);
STOCK FIELD STF;
REAL FIELD TYPE (1,9)[18:15],NBR(1,9)[33:15];
PARTNO X
        .
        .
        .
X:=PARTNO(*,*,[23,24,25,*,*,*,56,57,58]);
DLR:=DEALER(CARD(10,32),*,DLR);
ORDER(DLR:=DEALER) := STK := STOCK
```

7.3.4. <u>Record Relational Expressions</u>

In order to compare two addresses of records of the same type, the following relation expression may be used in any Boolean expression:

<u>recvar</u> <u>relop</u> <u>rexp</u>

where <u>recvar</u> is a reference variable, <u>relop</u> is one of the relational operators or their mnemonics, and <u>rexp</u> is a reference expression. For example, given the record class identifier DEALER,

DEALER DLR,X

⋮

DLR = X

DLR ≠ NIL

DLR = NIL OR X = NIL

are Boolean expressions.

7.3.5. Transfer Functions

On occasion, it is convenient to be able to treat a reference value as a number, and vice versa. This can be accomplished with the two type transfer functions:

REAL(recvar)

rci(aexp)

The first transforms the value of the reference variable recvar into an arithmetic primary; the second transforms the value of the arithmetic expression aexp into a reference value associated with the record class of the record class identifier rci. These transfer functions should be used with caution since they allow errors which would otherwise be prevented by syntactic restrictions.

7.3.6. Storage Reclamation

Disk records may be reclaimed by the RECLAIM statement:

RECLAIM(recvar)

where recvar is a reference variable. It is the programmer's responsibility to ensure that the value of reference variable is not the null reference and that the record to be reclaimed does not remain a member of some active list, or is, in any other way, referenced at some later time. Whenever a record is reclaimed, it is placed on a list of records

called the freelist. When a new record is to be generated and the free-list is not empty, that record is obtained from the freelist. A separate freelist is maintained for each RECORD file in the program; two or more record classes associated with a given RECORD file use this freelist in common.

7.3.7. Saving and Restoring Heads of Master Lists in Non-LOCAL Files

The heads of master lists of linked records contained in a non-LOCAL RECORD file must be saved at the end of a program, and must be restored at the beginning of the program if the RECORD file is of type OLD. In addition, two other parameters associated with the RECORD file must be saved and restored; these are the head of the freelist (see above) and the location of the next available record. The values of these parameters are accessed by the constructs

FREELIST(fileid)

NEXTAVL(fileid)

where fileid is the name of the RECORD file. These two constructs may be used in the left-part list of assignment statements and in arithmetic expressions as if they were REAL variables.

Normally, the heads of master lists, and the freelist and next available record, are written onto, and read from, the first record in the file (with disk address zero). This can be done without interference to the remainder of the system, since the record with disk address 0 corresponds to the null reference, which is never accessed. Simple reference variables, and the FREELIST and NEXTAVL constructs, may be included in the lists of READ and WRITE statements as if they were ordinary simple variables. For example, referring to the declarations given in paragraphs 7.3.1 and 7.3.2

```
DEALER DH; STOCK SH;
LIST SAVELIST (NEXTAVL(RANFILE),FREELIST(RANFILE),DH,SH)
      .
      .
      .
READ(RANFILE[0],*,SAVELIST)
      .
      .
      .
WRITE(RANFILE[0],*,SAVELIST)
```

The internal value of NEXTAVL(fileid) is actually the disk address of the next available record minus one. The programmer should keep this in mind if he attempts to use the NEXTAVL construct for any purpose other than saving and restoring this parameter on a non-LOCAL file (e.g., the file might be used as a stack instead of using the RECLAIM statement). Notice also that it allows the programmer to use a simple method of combining the file creation program and the file manipulation program into one program. For example, referring to the example given above, if RANFILE[0] is initially cleared to zero, then, the first time the program is executed, the parameters in the list SAVELIST will be set to zero; i.e., the heads of the master lists DH and SH, and the freelist, will be set to the null reference, and the first record generated by the program will have a disk address of one. The file itself could be created by the following program:

```
BEGIN
FILE RANFILE DISK RANDOM [20:300] "DEALERS" (1,10,30, SAVE 90);
WRITE(RANFILE[0]);
LOCK(RANFILE);
END.
```

A RECORD disk file created by one program may be updated in both form and content by associating additional record class identifiers with the file.

7.3.8. Printing Reference Values

If a reference-valued variable or field designator appears in a PRIN or PRINT statement, then the associated record class identifier followed by the actual value of the disk address will be printed (see paragraph 9.2.12. of Section IX).

7.4. THE CORE STORAGE PLEX PROCESSING SYSTEM

7.4.1. The Record Class Identifier

Since the core system is an extension of the GTL version of LISP, the record class identifier in this case will be the LISP 2 declarator SYMBOL. SYMBOL reference expressions are the SYMBOL expressions defined in Section VI.

7.4.2. Field Designators

The LISP system contains the predefined field identifiers CAR, CDR, and CTR, and their composite forms (e.g., CADR). In addition, programmer defined fields may be defined as described in Subsection 7.2. In the latter case, the SYMBOL expression to which the field identifier may be applied is restricted to the following: a SYMBOL variable, a SYMBOL standard function (e.g., APPEND, NCONC, etc), the transfer function ATSM, the SYMBOL assignment statement, and the SYMBOL-valued field designator.

7.4.3. Record Designator

The LISP record designator is the function CONS, which is normally used to generate the one word record containing the fields CAR and CDR. Another form of the CONS function used with the plex processing system is

CONS[field-expression pair list]

where field-expression pair list is a list of one or more field-expression pairs having the form

fieldid : expression

where fieldid represents a field identifier, and expression represents an expression whose type corresponds to the type of the field identifier. For example, given the declarations

SYMBOL X, Y

SYMBOL FIELD CARF [18:15],

CDRF [33:15]

the following two expressions are equivalent:

CONS[CARF: X, CDRF: Y]

CONS(X, Y)

Since CONS can create only one word at a time, a multi-word record is created by successive CONSes (assuming that the freelist is empty so that successive CONSes would produce consecutively-addressed one-word records). In addition, certain restrictions must be placed on the field in this form of record designator:

1) STRING fields must not be extended beyond a word boundary; in no case may the length of a string field be longer than 8 characters,
2) only the first field identifier in a series of field expression pairs may refer to an entire word,
3) indexed fields are not allowed, and
4) all of the fields in the field-expression list must refer to the same relative word location (within a multi-word record).

Any portion of the word which is not initialized by a field-expression pair is set to zero. In addition to the two forms of the CONS record

designator described above, the word CONS, used by itself, will create a one word record which is initialized to zero.

7.4.4. The SYMBOL PLEX Option

Since the user-defined field designator allows the specification of the contents and meaning of arbitrary fields, the GTL automatic storage reclamation system cannot be used. Another form of storage reclamation available is the RECLAIM statement described in Section 6. This form of storage reclamation is generally to be avoided however, since, as mentioned in paragraph 7.4.3. a non-empty freelist would make the creation of records consisting of consecutive words difficult or impossible.

Another option available for the plex processing system is the SYMBOL PLEX option which is specified at the beginning of the outermost block of the program by the pseudo-declaration "SYMBOL PLEX;":

```
BEGIN SYMBOL PLEX;
```

When this option is used, the value of the address of the next available word (minus one) may be accessed by the construct

```
NEXTAVL(SYMBOL)
```

This construct may be used in the left-part list of an assignment statement or in an arithmetic expression as if it were a REAL variable. This feature allows the programmer to use the entire block of words available for LISP records as a stack. (The variable NEXTINFO plays a similar role with respect to the INFO array in the B 5500 ALGOL compiler). The SYMBOL PLEX option was designed to be used with the GTL translator writing system described in Section VIII. A sample program using the SYMBOL PLEX option is given in Subsection 8.8.

With the SYMBOL PLEX option, the RECLAIM statement simply has the effect of resetting the NEXTAVL parameter to the value of its argument.

The user should keep in mind that, before resetting NEXTAVL(SYMBOL) to its previous value, any multi-character atomic symbols created since its value was first saved will be linked into the object list. (See Section 6.) Thus, these atomic symbols must be removed from the object list by the REMOB statement before the words occupied by the atomic symbols can be re-used; if this is not done the GTL symbol table mechanism will not work. In the sample program given in Subsection 8.8, a list of newly created atomic symbols is maintained for this purpose.

7.4.5. The ATSM Transfer Function

The Arithmetic To SyMbol transfer function, ATSM (paragraph 6.15.3) may be used to modify the address of a LISP record when used in the following form

ATSM(*aexp*, *sexp*)

where *aexp* is an arithmetic expression, the value of which is added to the value of the SYMBOL expression *sexp*. The value of *aexp* must be a non-negative integer. The following example illustrates a method of printing the contents of a list of variable length records, each record containing a sequence of whole word numbers. The first word is the length of the remainder of the record.

```
SYMBOL R, S;
REAL I;
REAL FIELD WHOLE
      .
      .
      .
```

```
FOR S IN R DO
    BEGIN
    N:=WHOLE(S);
    FOR I:=1 STEP 1 UNTIL N DO
        PRIN WHOLE (S:=ATSM(1,S)) SPACE;
    TERPRI
    END
```

It is assumed that the value of R is the list in question.

7.4.6. The RECALL and REMEMBER Statements

The RECALL and REMEMBER statements, as described in Subsection 6.21 may be used with the SYMBOL PLEX option. When the SYMBOL PLEX option is used, there are no restrictions on the use of these statements; they may be used at any point in the program as often as desired. This feature might be used, for example, in the implementation of a self-extending syntax-directed translator. The value of NEXTAVL(SYMBOL) must be set before a REMEMBER to a point above the last element of the linked list to be stored.

In general, a LISP memory file created by one program will not be compatible with another program, since the addresses of quoted atomic symbols (created at compile time) will almost always be different, unless they are all single-character atomic symbols.

7.5 RECOMMENDED PRACTICES

If the RECLAIM statement is used to reclaim records then the following program should be used to create the record file:

```
BEGIN
FILE FILENAME DISK [20:300](1,10,30,SAVE 90);
ARRAY A[0:9];
INTEGER I;
FOR I := 0 STEP 1 UNTIL 5999 DO WRITE (FILENAME,10,A[*]);
LOCK (FILENAME);
END.
```

If the programmer fails to use this type program and has reused all reclaimed records and is creating a new record, he will have an error termination. This is due to the way random files are handled by the MCP and not by GTL.

7.6 EXAMPLE PROGRAM

The program listed on the following pages was designed to illustrate the use of the disk-storage-oriented record processing system. The program maintains a data base of students and possible courses they might take. The data base can be updated and maintained from a terminal. The user may inquire into the status of students in regard to required courses taken, or needed to be taken, grade average, which students took a particular course, etc. A simple example of the program's operation is given at the end of the listing.

Attention should be given to the DELETESTUDENTS procedure which illustrates not only how to reclaim a record, but also how to reclaim all records to which only the reclaimed record points.

This program should not be construed to represent a practical application or to represent data base construction. The program merely illustrates the creation and deletion of records in record processing.

In order to understand the operation of the program, the programmer should be familiar with GTL string processing (Section V) and the Input-Output system (Section IX). Knowledge of list processing (Section VI) is helpful in understanding record class identifiers.

```
BEGIN
COMMENT
      THIS PROGRAM MAINTAINS 3 TYPES OF RECORDS IN ONE FILE.

      THE FIRST RECORD, STUDENT, CONTAINS A STUDENT"S NAME, ADDRESS,
      SOCIAL SECURITY NUMBER, AND COLLEGE DEGREE, PLUS
      A FIELD WHICH POINTS TO A RECORD, COURSELISTING, WHICH
      RECORDS THE HISTORY OF COURSES TAKEN BY THE STUDENT.  THE OTHER
      POINTER POINTS TO THE NEXT STUDENT"S RECORD.  IF THIS POINTER
      IS NULL THEN THERE ARE NO MORE STUDENT RECORDS.  THIS
      INDICATES END OF THE LIST.  THE STUDENT POINTER, SP, POINTS TO
      THE  FIRST RECORD IN THE LINKED LIST OF STUDENT RECORDS.

      THE COURSEDESCRIP RECORD CONTAINS A DESCRIPTION OF EVERY POSSIBLE
      COURSE THE STUDENT MAY TAKE.  THIS DESCRIPTION INCLUDES
      THE DEPARTMENT WHICH OFFERS THE COURSE, THE COURSE NUMBER, WHETHER
      THE COURSE IS REQUIRED OR NOT, AND THE TITLE DESCRIPTION OF THE
      COURSE.  CDP POINTS TO THE FIRST RECORD OF THIS LINKED LIST.

      THE THIRD RECORD, COURSELISTING, CONTAINS 2 FIELDS.  THE FIRST
      FIELD CONTAINS 3 ITEMS: THE QUARTER THE COURSE WAS TAKEN, A
      POINTER TO THE COURSEDESCRIP RECORD, AND THE GRADE RECEIVED.  THIS
      FIELD OCCURS 9 TIMES WITHIN THE RECORD, WHERE EACH FIELD OCCUPIES
      1 WORD.  THE SECOND FIELD, WHICH IS IN THE LAST WORD OF THE
      RECORD, POINTS TO ANOTHER COURSEDESCRIP IF ONE EXISTS.  THE NULL
      VALUE OF COURSEDESCRIP POINTER ENDS THE LIST OF COURSES TAKEN,
      WHEREAS A NULL COURSES POINTER ENDS THE LINKED LIST OF
      COURSELISTING RECORDS.

      THE PROGRAM ALLOWS THE USER TO ADD COURSES, TO ADD STUDENTS,
      AND TO ADD COURSES TAKEN TO ANY STUDENT"S RECORD.  STUDENTS OR
      COURSES MAY BE DELETED, CAUSING THE RECORDS TO BE "RECLAIMED"
      AND LINKED ONTO THE FREELIST BY THE PROGRAM.  AFTER ANY RECORDS
      ARE CREATED OR RECLAIMED, THE NEXTAVL,FREELIST,SP, AND CDP VALUES
      ARE UPDATED IN RECORD ZERO OF THE FILE TO MAINTAIN ITS INTEGRITY
      IN THE EVENT THE PROGRAM ABNORMALLY TERMINATES, OR THE MACHINE
      HANGS.

      ALL INFORMATION IN THE SYSTEM CAN BE LISTED IN VARIOUS FORMS.

      THE FIND OPTION ENABLES THE USER TO DETERMINE INFORMATION ABOUT
      THE DATA BASE HE HAS ESTABLISHED.  FOR EXAMPLE, THERE IS A
      COMMAND WHICH DETERMINES WHAT REQUIRED COURSES HAVE BEEN TAKEN OR
      NEED TO BE TAKEN, EITHER FOR A PARTICULAR STUDENT OR ALL
      STUDENTS.  OTHER VARIATIONS OF THIS COMMAND FIND THE GRADE
      AND GRADE POINT AVERAGE FOR ONE STUDENT OR ALL STUDENTS.
      A THIRD OPTION FINDS EITHER ALL PEOPLE WHO TOOK A PARTICULAR
      COURSE, OR OUT OF THE PEOPLE WHO TOOK THAT COURSE THOSE WHO MADE A
      CERTAIN GRADE, OR FINDS THIS INFORMATION FOR ALL COURSES. THE LAST
      FIND OPTION LISTS ALL STUDENTS WHO HAVE A PARTICULAR DEGREE;
```

```
FILE REMOTE;
RECORD FILE SF DISK OLD(1,10,30);
RECORD STUDENT SF(NAME,ADDRESS,SSNO,COURSES,DEGREE,NEXT);
RECORD COURSEDESCRIP SF(DEPT,COURSENO,REQUIRED,DESCRIPTION,
       HOURS,NEXTCOURSE);
RECORD COURSELISTING SF(QUARTER,COURSEPOINTER,GRADE,COURSES);
STRING FIELD NAME (0) [0:26],
       ADDRESS (3) [2:37],
       SSNO (7) [7:9];
COURSELISTING FIELD COURSES (9) [1:17];
STRING FIELD DEGREE (9) [3:2];
STUDENT FIELD NEXT (9) [30:18];
STRING FIELD DEPT (0) [0:4],
       COURSENO (0) [4:3],
       REQUIRED(0) [7:1],
       DESCRIPTION (1) [0:64],
       HOURS (9) [2:3];
COURSEDESCRIP FIELD NEXTCOURSE (9) [30:18],
       COURSEPOINTER (0,8) [30:18];
REAL FIELD GRADE (0,8) [18:6], % TREAT  AS ALPHA FIELDS
       QUARTER (0,8) [24:6];
STUDENT SP;
COURSEDESCRIP CDP;
STRING
       NAMESTR(26),
       ADDSTR(37),
       SSNUMBER(9),
       DEGREESTR(2),
       DEPTSTR(4),
       COURSENOSTR(3),
       DESCRIPSTR(64),
       HOURSTR(3);
INTEGER T;
LABEL START;
STUDENT PROCEDURE LOOKUPSSNO (SSNUMBER);
       STRING SSNUMBER;
       FORWARD;



COMMENT
       THIS PROCEDURE "PRINS" THE COURSE DESCRIPTION INFORMATION IN
       A RECORD POINTED TO BY P.  SINCE DESCRIPTION(P) MAY CONTAIN
       TRAILING BLANKS, THESE ARE EFFECTIVELY DELETED SO THAT TAB
       POINTS TO ONE BLANK BEYOND THE INFORMATION;
```

```
PROCEDURE PRINCOURSEDESCRIP (P);
      VALUE P;
      COURSEDESCRIP P;
      BEGIN
      PRIN DEPT(P),COURSENO(P) REQUIRED(P),HOURS(P) SPACE(1);
      DESCRIPSTR := DESCRIPTION(P);
      I := 63;
      WHILE DESCRIPSTR(I,1) = " " AND I GTR 0 DO I := I - 1;
      I := I + 1;
      PRIN DESCRIPSTR(0,I);
      END;
COMMENT
      THIS PROCEDURE FIRST PRINTS THE QUARTER AND GRADE AND THEN
      CALLS PRINCOURSEDESCRIP FOR ALL COURSES IN THE LIST
      OF COURSELISTING RECORDS POINTED TO BY P;
PROCEDURE PRINTCOURSES(P);
      VALUE P;
      COURSELISTING P;
      BEGIN
      ALPHA A;
      INTEGER I;
      LABEL START;
      IF P=NIL THEN RETURN; % REFERENCE TO NULL RECORD CAUSES ERROR
START:FOR I := 0 STEP 1 UNTIL 8 DO
      BEGIN
      IF COURSEPOINTER[I](P) = NIL THEN RETURN;
      CASE QUARTER[I](P) OF
      BEGIN
      PRIN #FA #;
      PRIN #WI #;
      PRIN #SP #;
      PRIN #SU #;
      END;
      PRIN A := GRADE[I](P),;
      PRINCOURSEDESCRIP(COURSEPOINTER[I](P));
      TERPRI;
      END;
      IF P := COURSES(P) NEQ NIL THEN GO START;
      END;
PROCEDURE PRINTSTUDENT (P,N);
      VALUE P,N;
      STUDENT P; % WHICH STUDENT
      INTEGER N; % KEY TO WHAT IS PRINTED ABOUT STUDENT
      CASE N OF BEGIN               % N =
      PRINT #S: # SSNO(P);          % 0 - PRINT SOCIAL SECURITY NUMBER
      PRINT #N: # NAME(P);          % 1 - PRINT STUDENTS NAME
      PRINT #A: # ADDRESS(P);       % 2 - PRINT STUDENTS ADDRESS
      PRINT #D: # DEGREE(P);        % 3 - PRINT STUDENTS DEGREE
      PRINTCOURSES(COURSES(P));     % 4 - PRINT COURSES STUDENT HAS TAKEN
      END;
```

```
COMMENT
      PRINTALL LISTS ALL STUDENTS" NAMES, ADDRESSES, SOCIAL SECURITY
      NUMBERS, DEGREES, COURSES TAKEN INCLUDING
      THE QUARTER THE COURSE WAS TAKEN AND THE GRADE RECEIVED;

PROCEDURE PRINTALL;
      BEGIN
      STUDENT P;
      INTEGER I;
      IF P := SP = NIL THEN
      BEGIN
      PRINT #STUDENT LIST EMPTY#;
      RETURN;
      END;
      WHILE P NEQ NIL DO
      BEGIN
      FOR I:=0 STEP 1 UNTIL 4 DO PRINTSTUDENT(P,I);
      TERPRI;
      P := NEXT(P);
      END;
      END;



COMMENT
      LISTER LISTS VARIOUS THINGS FROM THE DIFFERENT RECORDS.  THE
      COMMANDS ARE:

            L [SOCIAL SECURITY NUMBER] [OPTION LIST]

            WHERE
                  [OPTION LIST] ::= [OPTION] , [OPTION LIST]
                  [OPTION] ::= S % PRINT SOCIAL SECURITY NUMBER
                             / N % PRINT NAME
                             / A % PRINT ADDRESS
                             / D % PRINT DEGREE
                             / C % PRINT COURSES TAKEN

            L S   LISTS THE SOCIAL SECURITY NUMBER AND NAMES
                  OF ALL STUDENTS IN THE DATA BASE

            L C   LISTS THE DESCRIPTION OF ALL COURSES:
                  THE DEPARTMENT, COURSE NUMBER,
                  FOLLOWED BY AN OPTIONAL ASTERISK (*) WHICH
                  INDICATES THAT THE COURSE IS REQUIRED FOR GRADUATION,
                  THE COURSE HOURS AND COURSE TITLE;
```

```
PROCEDURE LISTER;
      BEGIN
      STUDENT S;
      COURSEDESCRIP C;
      INTEGER T;
      LABEL START;
      IF "0" LEQ REAL(TWXS1(2,1)) LEQ "9" THEN
      BEGIN
      SSNUMBER := TWXS1(2,9);
      IF S := LOOKUPSSNO(SSNUMBER) = NIL THEN
      BEGIN
      PRINT #STUDENT NOT FOUND#;
      RETURN;
      END;
      COL := 12;
START:IF T := READN(TWXA) = "S" THEN T := 0 ELSE
      IF T = "N" THEN T := 1 ELSE
      IF T = "A" THEN T := 2 ELSE
      IF T = "D" THEN T := 3 ELSE
      IF T = "C" THEN T := 4 ELSE
      BEGIN
      PRINT #ILLEGAL INPUT#;
      RETURN;
      END;
      PRINTSTUDENT(S,T);
      IF TWXS1(COL,1) = "," THEN
      BEGIN
      COL := COL + 1;
      GO START;
      END;
      RETURN;
      END;
      IF T := REAL(TWXS1(2,1)) = "S" THEN
      BEGIN
      IF S := SP = NIL THEN
      BEGIN
      PRINT #STUDENT LIST EMPTY#;
      RETURN;
      END;
      WHILE S NEQ NIL DO
      BEGIN
      PRINT SSNO(S),NAME(S);
      S := NEXT(S);
      END;
      RETURN;
      END;
      IF T = "C" THEN
      BEGIN
      IF C := CDP = NIL THEN
      BEGIN
```

```
        PRINT #COURSE LIST EMPTY#;
        RETURN;
        END;
        WHILE C NEQ NIL DO
        BEGIN
        PRINCOURSEDESCRIP (C);
        TERPRI;
        C := NEXTCOURSE(C);
        END;
        RETURN;
        END;
        PRINT #ILLEGAL LIST COMMAND#;
        END LISTER;

COMMENT
        THIS PROCEDURE SEARCHES FOR A STUDENT SOCIAL SECURITY NUMBER WHICH
        MATCHES SSNUMBER.  IF SUCH A STUDENT EXISTS THEN A POINTER
        TO HIS RECORD IS RETURNED, OTHERWISE THE NULL POINTER IS
        RETURNED;

STUDENT PROCEDURE LOOKUPSSNO (SSNUMBER);
        STRING SSNUMBER;
        BEGIN
        STUDENT P;
        P := SP;
        WHILE P NEQ NIL DO
        BEGIN
        IF SSNO(P) = SSNUMBER (0,9) THEN RETURN P;
        P := NEXT (P);
        END;
        END;

COMMENT
        THIS PROCEDURE ADDS STUDENTS TO THE DATA BASE.  THE
        PROCEDURE LOOPS ASKING FOR THE FOLLOWING INFORMATION
        FOR EACH STUDENT TO BE ENTERED INTO THE SYSTEM:

                SSNO:        EXPECTS A 9 DIGIT SOCIAL SECURITY NUMBER OR
                             IF THE WORD "STOP" OR A BLANK LINE
                             IS FOUND THEN THE PROCEDURE IS EXITED.
                NAME:        EXPECTS THE NAME OF THE STUDENT UP TO 20
                             CHARACTERS IN LENGTH.
                ADDRESS:     EXPECTS THE ADDRESS OF THE STUDENT UP TO 37
                             CHARACTERS IN LENGTH.
                DEGREE:      EXPECTS THE DEGREE OF THE STUDENT
                             (2 CHARACTERS) SUCH AS EE, IE, OR CE.

        IF THE STUDENT ALREADY EXISTS ON THE SYSTEM THEN AN ERROR
        MESSAGE IS TYPED, OTHERWISE THE NEW STUDENT IS ADDED TO THE
        SYSTEM;
```

```
PROCEDURE ADDSTUDENT;
      BEGIN
      LABEL LOOP;
LOOP: PRINT #SSNO: #; READ TWX; SSNUMBER := TWXS1(0,9);
      IF SSNUMBER(0,4) = "STOP" OR SSNUMBER = SPACE THEN RETURN;
      IF LOOKUPSSNO(SSNUMBER) NEQ NIL THEN
      BEGIN
      PRINT #STUDENT ALREADY ON SYSTEM#;
      GO TO LOOP;
      END;
      PRINT #NAME: #; READ TWX; NAMESTR := TWXS1(0,26);
      PRINT #ADDRESS: #; READ TWX; ADDSTR := TWXS1(0,37);
      PRINT #DEGREE: #; READ TWX; DEGREESTR := TWXS1(0,2);
      % CREATE NEW STUDENT RECORD
      SP := STUDENT(NAMESTR,ADDSTR,SSNUMBER,*,DEGREESTR,SP);
      GO TO LOOP; % DE LOOP
      WRITE(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,CDP);
      END;

COMMENT
      DELETESTUDENT DELETES STUDENTS CURRENTLY ON THE SYSTEM.
      WHEN A STUDENT IS DELETED HIS OLD RECORD IS LINKED INTO
      THE FREELIST BY THE RECLAIM STATEMENT SO THAT THE RECORD
      CAN LATER BE USED.  ALSO ALL OF HIS COURSELISTING RECORDS
      ARE RECLAIMED.  THE PROCEDURE EITHER ASKS FOR A SOCIAL
      SECURITY NUMBER OR IT MAY BE  GIVEN IN THE COMMAND "D S"
      SUCH AS "D S 405628801".  A SERIES OF SOCIAL SECURITY NUMBERS
      MAY BE GIVEN, IF EACH PRECEDING ONE IF FOLLOWED BY A COMMA.
      AS EACH STUDENT IS DELETED HIS NAME IS PRINTED AS FEEDBACK
      TO THE USER;

PROCEDURE DELETESTUDENT;
      BEGIN
      STUDENT S,T;
      COURSELISTING C,D;
      LABEL LOOP,RC,L,L1;
      IF SSNUMBER:=TWXS1(4,9) = SPACE(9) THEN
      BEGIN
      LOOP: PRINT #SSNO:#;
      READ TWX;
L:    SSNUMBER := TWXS1(0,9);
      END ELSE TWXS1 := TWXS1(4) & SPACE;
      IF SSNUMBER(0,4) = "STOP" OR SSNUMBER = SPACE THEN RETURN;
      IF S := T := SP = NIL THEN
      BEGIN
      PRINT #STUDENT LIST EMPTY#;
      RETURN;
      END;
```

```
        IF SSNUMBER(0,9) = SSNO(SP) THEN % ITS THE FIRST RECORD
        BEGIN
        PRINTSTUDENT(SP,1);
        C := COURSES(S);
        WHILE D:=C NEQ NIL DO % DELETE COURSE DESCRIPTION RECORDS
        BEGIN
        C := COURSES(C); % POINTER TO NEXT RECORD
        RECLAIM(D); % RECLAIM THE RECORD
        END;
        S := NEXT(SP);
        RECLAIM(SP);
        SP := S; % RESTORE STUDENT POINTER
        GO TO RC;
        END;
        WHILE S := NEXT(S) NEQ NIL DO % SCAN DOWN THE LIST
        BEGIN
        IF SSNUMBER(0,9) = SSNO(S) THEN % FOUND HIM
        BEGIN
        PRINTSTUDENT(S,1);
        C := COURSES(S);
        WHILE D := C NEQ NIL DO % DELETE COURSE DESCRIPTION RECORDS
        BEGIN
        C := COURSES(C);
        RECLAIM(D);
        END;
        NEXT(T) := NEXT(S); % DELINK THE RECORD
        RECLAIM(S); % LINK RECLAIMED RECORD ONTO FREELIST
        GO TO RC;
        END;
        T := S; % KEEP TAB OF TRAILING POINTER
        END OF WHILE LOOP;
        PRIN SSNUMBER, #NOT FOUND#; % NOT ON SYSTEM - NOTIFY USER
        IF TWXS1(9,1) = ",", THEN PRINT #,CONTINUING# ELSE TERPRI;
        GO TO L1;
RC:     WRITE(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,CDP); % UPDATE FILE
L1:     IF TWXS1(9,1) = "," THEN
        BEGIN
        TWXS1:=TWXS1(10) & SPACE;
        GO TO L;
        END;
        GO TO LOOP;
        END DELETESTUDENT;
```

COMMENT

```
        LOOKUPCOURSE IS THE SAME AS LOOKUPSSNO EXCEPT IT
        LOOKS UP A COURSE GIVEN BY DEPT NAME(DT) AND COURSE NUMBER(CN).
        IT RETURNS THE ADDRESS OF THE COURSEDESCRIP RECORD IF SUCH
        A COURSE EXISTS;
```

```
COURSEDESCRIP PROCEDURE LOOKUPCOURSE (DT,CN);
      STRING DT,CN;
      BEGIN
      COURSEDESCRIP P;
      P := CDP;
      WHILE P NEQ NIL DO
      BEGIN
      IF DT(0,4) = DEPT(P) AND CN(0,3) = COURSENO(P) THEN RETURN P;
      P := NEXTCOURSE (P);
      END;
      END;

COMMENT
      ADDCOURSE ADDS COURSES TO THE SYSTEM.  THE INPUT IS AS FOLLOWS

            DEPT:          EXPECTS THE DEPT NAME (4 CHARACTERS).
                           IF THE WORD "STOP" OR A BLANK LINE IS ENTERED
                           THEN THE PROCEDURE IS EXITED.
            COURSENO:      EXPECTS THE COURSE NUMBER OF THE COURSE.
                           AT THIS TIME THE COURSEDESCRIP RECORDS ARE
                           SEARCHED BY CALLING LOOKUP TO SEE IF THE
                           COURSE ALREADY EXISTS, AND IF IT DOES AN ERROR
                           MESSAGE IS TYPED.
            REQUIRED:      EXPECTS "Y" OR "YES" IF THE COURSE IS REQUIRED
                           FOR GRADUATION, OTHERWISE A BLANK LINE
                           WILL SUFFICE FOR A NO RESPONSE.
            DESCRIPTION:   EXPECTS THE COURSE TITLE TO BE ENTERED UP TO
                           64 CHARACTERS IN LENGTH
            HOURS:         EXPECTS 3 CHARACTERS OF THE FORM "303" WHERE
                           THE FIRST DIGIT INDICATES THE NUMBER
                           OF HOURS OF IN-CLASS INSTRUCTION, THE SECOND
                           DIGIT INDICATES THE NUMBER OF HOURS SPENT
                           IN LAB PER WEEK, AND THE THIRD DIGIT
                           INDICATES  THE HOURS EARNED FOR THAT COURSE.

      AT THIS TIME THE NEW RECORD IS CREATED AND THEN THE PROCEDURE
      LOOPS;

PROCEDURE ADDCOURSE;
      BEGIN
      BOOLEAN B;
      LABEL LOOP;
LOOP: PRINT #DEPT: #; READ TWX; DEPTSTR := TWXS1(0,4);
      IF DEPTSTR = "STOP" OR DEPTSTR = SPACE(4) THEN RETURN;
      PRINT #COURSE NO: #; READ TWX; COURSENOSTR := TWXS1(0,3);
      IF LOOKUPCOURSE(DEPTSTR,COURSENOSTR) NEQ NIL THEN
      BEGIN
      PRINT #COURSE ALREADY ENTERED#;
      GO TO LOOP;
      END;
```

```
        PRINT #REQUIRED:#; READ TWX;
        B := TWXS1(0,1) = "Y";
        PRINT #DESCRIPTION: #; READ TWX; DESCRIPSTR := TWXS1(0,64);
        PRINT #HOURS: #; READ TWX; HOURSTR := TWXS1(0,3);
        CDP := COURSEDESCRIP(DEPTSTR,COURSENOSTR, % CREATE COURSE RECORD
        STRING(IF B THEN "*" ELSE " ",1),DESCRIPSTR,
              HOURSTR,CDP);
        GO TO LOOP;
        WRITE(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,CDP);
        END;

COMMENT
        DELETECOURSE DELETES A COURSE SIMILAR TO DELETESTUDENT.
        THE DEPARTMENT NAME AND COURSE NUMBER ARE REQUESTED.
        IF THE COURSE RECORD IS FOUND THEN IT IS RECLAIMED
        SO THAT IT CAN BE USED LATER.  IF THE COURSE CAN NOT
        BE FOUND THEN AN ERROR MESSAGE IS PRINTED.
        ****** SOME CARE SHOULD BE USED IN DELETING COURSES BECAUSE ******
        ****** THERE MAY BE POINTERS IN COURSELISTING POINTING  TO  ******
        ****** THE RECORD BEING DELETED, THUS ERRONEOUS LISTINGS    ******
        ****** MAY RESULT                                           *****;
PROCEDURE DELETECOURSE;
        BEGIN
        COURSEDESCRIP C,T;
        LABEL LOOP,RC,L;
LOOP:   PRINT #DEPT:#; READ TWX; DEPTSTR := TWXS1(0,4);
        IF DEPTSTR = "STOP" OR DEPTSTR = SPACE THEN RETURN;
        PRINT #COURSE NO:#; READ TWX; L:COURSENOSTR:=TWXS1(0,3);
        IF C := T := CDP = NIL THEN
        BEGIN
        PRINT #COURSE LIST EMPTY#;
        RETURN;
        END;
        IF DEPTSTR = DEPT(CDP) AND COURSENOSTR = COURSENO(CDP) THEN
        BEGIN % ITS THE FIRST RECORD
        C := NEXTCOURSE(CDP);
        RECLAIM(CDP);
        CDP := C;
        GO TO RC;
        END;
        WHILE C := NEXTCOURSE(C) NEQ NIL DO % SCAN DOWN THE LIST
        BEGIN
        IF DEPTSTR = DEPT(C) AND COURSENOSTR = COURSENO(C) THEN % GOT HIM
        BEGIN
        NEXTCOURSE(T) := NEXTCOURSE(C);
        RECLAIM(C);
        GO TO RC;
        END;
        T := C;
        END OF WHILE LOOP; %
```

```
      PRINT #COURSE NOT FOUND#; % COURSE NOT ON SYSTEM
      GO TO LOOP;
RC:   WRITE(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,CDP);
      IF TWXS1(3,1) = "," THEN
      BEGIN
      TWXS1 := TWXS1(4) & SPACE;
      GO TO L;
      END;
      GO TO LOOP;
      END DELETECOURSE;

COMMENT
      ADDCOURSESTOSTUDENT ADDS COURSES TAKEN TO A STUDENT RECORD.
      THESE INPUTS ARE PUT INTO COURSELISTING AND
      THEN IF IT IS THE FIRST COURSE TAKEN, OR AN OLD RECORD IS
      FULL (EACH RECORD HOLDS 9 COURSES) THEN A NEW RECORD IS
      CREATED AND LINKED INTO THE LIST.  OTHERWISE THE INFORMATION
      IS ADDED TO AN EXISTING RECORD.  THE PROCEDURE VERIFIES
      THAT BOTH THE STUDENT AND THE COURSE EXIST.  THE INPUT
      IS REQUESTED AS FOLLOWS:

            SSNO:          EITHER AN SS NO. OR "STOP" OR A BLANK.
            DEPT:          DEPARTMENT NAME (4 CHARACTERS).
            COURSENO:      COURSE NUMBER.
            GRADE:         GRADE RECEIVED, EITHER A OR B OR C OR D OR F.
            QUARTER:       THIS INDICATES THE QUARTER THE COURSE WAS
                           TAKEN.  ENTER EITHER FALL OR F, SPRING OR
                           SP, WINTER OR WI, OR SUMMER OR SU.

      THE PROCEDURE LOOPS UNTIL "STOP" OR  BLANK IS ENTERED
      AT THE REQUEST FOR SSNO:.  IF AN ERROR OCCURS AN APPROPRIATE
      ERROR MESSAGE IS TYPED;

PROCEDURE ADDCOURSESTOSTUDENT;
      BEGIN
      STUDENT S;
      COURSEDESCRIP P;
      ALPHA A,B;
      COURSELISTING CL;
      INTEGER I;
      LABEL DONE,LOOP,Q1,GD;
      PRINT #SSNO: #; READ TWX; SSNUMBER := TWXS1(0,9);
      IF S := LOOKUPSSNO(SSNUMBER) = NIL THEN
      BEGIN
      PRINT #STUDENT NOT FOUND#;
      RETURN;
      END;
      PRINT NAME(S);
LOOP: PRINT #DEPT: #; READ TWX; DEPTSTR := TWXS1(0,4);
      IF DEPTSTR = "STOP" OR DEPTSTR = SPACE(4) THEN RETURN;
```

```
      PRINT #COURSENO: #; READ TWX; COURSENOSTR := TWXS1(0,3);
      IF P := LOOKUPCOURSE(DEPTSTR,COURSENOSTR) = NIL THEN
      BEGIN
      PRINT #NO SUCH COURSE#;
      GO TO LOOP;
      END;
GD:   PRINT #GRADE: #; READ TWX; A := REAL(TWXS1(0,1));
      IF "A" GEQ A GEQ "F" OR A = "E" THEN
      BEGIN
      PRINT #ILLEGAL GRADE#;
      GO GD;
      END;
QT:   PRINT #QUARTER: #; READ TWX; B := REAL(TWXS1(0,2));
      IF B = "F " OR B = "FA" THEN B := 0 ELSE
      IF B = "W " OR B = "WI" THEN B := 1 ELSE
      IF B = "SP" THEN B := 2 ELSE
      IF B = "SU" THEN B := 3 ELSE
      BEGIN
      PRINT #NO SUCH QUARTER#;
      GO TO QT;
      END;
      IF CL := COURSES(S) = NIL THEN GO DONE;
      WHILE CL NEQ NIL DO
      BEGIN
      FOR I := 0 STEP 1 UNTIL 8 DO
      IF COURSEPOINTER[I](CL) = NIL THEN GO DONE;
      CL := COURSES(CL);
      END;
DONE: IF I = 0 OR I = 9 THEN
      BEGIN % CREATE NEW COURSELISTING RECORD
      COURSES(S) := COURSELISTING(B,P,A,COURSES(S));
      WRITE(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,COP);
      END ELSE
      BEGIN % UPDATE OLD COURSELISTING RECORD
      QUARTER[I](CL) := B;
      COURSEPOINTER[I](CL) := P;
      GRADE[I] (CL) := A;
      END;
      GO TO LOOP;
      END;



COMMENT
      FINDDEGREE FINDS ALL STUDENTS WHO HAVE A DEGREE GIVEN IN
      THE COMMAND "F D [DEGREE]" WHERE [DEGREE] IS ANY 2
      CHARACTERS;
```

```
PROCEDURE FINDDEGREE;
      BEGIN
      STUDENT S;
      DEGREESTR := TWXS1(4,2);
      S := SP;
      WHILE S NEQ NIL DO
      BEGIN
      IF DEGREESTR = DEGREE(S) THEN PRINT SSNO(S);
      S := NEXT (S);
      END;
      END;


COMMENT
      FINDGRADE EITHER FINDS THE GRADES FOR A PARTICULAR STUDENT
      OR ALL STUDENTS.  IF THE SOCIAL SECURITY NUMBER IS GIVEN
      IN THE COMMAND "F G [SSNO]" THEN THAT STUDENT"S GRADES ARE
      SUMARIZED, OTHERWISE ALL STUDENTS" SUMMARIES ARE PRINTED;

PROCEDURE FINDGRADE;
      BEGIN
      STUDENT S;
      COURSELISTING CL;
      STRING LINE(26);
      ALPHA G;
      INTEGER I,QP,HT,H;
      ARRAY GC[0:4];
      BOOLEAN B,PRINTED;
      FORMAT HEAD("  A  B  C  D  F QP HE  PA  NAME");
      FORMAT INFO(7(I3),F5.2);
      LABEL L1,L2,L3,LS;
      IF B := SSNUMBER := TWXS1(4,9) NEQ SPACE THEN
      BEGIN
      IF S := LOOKUPSSNO(SSNUMBER) = NIL THEN
      BEGIN
      PRINT #STUDENT NOT FOUND#;
      RETURN;
      END;
      END ELSE S := SP;
      IF S = NIL THEN
      BEGIN
      PRINT #STUDENT LIST EMPTY#;
      RETURN;
      END;
      WHILE S NEQ NIL DO
      BEGIN
      IF CL := COURSES(S) = NIL THEN GO TO LS;
      FOR I := 0 STEP 1 UNTIL 4 DO GC[I]:=0;
      HT := QP := 0;
```

```
L1:    FOR I := 0 STEP 1 UNTIL 8 DO
       BEGIN
       IF COURSEPOINTER[I](CL) = NIL THEN GO TO L2;
       GC[GRADE[I](CL)-"A"] := * + 1;
       LINE := HOURS(COURSEPOINTER[I](CL));
       H := REAL(LINE(2,1));
       HT := HT + H;
       GP := GP + H x (IF H:=GRADE[I](CL) = "F" THEN 0 ELSE "E"-H);
       END;
       IF CL := COURSES(CL) NEQ NIL THEN GO TO L1;
L2:    IF HT NEQ 0 THEN
       BEGIN
       IF NOT PRINTED THEN WRITE(TWXF2,HEAD);
       PRINTED := TRUE;
       WRITE(LINE,INFO,FOR I:=0 STEP 1 UNTIL 4 DO GC[I],GP,HT,GP/HT);
       PRINT LINE(0,26), NAME(S);
       END;
LS:    IF B THEN GO TO L3;;
       S := NEXT(S);
       END;
L3:    IF NOT PRINTED THEN PRINT #NULL#;
       END;

COMMENT
       FINDCOURSE FINDS EITHER ALL PEOPLE WHO HAVE TAKEN A PARTICULAR
       COURSE OR ALL PEOPLE WHO MADE A PARTICULAR GRADE.
       THE COMMAND "F G" CAUSES THIS PROCEDURE TO BE ENTERED WHICH
       THEN REQUESTS THE REST OF THE NECESSARY INFORMATION.
       THE FOLLOWING IS REQUESTED:

              DEPT:          SAME AS FOR ADD COURSES
              COURSENO:          "
              GRADE:         IF A GRADE IS GIVEN THEN THAT COURSE
                             WILL BE LISTED WITH ALL PEOPLE WHO TOOK
                             IT AND MADE THAT GRADE ELSE ALL PEOPLE
                             WHO TOOK THAT COURSE WILL BE LISTED;

PROCEDURE FINDCOURSE;
       BEGIN
       STUDENT S;
       COURSELISTING CL;
       COURSEDESCRIP CP;
       ALPHA G;
       BOOLEAN B,PRINTED;
       INTEGER I;
       LABEL L1,L2,LOOP;
LOOP:  PRINT #DEPT:#; READ TWX; DEPTSTR := TWXS1(0,4);
       IF DEPTSTR = "STOP" OR DEPTSTR = SPACE THEN RETURN;
       PRINT #COURSENO:#; READ TWX; COURSENOSTR := TWXS1(0,3);
       PRINT #GRADE:#; READ TWX; B := G := REAL(TWXS1(0,1)) = " ";
```

```
      IF LOOKUPCOURSE(DEPTSTR,COURSENOSTR) = NIL THEN
      BEGIN
      PRINT #NO SUCH COURSE#;
      GO TO LOOP;
      END;
      S := SP;
      WHILE S NEQ NIL DO
      BEGIN
      IF CL := COURSES(S) = NIL THEN GO TO L2;
L1:   FOR I := 0 STEP 1 UNTIL 8 DO
      BEGIN
      IF CP := COURSEPOINTER[I](CL) = NIL THEN GO TO L2;
      IF DEPT(CP) = DEPTSTR AND COURSENO(CP) = COURSENOSTR AND
            (B OR GRADE[I](CL) = G) THEN
      BEGIN
      PRINT SSNO(S), NAME(S);
      PRINTED := TRUE;
      GO TO L2;
      END;
      END;
      IF CL := COURSES(CL) NEQ NIL THEN GO TO L1;
L2:   S := NEXT(S);
      END;
      IF NOT PRINTED THEN PRINT #NULL#;
      END FINDCOURSE;
```

```
COMMENT
      FINDREQUIRED FINDS ALL OF THE REQUIRED COURSES THAT A
      STUDENT HAS OR HAS NOT TAKEN.   THIS CAN BE DONE FOR EITHER AN

      INDIVIDUAL OR ALL STUDENTS.  THE COMMANDS ARE:

            F R [SOCIAL SECURITY NO] [OPTION]

                  [OPTION] ::= L / [EMPTY]

      IF THE SOCIAL SECURITY NUMBER IS OMITTED THEN ALL
      STUDENTS ARE PRINTED.  IF THE OPTION FIELD IS EMPTY THEN ALL
      REQUIRED COURSES THAT HAVE BEEN TAKEN ARE LISTED, ELSE
      IF THE "L" APPEARS THEN THE COURSES THAT ARE
      REQUIRED  AND HAVE NOT BEEN TAKEN ARE LISTED;
```

```
PROCEDURE FINDREQUIRED;
        BEGIN
        STUDENT S;
        COURSELISTING CL;
        COURSEDESCRIP CP,DP;
        INTEGER I;
        LABEL L1,L2,L3,L4,L5,L6;
        BOOLEAN B,PRINTED,NOCOURSES;
        IF B := SSNUMBER := TWXS1(4,9) = SPACE OR
                SSNUMBER(0,1) = "L" THEN S := SP ELSE
        IF S := LOOKUPSSNO(SSNUMBER) = NIL THEN
        BEGIN
        PRINT IF SP = NIL THEN #STUDENT LIST EMPTY# ELSE
                #STUDENT NOT FOUND#;
        RETURN;
        END;
        IF TWXS1(IF B THEN 3 ELSE 13,3) NEQ SPACE(3) THEN GO TO L3;
        WHILE S NEQ NIL DO
        BEGIN
        PRINTED := FALSE;
        IF CL := COURSES(S) = NIL THEN GO TO L2;
L1:     FOR I := 0 STEP 1 UNTIL 8 DO
        BEGIN
        IF CP := COURSEPOINTER[I](CL) = NIL THEN GO TO L2;
        IF REQUIRED(CP) = "*" THEN
        BEGIN
        IF NOT PRINTED THEN
        BEGIN
        TERPRI;
        IF B THEN PRINT SSNO(S),NAME(S) ELSE PRINTSTUDENT(S,1);
        PRINTED := TRUE;
        END;
        PRINCOURSEDESCRIP(CP);
        TERPRI;
        END;
        END;
        IF CL := COURSES(CL) NEQ NIL THEN GO TO L1;
L2:     IF NOT PRINTED THEN
        BEGIN
        TERPRI;
        IF B THEN PRINT SSNO(S),NAME(S) ELSE PRINTSTUDENT(S,1);
        PRINT #NO REQUIRED COURSES TAKEN YET#;
        END;
        IF NOT B THEN RETURN;
        S := NEXT(S);
        END;
        RETURN;
L3:     WHILE S NEQ NIL DO
        BEGIN
        PRINTED := FALSE;
```

```
      DP := CDP;
      WHILE DP NEQ NIL DO
      BEGIN
      IF REQUIRED(DP) NEQ "*" THEN GO TO L6;
      NOCOURSES := CL := CCURSES(S) = NIL;
L4:   IF NOCOURSES THEN GO TO L5;
      FOR I := 0 STEP 1 UNTIL 8 DO
      BEGIN
      IF CP := COURSEPOINTER[I](CL) = NIL THEN GO TO L5;
      IF CP = DP THEN GO TO L6;
      END;
      IF CL := COURSES(CL) NEQ NIL THEN GO TO L4;
L5:   IF NOT PRINTED THEN
      BEGIN
      IF B THEN PRINT SSNO(S),NAME(S) ELSE PRINTSTUDENT(S,1);
      PRINTED := TRUE;
      END;
      PRINCOURSEDESCRIP(DP);
      TERPRI;
L6:   DP := NEXTCOURSE(DP);
      END;
      IF NOT PRINTED THEN
      BEGIN
      PRINTSTUDENT(S,1);
      PRINT #ALL REQUIREMENTS SATISFIED#;
      PRINTED := TRUE;
      END;
      IF NOT B THEN RETURN;
      S := NEXT(S);
      END;
      END FINDREQUIRED;

COMMENT
      THE COMMANDS ARE:

              COMMANDS        ACTION

              LIST             CALL PRINTALL
              L ALL            CALL PRINTALL
              L [OPTIONS]      CALL LISTER
                               SEE LISTER FOR [OPTIONS]
              D S              CALL DELETESTUDENT
              D S [SS NO, ...]

              A S              CALL ADDSTUDENT
              A C              CALL ADDCOURSES

              U C              CALL ADDCOURSESTOSTUDENT %UPDATE COURSES

              STOP             END OF PROGRAM %WRAP UP
```

```
      FIND COMMANDS

            F D [DEGREE]    CALL FINDDEGREE

            F G             CALL FINDGRADE
            F G [SSNO]      CALL FINDGRADE

            F C             CALL FINDCOURSES

            F R             CALL FINDREQUIRED
            F R L           CALL FINDREQUIRED
            F R [SSNO]      CALL FINDREQUIRED
            F R [SSNO] L    CALL FINDREQUIRED;

      PRINT #GO AHEAD.#;
      READ(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,COP);
START:PRINT #:#;
      READ TWX;
      IF T := REAL(TWXS1(0,4)) = "LIST" OR T = "L AL" THEN PRINTALL ELSE
      IF TWXS1(0,1) ="L" THEN LISTER ELSE
      IF T = "D S " THEN DELETESTUDENT ELSE
      IF T = "D C " THEN DELETECOURSE ELSE
      IF T = "A S " THEN ADDSTUDENT ELSE
      IF T = "A C " THEN ADDCOURSE ELSE
      IF T = "U C " THEN ADDCOURSESTOSTUDENT ELSE
      IF T = "F D " THEN FINDDEGREE ELSE
      IF T = "F G " THEN FINDGRADE ELSE
      IF T = "F C " THEN FINDCOURSE ELSE
      IF T = "F R " THEN FINDREQUIRED ELSE
      IF T = "STOP" THEN
      BEGIN
      WRITE(SF[0],*,NEXTAVL(SF),FREELIST(SF),SP,COP);
      PRINT #END-OF-UPDATE#;
      PRINT #GOODBYE#;
      EXIT;
      END ELSE
      PRINT #ILLEGAL INPUT#;
      GO START;
END.
```

RUN RECORD

```
 -BLJ- ORECORD
GO AHEAD.
:L C
ICS   452   536 LOGIC DESIGN AND SWITCHING THEORY
ICS   445   303 LOGISTIC SYSTEMS
ICS   436   303 INFORMATION SYSTEMS
ICS   424   303 ELEMENTS OF INFORMATION THEORY
ICS   423   303 MATHEMATICAL TECHNIQUES FOR INFORMATION SCIENCE
ICS   415   233 THE LITERATURE OF SCIENCE AND ENGINEERING
ICS   410*  303 PROBLEM SOLVING
ICS   406*  303 COMPUTING LANGUAGES
ICS   404   303 TOPICS IN LINGUISTICS
ICS   402   303 LANGUAGES FOR SCIENCE AND TECHNOLOGY
ICS   401   303 LANGUAGES FOR SCIENCE AND TECHNOLOGY
ICS   355*  303 INFORMATION STRUCTURES AND PROCESSES
ICS   342*  303 INTRODUCTION TO SEMIOTICS
ICS   336   303 INTRODUCTION TO INFORMATION ENGINEERING
ICS   325*  303 INTRODUCTION TO CYBERNETICS
MATH  239*  303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310*  233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256*  303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   251   233 AUTOMATIC DATA PROCESSING
ICS   215   101 TECHNICAL INFORMATION RESOURSES
ICS   151*  233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING
:L S
405628804 CHUCK COVERALL
405628801 BARRY FOLSOM
405628806 BILL BROWN
405628805 RAY SPUTNICK
405628803 JERRY CIGARS
405628802 JOHN FOSTER
:F G
  A  B  C  D  F GP HE  PA   NAME
  2  1  1  0  0 39 12 3.25 BILL BROWN
  3  3  2  1  0 78 27 2.89 RAY SPUTNICK
:F R 405628804 L
N: CHUCK COVERALL
ICS   410*  303 PROBLEM SOLVING
ICS   406*  303 COMPUTING LANGUAGES
ICS   355*  303 INFORMATION STRUCTURES AND PROCESSES
ICS   342*  303 INTRODUCTION TO SEMIOTICS
ICS   325*  303 INTRODUCTION TO CYBERNETICS
MATH  239*  303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310*  233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256*  303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   151*  233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING
:F R 405628805 L
N: RAY SPUTNICK
```

```
ALL REQUIREMENTS SATISFIED
:F R 405628801

N: BARRY FOLSOM
NO REQUIRED COURSES TAKEN YET
:A S
SSNO:405628807
NAME:BUD PASSCUT
ADDRESS:5700 MOORES CENTER PLACE
DEGREE:IE
SSNO:
:U C
SSNO:405628807
BUD PASSCUT
DEPT:ICS
COURSENO:151
GRADE:A
QUARTER:FALL
DEPT:ICS
COURSENO:250
NO SUCH COURSE
DEPT:
:F C
DEPT:ICS
COURSENO:151
GRADE:
405628807 BUD PASSOUT
405628806 BILL BROWN
405628805 RAY SPUTNICK
:F R

405628807 BUD PASSOUT
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING

405628804 CHUCK COVERALL
NO REQUIRED COURSES TAKEN YET

405628801 BARRY FOLSOM
NO REQUIRED COURSES TAKEN YET

405628806 BILL BROWN
MATH 239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   325* 303 INTRODUCTION TO CYBERNETICS
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING

405628805 RAY SPUTNICK
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
```

```
ICS   325* 303 INTRODUCTION TO CYBERNETICS
MATH 239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING

405628803 JERRY CIGARS
NO REQUIRED COURSES TAKEN YET

405628802 JOHN FOSTER
NO REQUIRED COURSES TAKEN YET
:F R L
405628807 BUD PASSOUT
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
ICS   325* 303 INTRODUCTION TO CYBERNETICS
MATH 239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
405628804 CHUCK COVERALL
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
ICS   325* 303 INTRODUCTION TO CYBERNETICS
MATH 239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING
405628801 BARRY FOLSOM
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
ICS   325* 303 INTRODUCTION TO CYBERNETICS
MATH 239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING
405628806 BILL BROWN
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
N: RAY SPUTNICK
ALL REQUIREMENTS SATISFIED
```

```
405626803 JERRY CIGARS
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
ICS   325* 303 INTRODUCTION TO CYBERNETICS
MATH  239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING
405626802 JOHN FOSTER
ICS   410* 303 PROBLEM SOLVING
ICS   406* 303 COMPUTING LANGUAGES
ICS   355* 303 INFORMATION STRUCTURES AND PROCESSES
ICS   342* 303 INTRODUCTION TO SEMIOTICS
ICS   325* 303 INTRODUCTION TO CYBERNETICS
MATH  239* 303 INTRODUCTION TO SET-THEORETIC CONCEPTS
ICS   310* 233 COMPUTER-ORIENTED NUMERICAL METHODS
ICS   256* 303 COMPUTER AND PROGRAMMING SYSTEMS
ICS   151* 233 DIGITAL COMPUTER ORGANIZATION AND PROGRAMMING
:STOP
END-OF-UPDATE
GOODBYE
```

VIII. SYNTAX-DIRECTED PARSING

8.1 INTRODUCTION

Experiments with the GTL list processing facility and the GTL string processing system indicate that GTL may often serve as a convenient basis for writing artificial language translators. As a result of these experiments, several extensions of GTL have been implemented with the goal of achieving a general purpose translator-writing system within the framework of GTL. This section describes the extension of GTL in this direction.

The GTL parsing facility is intended to be used primarily as a top-down no-backtracking parser (Reference 5). Syntax rules are specified through BNF-like declarations called "SYMBOL FORMAT" declarations. Matching of sequences of symbol strings is intended to be accomplished primarily through the use of the symbol table provided by the GTL system for LISP atomic symbols. It is suggested that, in order to make effective use of the constructs described, the user be familiar with Sections V, VI, and IX. Also, the plex processing system described in Section VII has been found to be helpful in writing translators.

In order to illustrate the general form and meaning of a GTL syntax declaration, consider the following simple BNF definition of a simplified arithmetic expression.

```
⟨ae⟩  ::= ⟨p⟩ ⟨sec⟩
⟨sec⟩ ::= ⟨op⟩ ⟨p⟩ ⟨sec⟩ | ⟨empty⟩
⟨op⟩  ::= + | -
⟨p⟩   ::= A | B | C | (⟨ae⟩)
```

Lower case identifiers represent nonterminal symbols, upper case identifiers and special characters represent terminal symbols, and the vertical bar | means "or."

The correspondence between BNF and GTL syntax rules is indicated in the table given below.

BNF	GTL Equivalent	Meaning
|	ELSE	"or"
TERM	"TERM"	example of terminal symbol
⟨nonterm⟩	*NONTERM	example of nonterminal symbol
⟨empty⟩	NIL	the "empty" string of symbols

The SYMBOL FORMAT declarations corresponding to the BNF definition given above are:

```
SYMBOL FORMAT AE; [*P, *SEC]
SYMBOL FORMAT SEC; [*OP, *P, *SEC ELSE NIL]
SYMBOL FORMAT OP; ["+" ELSE "-"]
SYMBOL FORMAT P;
    ["A" ELSE "B" ELSE "C" ELSE "(", *AE, ")"]
```

Note that two or more components of a syntax rule are separated by commas. The identifiers corresponding to the BNF nonterminal symbols are called SYMBOL FORMAT identifiers (hereafter, SF identifiers); the expression contained in brackets, corresponding to the right-hand side of a BNF definition, is called a SYMBOL FORMAT expression (hereafter, SF expression). (In an actual program, all of the SF identifiers appearing in an SF expression must be previously defined; this is done by reordering the sequence of declarations, or by declaring an SF identifier FORWARD, as defined in Subsection 8.3.)

In GTL, recursive syntax definitions, such as that given for SEC, can often be replaced by equivalent "iterative" definitions. For example,

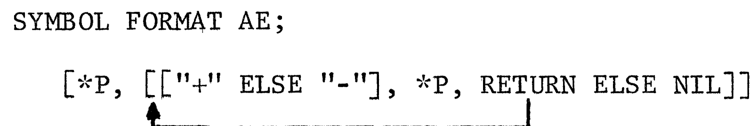
SEC may be declared as

SYMBOL FORMAT SEC; [*OP, *P, RETURN ELSE NIL]

In an SF expression, the word RETURN effectively returns control of the scanning sequence to the beginning of the SF expression; thus, SEC could scan an indefinite number of operator (OP) - primary (P) pairs.

An SF expression may contain subexpressions; an SF identifier may be replaced by its definition. For example, AE, SEC, and OP, as defined above, may be combined into one declaration:

SYMBOL FORMAT AE;



As indicated by the arrow in the above example, the word RETURN returns control to the beginning of the innermost subexpression in which it appears. The phrase RETURN START may be used in an SF expression to return control to the outermost level of an SF expression. For example, the following SF declaration could be used to scan an SF expression (as defined up to this point):

```
SYMBOL FORMAT SFX;
   [[[*TERM ELSE "[", *SFX ELSE "*", *SFID],
      [",", RETURN START ELSE NIL] ELSE
     "RETURN", ["START" ELSE NIL]]
    ["ELSE", RETURN START ELSE "]"]]
```

The undefined SF identifiers TERM and SFID are understood to match terminal symbols and SF identifiers, respectively.

All of the quoted terminal symbols appearing in SF expressions are LISP atomic symbols. The atomic symbol itself is not directly compared with the current input symbol; instead, a syntactic class number which is assigned to the atomic symbol by the programmer is compared with the syntactic class of the current input symbol. A syntactic class number may be specified directly in an SF expression. For example, [2,3,4] will match a sequence of three input symbols whose class numbers are 2, 3, and 4. This indirection allows the programmer to specify terminal symbols other than atomic symbols, such as numbers and "composite" symbols (e.g., quoted strings). Various means of assigning syntactic class numbers (both at compile time and run time) are described in paragraph 8.5.1. The class of the current input symbol is always contained in a REAL or INTEGER variable declared by the programmer. There is a separate special declaration which tells the GTL compiler that a particular variable is intended to be used as a class variable (paragraph 8.5.2). When a syntactic class number is successfully matched with the class of the current input symbol, the class variable must be reset to the class of the next input symbol. This is done automatically through a procedure supplied to the system by the programmer. Whenever a syntactic class match is made, the system calls on this procedure. It is the responsibility of the programmer to ensure that the procedure will obtain the next input symbol and assign its syntactic class number to the class variable. In the remainder of this manual, this procedure will be referred to as the "getnext" procedure. A special declaration which tells the GTL compiler the name of the intended getnext procedure must be supplied by the programmer (paragraph 8.5.3).

The SYMBOL FORMAT declaration is effectively equivalent to the declaration of a BOOLEAN procedure. The SF identifier, which may have associated formal parameters, is analogous to a BOOLEAN procedure identifier, and may be used as a Boolean primary in any GTL Boolean expression. The SF expression associated with the SF identifier is analogous to the procedure body of a BOOLEAN procedure declaration. There is also a "block" form of the declaration, consisting of the word BEGIN followed by a series of declarations, the SF expression and END. For example, the previously defined SF identifier AE could be declared by

```
SYMBOL FORMAT AE;
BEGIN LABEL L;
    [*P, L: [["+" ELSE "-"], *P, GO TO L ELSE NIL]]
END
```

Note that this example also illustrates the use of labels and statements in SF expressions.

The "semantics" of the language defined by the SF expression is determined by the inclusion of statements in the SF expression. These statements may appear anywhere in the SF expression, and each statement must be followed by a semicolon (except before an "ELSE"). For example,

```
["PRINT", ["A", PRINT A ELSE "B", PRINT B;]]
```

It is often convenient to execute a sequence of statements between a syntax class match and the call on the getnext procedure. This can be done by using the delayed getnext construct: If a colon is placed immediately after a terminal component of an SF expression then the call on the getnext procedure is delayed until another (nonterminal or terminal) matching

component, or an ELSE,], [, NIL, RETURN or label is encountered. For example, if 2 is the syntactic class of a number which is contained in the variable INREAL, and T is a REAL variable, then

```
[2: T := INREAL; "+", 2: T := T + INREAL;
    PRINT #SUM = #T;]
```

will match two numbers separated by a + and print their sum.

As an example of some of the features described so far, a set of SYMBOL FORMAT declarations which will transform an arithmetic expression into a suffix Polish string is presented below. It is assumed that syntactic class numbers of 2 and 3 have been chosen for variables and numbers, respectively. To make the SF expressions more readable, the following define declaration is used:

```
DEFINE VARIABLE = 2#, NUMBER = 3#
```

It is also assumed that the class variable and getnext procedure declarations, such as those given below, have been made.

```
REAL FIELD CDRF [33:15];
REAL CLASS;
PROCEDURE GETNEXT;
CLASS:=

CASE READCON(FALSE) OF
    BEGIN
    1; COMMENT END OF FILE CLASS;
    3; COMMENT ILLEGAL NUMBER CLASS;
    3; COMMENT NUMBER CLASS;
    CDRF(INSYM); COMMENT ATOMIC SYMBOL CLASSES;
    0; COMMENT MULTI-CHARACTER STRING CLASS;
    END
```

The READCON function is described in paragraph 9.4.4. The expression

CDRF(INSYM)

yields the last 15 bits (the CDR field) of the contents of the atomic symbol contained in INSYM (see paragraph 7.2.4). It is assumed here that the CDR fields of the atomic symbols representing variables in the arithmetic expressions to be scanned have been preset to the value two. Although specific class number assignments have been indicated for this example, it should be noted that the declarations given below contain no direct or explicit reference to a class number. In two of the declarations, AEXP and TERM, the class number of an atomic symbol is indirectly referenced in an arithmetic expression by preceding the quoted atomic symbol by an =.

```
SYMBOL FORMAT AEXP; FORWARD;
SYMBOL FORMAT PRIMARY;
     [VARIABLE: PRIN INSYM SPACE ELSE
      NUMBER: PRIN INREAL SPACE ELSE
      "(", *AEXP, ")"];
SYMBOL FORMAT FACTOR;
   [*PRIMARY, ["*", *PRIMARY, PRIN #* #; RETURN ELSE NIL]];
SYMBOL FORMAT TERM;
BEGIN BOOLEAN TIMES;
     [*FACTOR, ["⊗"*"/": TIMES := CLASS = (="⊗");
               *FACTOR, PRIN IF TIMES THEN #⊗ # ELSE # / #;
               RETURN ELSE NIL]]
END;
```

```
SYMBOL FORMAT AEXP;
BEGIN BOOLEAN MINUS;
     [*TERM, ["+" * "-": MINUS := CLASS = (="-");
              *TERM, PRIN IF MINUS THEN #- # ELSE #+ #;
              RETURN ELSE NIL]]
END
```

The composite terminal symbols, "⊗" * "/" in TERM and "+" * "-" in AEXP*, allow a match on either one of the atomic symbols (see paragraph 8.2.1). Also, since AEXP and PRIMARY call on each other indirectly, it was necessary to declare one of these SF identifiers FORWARD. If the input string consists of

```
A + (B - C) / D + E * F
```

then a call on AEXP, followed by a TERPRI, will produce the following output:

```
A B C - D / + E F * +
```

In order to give a better idea of how the GTL parsing declarations actually work, AEXP and PRIMARY are redefined in paragraph 8.7 as BOOLEAN procedures, and are also used in a program in the EXAMPLES section.

In the remaining paragraphs of this section, SYMBOL FORMAT declarations are used to define the syntax and semantics of the entire GTL syntax-directed parsing system.

---

*The arithmetic scanner listed above does not handle unary operators. See the program in Subsection 8.8 for a complete example of arithmetic expression parsing.

8.2 SYMBOL FORMAT EXPRESSIONS

8.2.1 Terminal Symbols

A simple terminal symbol matching component in an SF expression is either a number or a quoted atomic symbol. The CDR field of the quoted atomic symbol will contain a syntactic class number, which is either preset by the programmer (see paragraph 8.5.1) or made by default by the GTL compiler. The rule by which a default class assignment is made is described below.

A "composite" terminal symbol matching component may be formed with two or more class numbers and/or quoted atomic symbols separated by asterisks or equal signs. The * means that the next class number must be strictly greater than the previous class number; and the = implies strict equality. In order for a match to occur with the composite terminal symbol component, the value of the class variable must be greater than or equal to the first member and less than or equal to the last member of the sequence of class numbers or quoted atomic symbols. If default assignment occurs after an asterisk, the value assigned will be one greater than the previous class number. For example, if the CDR fields of + and - have not been previously assigned,

2 = "+" * "-"

will assign the class number 2 and 3 to "+" and "-", respectively, and a match will occur if the value of the class variable is a 2 or a 3.

As indicated in Subsection 8.1, the class number assigned to an atomic symbol used as a terminal matching component may be referenced indirectly in an arithmetic expression by preceding the quoted atomic symbol by an =. If no syntactic class assignment has been made previous to the occurrence

of the atomic symbol in an arithmetic expression context, a default class assignment will be made.

There is also an indirect terminal symbol matching component, which consists of a period followed by a REAL simple variable identifier. The value of the variable is compared with the value of the class number of the current input symbol. For example,

```
SYMBOL FORMAT LST(X); VALUE X; REAL X;
  [.X, [",", RETURN START ELSE NIL]]
```

The SYMBOL FORMAT declaration given below indicates how the GTL compiler scans terminal symbol components of SF expressions and makes default class assignments. For the purpose of this declaration, it is assumed that LITNO, ATOMICSYMBOL, REALID are previously defined identifiers representing the syntactic class numbers assigned to unsigned integers, LISP atomic symbols, and REAL variable identifiers, respectively. The values of the constants and atomic symbols are assumed to be contained in the variables INREAL and INSYM, respectively. Also, CLASS is a REAL variable used as the class variable, SFC is a real variable which is used to control the default class assignment, and FLAG is an error message procedure.

```
SYMBOL FORMAT TERMINAL;
BEGIN
BOOLEAN AST;
REAL FIELD CDRF[33:15];
REAL LAST, T;
LABEL L;
     [AST := TRUE; LAST := 0;
L:    [LITNO:  SFC := INREAL ELSE
      """", ATOMICSYMBOL:
                         IF T := CDRF(INSYM) = 0 THEN
                             BEGIN
                             IF AST THEN SFC := SFC + 1;
                             CDRF(INSYM) := SFC;
                             PRINT #DCA # SFC
                             END
                         ELSE
                             SFC := T; """" ELSE
      ".", REALID],
      IF SFC < LAST THEN FLAG(670);
      LAST := SFC;
      ["*" * "=": AST := CLASS = (="*"); GO TO L ELSE
       NIL]]
END
```

As can be seen from this declaration, every default class assignment will cause "DCA" followed by the class number to be printed during the compilation. The variable SFC is set to zero at the beginning of the compilation.

As indicated in Subsection 8.1, a call on the getnext procedure normally occurs immediately after a match. The call on the getnext procedure may be delayed for the purpose of executing a series of statements which may operate on additional information associated with the current input symbol by placing a colon immediately after the terminal symbol matching component. This construct may also be used in conjunction with the error message option described in paragraph 8.2.8.

8.2.2 Nonterminal Symbols

As defined in Subsection 8.1, a nonterminal symbol in an SF expression is an asterisk followed by an SF identifier. In addition, the asterisk may be followed by any Boolean expression; a syntax match will occur if and only if the value of the Boolean expression is TRUE. For example,

[* CLASS ≥ 2 AND CLASS ≤ 3, GETNEXT;]

has the same effect as

[2 = "+" * "-"]

(if GETNEXT is the name of the getnext procedure).

8.2.3 NIL

NIL is the matching component which matches the empty string of symbols, and is used primarily to indicate that an SF expression can be satisfied even if the preceding alternatives are not. It is usually introduced when transforming a non-deterministic syntax rule into a form acceptable to a no-backtracking parser (p. 90, Reference 5). For example,

⟨factor⟩ ::= ⟨primary⟩ * ⟨factor⟩ | ⟨primary⟩

is transformed into

⟨factor⟩ ::= ⟨primary⟩ ⟨remfact⟩

⟨remfact⟩ ::= ⟨factor⟩ | ⟨empty⟩

which is combined into the SF declaration

```
SYMBOL FORMAT FACTOR;
    [*PRIMARY, ["*", *FACTOR ELSE NIL]];
```

NIL may also appear among a series of statements following a delayed getnext for the purpose of forcing the call on the getnext procedure.

8.2.4 <u>Statements</u>

Any GTL statement may be included in an SF expression. The statement must be followed by a semicolon except before an ELSE. The use of identifiers in statements which are part of SYMBOL FORMAT declarations are subject to the same restrictions as those in procedure declarations. A sequence of one or more statements may be used in place of NIL to match the empty string of symbols; for example,

```
["RETURN", ["START", BV := TRUE ELSE BV := FALSE]]
```

8.2.5 <u>Labels</u>

Labels may appear anywhere in the SF expression. Each label must be followed by a colon. An example of the use of a label for simplifying an SF expression is given in paragraph 8.2.9.

8.2.6 RETURN

The RETURN part of an SF expression causes transfer of control to the beginning of an SF expression, as described in Subsection 8.1. An optional UNTIL part allows the programmer to limit the number of possible iterations; for example,

```
["C", J := 0; [["A" ELSE "D"], RETURN UNTIL (J := J + 1) = 10 ELSE "R"]]
```

The syntax of the RETURN part is defined below.

```
SYMBOL FORMAT RETRN;
  ["RETURN", ["START" ELSE NIL], ["UNTIL", *BEXP ELSE NIL]]
```

BEXP represents an SF identifier which defines the syntax of Boolean expressions. When the UNTIL option is used in the RETURN part of the SF expression, transfer of control will continue until the value of the Boolean expression is TRUE.

8.2.7 The SWITCH Option

When the first components of a series of alternative rules in an SF expression are all consecutively numbered terminal components, and when the number of alternative rules is relatively large, the scanning speed can be substantially increased by using the SWITCH option. The SWITCH option is effected by placing the word SWITCH immediately after the [in the SF expression. The code generated is similar to that produced by the declaration of an ALGOL switch although the meaning of the remainder of the SF expression remains unchanged. For example, if LABELID, REALID, and INTID represent class numbers of label and real and integer variable identifiers, respectively, then

```
SYMBOL FORMAT STMT;
   [SWITCH
          LABELID: LABELR; ":", RETURN ELSE
          REALID*INTID: VARIABLE(FS) ELSE
          "WHILE", WHILESTMT ELSE
          "DO", DOSTMT ELSE
          "FOR", FORSTMT ELSE
          "READ", READSTMT ELSE
          "WRITE", WRITESTMT]
```

indicates how a subset of labeled ALGOL statements might be scanned. Any resemblance between STMT and the STMT procedure of the B 5500 ALGOL compiler is not coincidental.

8.2.8 The Error Message Option

If the delayed getnext option (paragraph 8.2.1) is used, the colon following the terminal component may be followed by a number or by a statement preceded by an asterisk for the purpose of generating an error message if the terminal symbol is not matched. If a number is given, there will be an implicit call on an error message procedure supplied by the programmer with that number as its argument. For example, given the procedure

```
PROCEDURE ERR(X); VALUE X; REAL X;
     PRINT #ERROR NUMBER # X
```

the SF expressions

```
["(", *AEXP, ")": 104;]
  ["(", *AEXP, ")": *PRINT #MISSING )#;]
```

would generate the following error messages if the ")" were missing from the input text:

```
ERROR NUMBER 104
MISSING )
```

8.2.9 Syntax and Semantics of SYMBOL FORMAT Expressions

The complete syntax of SF expressions is defined by the following SF declarations:

```
SYMBOL FORMAT SFEXP; ["[", *SFXP];
SYMBOL FORMAT SFXP; [["SWITCH" ELSE NIL], *SFX];
SYMBOL FORMAT SFX;
BEGIN
LABEL L;
    [[*TERMINAL, [":", [LITNO ELSE "*", *STMT ELSE *STMT],
                        ";", RETURN START ELSE
                  GO TO L] ELSE
     ["*", *BEXP ELSE "NIL" ELSE "[", *SFXP],
L:                ["," , RETURN START ELSE NIL] ELSE
     ["RETURN",  ["START" ELSE NIL],
                  ["UNTIL", *BEXP ELSE NIL] ELSE
     LABELID, ":", RETURN START ELSE
    *STMT], [";", RETURN START ELSE NIL],
                  ["ELSE", RETURN START ELSE"]"]]
END
```

The SF identifiers TERMINAL, BEXP, and STMT are intended to match terminal components (paragraph 8.2.1), Boolean expressions and statements, respectively.

The identifier LITNO represents the class number of an unsigned integer, and LABELID represents the class number of label identifiers.

The value assigned to an SF identifier is TRUE if one of the alternative rules in the SF expressions is satisfied, and is FALSE otherwise. The actual REAL equivalent of these Boolean values can be one of three values as indicated in the table below.

Value of SF identifier	Equivalent Value	Meaning
TRUE	Boolean (1)	syntax OK
FALSE	Boolean (0)	syntax not satisfied; getnext procedure not called
FALSE	Boolean (2)	syntax not satisfied; getnext procedure called

As an example of how the two different "FALSE" values can arise, consider the declaration of the SF identifier TERMINAL given in paragraph 8.2.1. If the class number of the current input symbol does not match the class numbers assigned to LITNO, quote, or period, then the value of TERMINAL will be BOOLEAN (0) (FALSE); however, if period is matched and REALID is not matched then the value of TERMINAL will be BOOLEAN (2) (also FALSE), since a call on the getnext procedure occurred after matching the period. The two different FALSE values of an SF identifier can make a difference in the evaluation of an SF expression in which it appears. For example, referring to the declaration of SFX given above, if the value of TERMINAL is BOOLEAN (0), then the getnext procedure has not been called, and it makes sense to test the next two terminal components (* and [).

However, if the value of TERMINAL is BOOLEAN (2), the getnext procedure has been called, and since the GTL system provides no backtracking facility, the current input symbol cannot be restored to its previous

value, the remaining alternatives are not tested and the value of SFX will be BOOLEAN (2).

In general, the following rule is used in testing the value of a non-terminal component (an asterisk followed by a Boolean expression) in an SF expression: If the value of the Boolean expression is TRUE (the REAL equivalent is an odd number), the next component in the rule is tested. If all of the components of a rule are satisfied, then the value of the associated SF identifier will be BOOLEAN (1) (TRUE). If the REAL value of the Boolean expression is zero and it is the first component of a rule, then the first component of the next alternative rule following the ELSE will be tested; if no alternative rules remain, then the value of the associated SF identifier will be BOOLEAN (0) (since the getnext procedure was not called.) In all other cases, including the failure of a terminal component which is not the first component of a rule, the evaluation of the SF expression is immediately halted, and the value of the associated SF identifier will be set to BOOLEAN (2).

8.3 SYMBOL FORMAT DECLARATIONS

The syntax of SYMBOL FORMAT declarations is defined by the following SYMBOL FORMAT declaration:

```
SYMBOL FORMAT SFDECLARATION;
  ["SYMBOL", "FORMAT", *IDENTIFIER,
       [*FORMALPARAPART ELSE NIL], ";",
       ["BEGIN",*DECLARATIONS, *SFEXP, "END" ELSE *SFEXP]]
```

where IDENTIFIER matches the SF identifier to be declared, FORMALPARAPART scans the formal parameter part of the declaration, such as might occur in

a BOOLEAN procedure declaration, and DECLARATIONS scans a series of GTL declarations, separated by semicolons. SFEXP is defined in paragraph 8.2.9 above.

In addition, SYMBOL FORMAT formal parameter declarations are allowed; for example,

```
PROCEDURE TEST(X); SYMBOL FORMAT X;
     PRINT IF X THEN #SYNTAX OK# ELSE #SYNTAX ERROR#;
SYMBOL FORMAT LST(Y); SYMBOL FORMAT Y;
   [*Y, [";", RETURN START ELSE NIL]]
                    .
                    .
                    .
TEST(SFD);
IF LST(SFD) THEN ....
```

The actual parameter corresponding to a SYMBOL FORMAT formal parameter must be a SYMBOL FORMAT identifier which itself has no arguments (there are no formal parameters specified in the SYMBOL FORMAT declaration).

SYMBOL FORMAT forward declarations have the same meaning and are made in the same form as forward procedure declarations (paragraph 9-106, Reference 4); for example,

```
SYMBOL FORMAT SFD; FORWARD;
SYMBOL FORMAT LST(Y); SYMBOL FORMAT Y; FORWARD;
```

8.4 SYMBOL FORMAT STATEMENTS

An SF expression may be used as a statement if a colon followed by a label is placed immediately after the last]; for example,

```
["[", *SFXP, ":", LABELID] : SYNTAXERROR
```

If a syntax error occurs during the execution of the SF expression, a branch is made to the specified local label; otherwise, control continues in sequence.

8.5 SYMBOL FORMAT AUXILIARY DECLARATIONS

All of the SYMBOL FORMAT auxiliary declarations must occur in the outermost block of the program, and, with the exception of the trace declaration (paragraph 8.5.5), must precede the declarations of SF identifiers.

8.5.1 Syntactic Class Declaration

The syntactic class declaration provides a convenient means of assigning class numbers to the CDR fields of quoted atomic symbols, and to previously undefined identifiers. Its effect is similar to the default class assignment described in paragraph 8.2.1. The declaration has the form

SYMBOL FORMAT * classdec

where classdec represents a sequence of constants, quoted atomic symbols, and previously undefined identifiers. The syntax of classdec is defined as follows:

```
SYMBOL FORMAT CLASSDEC;

BEGIN

REAL FIELD CDRF [33:15];

REAL T, FIRST, LAST;

BOOLEAN FST, AST;

LABEL L;

     [AST := FST := TRUE;

L:     [LITNO: SFC := INREAL ELSE

       """", ATOMICSYMBOL:

                              IF T := CDRF(INSYM) = 0 THEN

                                 BEGIN

                                 IF AST THEN SFC := SFC + 1;

                                 CDRF(INSYM) := SFC

                                 END

                              ELSE

                                 SFC := T; """" ELSE

     UNDEFINEDID: IF AST THEN SFC := SFC +1;

                  ENTER(DEFINEDID, SFC)],

     IF FST THEN

        BEGIN FST := FALSE; FIRST := LAST := SFC END

     ELSE

     IF SFC < LAST THEN FLAG(670);

     SFC := LAST;

     ["*" * "=": AST := CLASS = (="*"); GO TO L ELSE

     NIL],

     PRINT #RANGE # FIRST # TO # LAST]

END
```

The identifiers LITNO, ATOMICSYMBOL, INREAL, INSYM, and SFC have the same meanings as defined for the SF declaration given in paragraph 8.2.1. UNDEFINEDID is the class number assigned to a previously undefined identifier, and ENTER is assumed to be the name of a procedure which assigns the class number DEFINEDID to the current symbol and places SFC in additional info, linked through the CDR field. Note that the range of syntactic class assignments is printed at the end of the declaration.

The quantity placed in the CDR field of the atomic symbol is the number itself, and not a link to an atom representing the class number (i.e., the class number is not represented as an atomic number). Thus, in a GTL parsing program, the CDR field of an atomic symbol may contain two different data types: a class number and a reference value. These two data types are usually distinguished by their magnitude. One method of obtaining the REAL value of the CDR field contents is the use of the CTSM transfer function (see paragraph 6.15.1).

CTSM(<u>sexp</u>).[33:15]

where <u>sexp</u> represents a SYMBOL expression whose value should be an atomic symbol. Another, sometimes more convenient, method is the REAL valued field designator (see paragraph 7.2.2); for example, given the field declaration

REAL FIELD CDRF [33:15]

the value of the field designator

CDRF(<u>sexp</u>)

will be the REAL value contents of the CDR field.

If it is assumed that the CDR field of an atomic symbol will not be used to reference an atom, then the distinction between the two data types can be easily made. The addresses 0 to 63 are reserved for the 64 single character atoms, so that the address of a quoted multi-character atomic symbol (created at compile time) will be greater than 63. Each multi-character atomic symbol requires two or more words. Thus, a safe lower bound for the maximum class number is twice the number of atomic symbols appearing in the program plus 63. In most cases, the range of class values will be found to be adequate; if not, the address of the first available record may be reset to one greater than the maximum class number by means of the assignment statement described in paragraph 7.4.4. The following example illustrates a possible method of class variable assignment in the getnext procedure. The class variable is CLASS, and the SYMBOL variable INFO is understood to be a reference to additional information (including the class number).

```
IF CLASS := CDRF(INSYM) ≤ CLASSMAX THEN
     INFO := NIL
ELSE
     CLASS := CLASSF(INFO := ATSM(CLASS))
```

where ATSM is the Arithmetic To SyMbol transfer function and CLASSF is a field identifier referencing a predefined REAL-valued class field (Section VII), and CLASSMAX is a defined identifier representing the the maximum class number. If automatic storage reclamation is used, then the maximum class number is limited to 63, since the LISP garbage collector expects that all reference-valued LISP fields will contain an actual

reference value. In most translator applications, however, atuomatic storage reclamation has been found to be unnecessary, even in quite large cases.

8.5.2 Class Variable Declaration

The programmer indicates to the GTL compiler which variable is to be used as the class variable by the following declaration:

SYMBOL FORMAT * class variable

For example,

REAL CLASS; SYMBOL FORMAT * CLASS

The class variable must be of type REAL or INTEGER and must have been previously declared.

8.5.3 Getnext Procedure Declaration

The programmer indicates to the GTL compiler the name of the procedure to be used as the getnext procedure by the following declaration:

SYMBOL FORMAT * getnext procedure

For example, using the procedure declared in Subsection 8.1,

SYMBOL FORMAT * GETNEXT

The getnext procedure must have been previously declared and must have no formal parameters.

8.5.4 Error Procedure

The programmer indicates to the GTL compiler the name of the procedure to be used to generate error messages with the following declaration:

SYMBOL FORMAT * error procedure

For example, using the procedure declared in paragraph 8.2.8,

SYMBOL FORMAT * ERR

The error message procedure must have been previously declared and must have one formal parameter of type REAL called by value.

8.5.5 The Trace Option

The sequence of SF identifiers executed during a scan, and their REAL equivalent values, can be traced if an optional Boolean trace variable supplied by the programmer is set to TRUE; the name of the trace variable may be specified to the GTL compiler by the declaration

SYMBOL FORMAT * trace variable

For example,

BOOLEAN TRACE; SYMBOL FORMAT * TRACE

If any SF declarations precede the trace declaration, they will not be traced. It is recommended that the trace option be used for debugging or experimental purposes only, due to the additional code generated.

8.6 RECOMMENDED PRACTICES

The association of additional information with an atomic symbol, such as run-time class number assignment, can usually be most easily accomplished with the constructs provided by the GTL plex processing system (Section VII).

It is important to remember that the class variable should contain the class number of the current input symbol before an SF expression is executed. This usually means that the getnext procedure should be executed once at the beginning of the program before executing any SF expressions.

It should be noted from paragraph 8.5.1 that, by using defined identifiers representing syntactic class numbers, instead of making explicit reference to the numbers themselves, it would be possible to insert additional syntactic categories without the necessity of making any compensating changes in the remainder of the program. Also, some attention should be given to the ordering of the syntax classes so as to make optimal use of the SWITCH option described in paragraph 8.2.7.

When constructing the getnext procedure, it should be noted that every GTL read function, with the exception of the SCAN function, will ordinarily read a signed number as one item. It is often desirable to be able to read a number and its associated sign (a + or - immediately preceding the number) separate, as for example, would be required when parsing arithmetic expressions. To do so requires the use of the appropriate form of the INPUT statement containing the sign separation option (see paragraph 9.5.4).

8.7 BOOLEAN PROCEDURE EQUIVALENT OF SYMBOL FORMAT DECLARATION

The following BOOLEAN procedure declarations are (effectively) equivalent to the SYMBOL FORMAT declarations of AEXP and PRIMARY given in Subsection 8.1

```
BOOLEAN PROCEDURE AEXP;
     BEGIN
     LABEL LR, LFN, EXIT;
     BOOLEAN MINUS;
          IF TERM THAN
               BEGIN
     LR:       IF CLASS ≥ (="+") AND CLASS ≤ (="-") THEN
                    BEGIN
                    MINUS := CLASS = (="-");
                    GETNEXT;
                    IF TERM THEN
                         BEGIN
                         PRIN IF MINUS THEN #- # ELSE #+ #;
                         GO TO LR
                         END
                    ELSE
                         GO TO LFN;
                    END;
                    AEXP := TRUE;
                    GO TO EXIT;
               END
          ELSE
     LFN:      AEXP := BOOLEAN(2);
     EXIT:
     END OF AEXP
```

```
BOOLEAN PROCEDURE PRIMARY;
     BEGIN
     LABEL LFN, EXIT;
          IF CLASS = VARIABLE THEN
               BEGIN
               PRIN INSYM SPACE; GETNEXT
               END
          ELSE
          IF CLASS = NUMBER THEN
               BEGIN
               PRIN INREAL SPACE; GETNEXT
               END
          ELSE
          IF CLASS = (="(") THEN
               BEGIN
               GETNEXT;
               IF AEXP THEN
                    IF CLASS = (=")") THEN
                         GETNEXT
                    ELSE
                         GO TO LFN
               ELSE
                    GO TO LFN
               END
          ELSE
               BEGIN
               AEXP := FALSE; GO TO EXIT
               END;
          AEXP := TRUE; GO TO EXIT;
     LFN: AEXP := BOOLEAN(2);
     EXIT:
     END OF PRIMARY
```

8.8 EXAMPLE PROGRAM

The program listed on the following pages was designed to illustrate the use of most of the constructs described in this section. The program accepts ALGOL-like function definitions from a remote terminal and compiles them into a simple interpreter language. After compilation, a function may then be evaluated to produce its graph on the remote terminal. Compilation takes place while the function is typed in (line by line); if a syntax error is detected, the compiler attempts to recover so that compilation can continue. A simple example of the program's operation is given at the end of the listing.

It may be of interest to note that the organization of the compiler resembles that of the B 5500 ALGOL compiler, and that the interpreter language resembles, in some respects, the B 5500 machine language. In effect, the compiler is a miniature version of an ALGOL compiler. It might also be noted that the interpreter itself was implemented with the help of SYMBOL FORMAT declarations.

In order to understand the operation of the program, the programmer should be familiar with the GTL list processing, record processing, string processing, and Input-Output systems as described in Sections VI, VII, V, and IX, respectively.

```
BEGIN SYMBOL FLEX;
%
%       THIS A IS REMOTE TERMINAL PLOTTER PROGRAM.
%       ----------------------------------------
%
%
%       IT ACCEPTS ALGOL-LIKE FUNCTION DEFINITIONS FROM A REMOTE
%       TERMINAL AND COMPILES THEM INTO A SIMPLE INTERPRETER
%       LANGUAGE.  AFTER COMPILATION, A FUNCTION MAY THEN BE
%       EVALUATED TO PRODUCE IT"S GRAPH ON THE REMOTE  TERMINAL.
%       COMPILATION TAKES PLACE WHILE THE FUNCTION IS TYPED IN
%       (LINE BY LINE); IF A SYNTAX ERROR IS DETECTED, THE
%       COMPILER ATTEMPTS TO RECOVER SO THAT COMPILATION CAN
%       CONTINUE.  THE SYNATX OF THE INTERPRETER LANGUAGE IS GIVEN
%       IN BACKUS NORMAL FORM OR BNF, FOLLOWED BY AN OPTIONAL
%       SEMANTIC DESCRIPTION.
%
%       IT IS ASSUMED THAT THE PERSON ATTEMPTING TO USE THIS
%       PROGRAM KNOWS ALGOL WELL ENOUGH TO HAVE WRITTEN TWO OR
%       THREE PROGRAMS ALREADY, THERFORE THE DESCRIPTION IS FOR
%       RELATING THE SYNTAX OF THE PROGRAM TO THE USER.  A TOP-TO-
%       BOTTOM APPROACH OF THE SYNTAX WAS CHOOSEN, SINCE THE
%       INTERPRETER ITSELF IS WRITTEN TOP-TO-BOTTOM.
%
%PLOTTER COMMAND LANGUAGE.
%       SYNTAX.
%               <PLOTTER COMMAND LANGUAGE> ::= FUNCTION <FUNCTIONDEC>
%                       / LIST / PLOT <PLOTTER> / DELETE / STOP
%       SEMANTICS.
%               "FUNCTION" ALLOWS THE DECLARATION OF AN ALGOL-LIKE
%               PROCEDURE DECLARATION, WHICH, WHEN PASSED PARAMETERS
%               WILL RETURN VALUES TO BE PLOTTED.
%               THE DESCRIPTION OF THE FUNCTION IS THEN TYPED IN
%               LINE BY LINE.  THE COMPILER WILL TYPE LINE NUMBERS
%               AS COMPILATION PROCEDES.
%               NOTE THAT FUNCTIONS MAY RECURSE, CALL EACH OTHER,
%               ETC....
%
%               "LIST" WILL LIST ALL FUNCTIONS DECLARED.
%
%               "PLOT" PLOTS THE GIVEN FUNCTION, WHICH MAY INVOLVE
%               CALLS ON OTHER FUNCTIONS.  SEE THE INFORMATION UNDER
%               PLOTTER.
%
%               "DELETE" DELETES THE FIRST PROCEDURE THAT THE "LIST"
%               COMMAND LISTS.  THUS IF TWO OR MORE FUNCTIONS ARE TO
%               BE DELETED, "DELETE" WILL HAVE TO ENTERED SEVERAL
%               TIMES.  THE LIST IS STRUCTURED SO THAT THE MOST
%               RECENT DECLARATION IS FIRST, WITH THE END OF THE
%               LIST BEING THE FIRST FUNCTION ENTERED.
```

```
%            "STOP" CAUSES THE PROGRAM TO GO TO END-OF-JOB.
%
%            AFTER COMPLETING A COMMAND, EXCEPT THE "STOP" COMMAND,
%            THE PROGRAM TYPES "GO AHEAD" TO SIGNIFY THAT IT IS
%            READY FOR ANOTHER COMMAND.
%
%
%FUNCTION DECLARATION.
%      SYNTAX.
%            <FUNCTIONDEC> ::= <FUNCTION HEADING> <FUNCTION BODY>
%            <FUNCTION HEADING> ::= <FUNCTION IDENTIFIER>
%                  <FORMAL PARAMTER PART>
%            <FORMAL PARAMETER PART> ::= <EMPTY> / ( <FORMAL
%                  PARAMTER LIST> )
%            <FORMAL PARAMETER LIST> ::= <IDENTIFIER> /
%                  <FORMAL PARAMETER LIST> , <IDENTIFIER>
%            <PROCEDURE BODY> ::= BEGIN <DECLARATIONS>
%                  <COMPOUNDTAIL> / <STATEMENT>
%      SEMANTICS.
%            THE FUNCTION DECLARATION ALLOWS A "REAL" FUNCTION
%            TO BE DECLARED WITH OPTIONAL FORMAL PARAMETERS, WHICH
%            ARE BY DEFAULT OF TYPE "REAL", FOLLOWED BY EITHER A
%            BLOCK OR A STATEMENT.
%
%DECLARATIONS.
%      SYNTAX.
%            <DECLARATION> ::= <TYPE DECLARATION> /
%                  <DECLARATION> ; <TYPE DECLARATION>
%            <TYPE DECLARATION> ::= <TYPE> <TYPE LIST>
%            <TYPE> ::= LABEL / REAL / INTEGER / BOOLEAN
%            <TYPE LIST> ::= <IDENTIFIER> / <TYPE LIST> , <IDENTIFIER>
%      SEMANTICS.
%            DECLARATIONS ALLOW LOCAL VARIABLES TO BE DECLARED FOR
%            USE IN THE CURRENT FUNCTION DECLARATION.  ONLY
%            PREVIOUSLY DECLARED FUNCTIONS ARE ALLOWED AS
%            GLOBAL QUANITIES.
%
%STATEMENT.
%      SYNTAX.
%            <STATEMENT>  ::= LABEL : <STATEMENT> /
%                  <ASSIGNMENT STATEMENT> / GO TO LABEL / <EMPTY> /
%                  <CONDITIONAL STATEMENT> / <WHILE STATEMENT> /
%                  <DO STATEMENT> / RETURN <AEXP> / BEGIN
%                  <COMPOUNDTAIL>
%            <COMPOUNDTAIL> ::= <STATEMENT> END /
%                  <STATEMENT> ; <COMPOUNDTAIL>
%      SEMANTICS.
%            THE BASIC CONSTITUENTS OF THE INTERPRETER
%            LANGUAGE ARE STATEMENTS.   THESE STATEMENTS ARE VERY
%            SIMILAR TO ALGOL STATEMENTS.
```

```
%
%ASSIGNMENT STATEMENT.
%       SYNTAX.
%               <ASSIGNMENT STATEMENT> ::= <LEFT PART LIST> <AEXP>
%               <LEFT PART LIST> ::= <VARIABLE> := / <VARIABLE> :=
%                       <LEFT PART LIST>
%       SEMANTICS.
%               THE ASSIGNMENT STATEMENT CAUSES THE VAULE
%               REPRESENTED BY THE ARITHMETIC EXPRESSION (AEXP)
%               TO BE ASSIGNED TO THE VARIABLES APPEARING ON THE
%               LEFT OF EACH ASSIGNMENT SYMBOL.
%
%CONDITIONAL STATEMENT.
%       SYNTAX.
%               <CONDITIONAL STATEMENT>  ::= <IF CLAUSE> <STATEMENT>  /
%                       <IF CLAUSE> <STATEMENT> ELSE <STATEMENT>
%               <IF CLAUSE> ::= IF <BEXP> THEN <STATEMENT>
%       SEMANTICS.
%               CONDITIONAL STATEMENTS PROVIDE A MEANS WHEREBY THE
%               THE EXECUTION OF A STATEMENT, OR A SERIES OF
%               STATEMENTS, IS DEPENDENT UPON THE LOGICAL VALUE
%               PRODUCED BY A "BOOLEAN" EXPRESSION (BEXP).
%
%WHILE STATEMENT.
%       SYNTAX.
%               <WHILE STATEMENT> ::= WHILE <BEXP> DO <STATEMENT>
%       SEMANTICS.
%               THE "WHILE" STATEMENT PROVIDES A METHOD OF CONTROLLING
%               AN ITERATIVE PROCESS IN WHICH EXIT FROM THE LOOP
%               DEPENDS ON EXCEEDING A LIMIT.  THE "BOOLEAN" EXPRESSION
%               IS FIRST TESTED; THE FOLLOWING STATEMENT IS THEN
%               EXECUTED AS LONG AS THE BOOLEAN EXPRESSION IS "TRUE".
%
%DO STATEMENT.
%       SYNTAX.
%               <DO STATEMENT> ::= DO <STATEMENT> UNTIL <BEXP>
%       SEMANTICS.
%               THE "DO" STATEMENT PROVIDES A METHOD OF CONTROLLING
%               AN ITERATIVE PROCESS IN WHICH EXIT FROM THE LOOP
%               DEPENDS ON REACHING A LIMIT.  THE STATEMENT IS
%               FIRST EXECUTED THEN THE TEST IS MADE, AND THE
%               EXECUTION OF THE STATEMENT IS REPEATED AS LONG AS THE
%               "BOOLEAN" EXPRESSION IS "FALSE".  THIS IS VERY
%               SIMILAR TO A FORTRAN "DO" LOOP.
%
%ARITHEMTIC EXPRESSIONS (AEXP) AND BOOLEAN EXPREXSIONS (BEXP).
%       SEMANTICS.
%               THESE EXPRESSIONS ARE IDENTICAL TO THEIR ALGOL
%               COUNTERPARTS, WITH THE RESTRICTION THAT STRINGS
%               ARENNT ALLOWED.  CONSULT THE ALGOL MANUAL.
```

```
%
%STANDARD FUNCTIONS.
%      THE STANDARD OR INTRINSIC FUNCTIONS ARE LISTED BELOW
%      WITH APPROPRIATE DEFINITIONS.  GIVEN THAT AE IS AN <AEXP>, THEN:
%
%      ABS(AE)      PRODUCES ABSOLUTE VALUE OF AE
%      SIN(AE)      PRODUCES SINE OF AE
%      COS(AE)      PRODUCES THE COSINE OF AE
%      EXP(AE)      PRODUCES THE EXPONENTIAL FUNCTION OF AE
%      LN(AE)       PRODUCES THE NATURAL LOGARITHM OF AE
%      SQRT(AE)     PRODUCES THE SQUARE ROOT OF AE
%
%TYPE TRANSFER FUNCTIONS.
%      THE TYPE TRANSFER FUNCTIONS ARE LISTED BELOW:
%
%      REAL.
%            THE FUNCTION "REAL(BE)" YIELDS A VALUE OF TYPE
%            "REAL" FROM A BOOLEAN EXPRESSION.  THIS ALLOWS
%            ARITHMETIC OPERATIONS TO BE CARRIED OUT ON
%            BOOLEAN QUANITITES BUT DOES NOT ALTER THEIR
%            INTERNAL SYSTEM REPRESENTATION.
%      BOOLEAN.
%            THE FUNCTION "BOOLEAN(AE)" YIELDS A VALUE OF TYPE
%            "BOOLEAN" FROM AN ARITHMETIC EXPRESSION.  THIS ALLOWS
%            BOOLEAN OPERATIONS TO BE CARRIED OUT ON ARITHMETIC
%            QUANTITES BUT DOES NOT ALTER THEIR INTERNAL SYSTEM
%            REPRESENTATION.
%
%

%
%
%PLOTTER SECTION
%
%      SEMANTICS.
%            THE "PLOT" COMMAND IS FOLLOWED BY THE FUNCTION
%            IDENTIFIER TO BE PLOTTED.  IT MUST HAVE BEEN DECLARED
%            WITH AT LEAST ONE PARAMETER  WHICH WILL BE USED IN THE
%            PLOTTING.  IF THE FUNCTION HAS MORE THAN ONE PARAMETER THE
%            PROGRAM WILL REQUEST THEIR VALUES.
%
%            THE PROGRAM WILL THEN ASK FOR THE
%            BEGINNING POSITION, INCREMENT, AND FINAL VALUE FOR THE
%            RANGE OF THE PLOT.
%
%            THE PLOT OF THE FUNCTION WILL BE TYPED ON THE
%            TERMINAL AND THEN THE PROGRAM WILL RETURN
%            TO COMMAND MODE.
%
%
%
```

```
%   OUTLINE OF REMOTE PLOTTER PROGRAM
%   ALL OF THE PROCEDURES USED AND THEIR FUNCTIONS ARE
%    DESCRIBED BELOW
%
%   THE GETNEXT AND ERROR PROCEDURES
%             GETNEXT          THE "GETNEXT" PROCEDURE
%             ERROR            THE "ERROR" PROCEDURE
%   MISC. ROUTINES FOR ACCESSING CODE STRING
%             PUT              PLACES CHARACTER IN CODE STRING
%             PUTADR           PUTS ADDRESS IN CODE STRING
%             GET              GETS CHARACTER FROM CODE STRING
%             GETADR           GETS ADDRESS FROM CODE STRING
%             EMIT             EMITS ONE INSTRUCTION
%             EMITADR          EMITS ADDRESS OF INSTRUCTION
%             EMITNUM          EMITS NUMBERS
%   EXPRESSION SCANNERS
%             IFEXP            HANDLES CONDITIONAL EXPRESSIONS
%             VARIABLE         COMPILES VARIABLES & ASSIGNMENTS
%             PANA             PARENTHESIS AND ARITHMETIC EXPRES
%             PRIMARY          COMPILES ARITHMETIC PRIMARIES
%             FACTOR           COMPILES ARITHMETIC FACTORS
%             TERM             COMPILES ARITHMETIC TERMS
%             AEXP             COMPILES ARITHMETIC EXPRESSIONS
%             BOOPRIM          COMPILES BOOLEAN PRIMARIES
%             BOOSEC           COMPILES BOOLEAN TERMS
%             EXPRESS          COMPILES BOTH ARITHMETIC AND
%                              BOOLEAN EXPRESSIONS
%             BEXP             COMPILES BOOLEAN EXPRESSIONS
%   STATEMENT SCANNERS
%             COMPOUNDTAIL     TAIL END OF COMPOUND STATEMENT
%             STMT             SCANS SOME STATEMENT BEGINNERS
%             RESETLABELS      RESETS FORWARD LABEL REFERENCES
%                              WHEN "UNCOMPILING" (RECOVERING
%                              FROM ERRORS)
%             STATEMENT        COMPILES ALL STATEMENTS & HANDLES
%                              RECOVERY FROM ERRORS IN STATEMENTS
%   DECLARATION SCANNERS
%             ENTRY            RUN TIME SYNTAX CLASS ASSIGNMENT
%             ENTER            APPLIES ENTRY TO LIST OF
%                              IDENTIFIERS
%             DECLARATION      HANDLES DECLARATIONS
%             PURGE            REMOVES ATOMIC SYMBOLS FROM
%                              OBJECT LIST WHEN LEAVING THE
%                              BLOCK AND CHECKS FOR MISSING
%                              LABELS
%             DECLARE          HANDLES SERIES OF DECLARATIONS
%                              AND PROVIDES FOR RECOVERY FROM
%                              SYNTAX ERRORS
%
%
```

```
%          DUMPCODE          TRANSFERS CODE FROM CODE STRING
%                            TO THE LISP "STACK" AT A POINT
%                            IMMEDIATELY FOLLOWING THE INFO
%                            WORD OF THE ATOMIC SYMBOL
%                            REPRESENTING THE FUNCTION
%          PROCEDUREDEC      HANDLES DECLARATIONS OF FUNCTIONS
%                            TO BE PLOTTED
%   INTERPRETER SECTION
%          MKADR             REMOVES 2 CHARACTERS FROM CODE
%                            STRING TO BE USED AS AN ADDRESS
%          INTERPRET         INTERPRETS THE CODE STRING
%          EXECUTE           MAKES CALLS ON FUNCTIONS; USES
%                            INTERPRET
%          PLOTTER           PLOTS FUNCTION EXECUTED BY
%                            INTERPRETER
      %  REMOTE TERMINAL FILE DECLARATION
FILE REMOTE;
BOOLEAN
      NEWLINE,            %  TELLS GETNEXT TO GET NEW LINE
                          %  WHEN TRUE
      COMPILING,          %  TELLS GETNEXT TO PRINT LINE
                          %  NUMBERS
      FORMTOG,            %  SET TO TRUE WHEN ASSIGNING
                          %  CLASSES TO FORMAL PARAMETERS OF
                          %  A FUNCTION
      NOLABEL,            %  IS SET TO TRUE BY PURGE IF A
                          %  DECLARED LABEL IS NOT USED
      INTERPRETING;       %  TELLS GETNEXT THAT NEXT "SYMBOL"
                          %  IS TO COME FROM PRECOMPILED
                          %  CODE AND "CLASS" IS TO BE THE
                          %  NEXT INSTRUCTION
REAL
      CMAX,               %  IS THE NUMBER OF NON-INTEGER
                          %  CONSTANTS OCCURING IN A FUNCTION
                          %  DECLARATION
      COUNT,              %  COUNTS NUMBER OF IDENTIFIERS
                          %  ENTERED BY ENTRY
      CLASS,              %  THE SYNTAX CLASS VARIABLE
      DL,                 %  TEMP VARIABLE USED AT END OF
                          %  PROGRAM
      FR,                 %  USED LIKE B5500 "F" REGISTER
      SAVEI,              %  SAVES LOCATION IN "STACK" OF
                          %  POINT WHERE FUNCTION VALUE IS
                          %  TO BE RETURNED
      I,                  %  INDEX TO "STACK"
      WHL,                %  THE WHOLE INFORMATION WORD
                          %  ASSOCIATED WITH EACH DECLARED
                          %  IDENTIFIER - SET BY GETNEXT
      ADDRESS;            %  ADDRESS OF DECLARED IDENTIFIER -
                          %  SET BY GETNEXT
```

```
INTEGER
      LINENO,             %  LINE NUMBER PRINTED BEFORE EACH
                          %  LINE OF FUNCTION DECLARATION
      L,                  %  RELATIVE LOCATION OF CHARACTER
                          %  IN CODE STRING
      J;                  %  TEMP VARIABLE
SYMBOL
      BASE,               %  POINTER TO FIRST WORD FOLLOWING INFO WORD
                          %  ASSOCIATED WITH A FUNCTION IDENTIFIER
                          %  START OF FUNCTION CODE STRING
      PLIST,              %  LIST OF IDENTIFIERS CREATED BY
                          %  ENTRY
      INF,                %  POINTER TO INFO WORD CREATED BY
                          %  ENTRY
      INFO;               %  POINTER TO INFO WORD - SET BY
                          %  GETNEXT
ARRAY
      STACK[0:99],        %  THE "STACK" USED BY THE
                          %  INTERPRETER
      CONSTANT,           %  TABLE OF NON-INTEGER CONSTANTS
                          %  APPEARING IN FUNCTION DECLARATION
      CONSTADR[0:127];%   LINKS TO ADDRESS PART OF OF
                          %  "RCN" INSTRUCTION
STRING ARRAY STR[0:99](8);         % CODE STRING USED DURING COMPILATION
LABEL START, RESTART, EXIT;
DEFINE
      BUMPL = L := L + 2#,
      CLSS  =[33:15]#,
      ADRS  = [18:15]#,
      LINK  = [3:15]#,
      LINKF = NPAR#,
      CLASSF = CORF#,
      NPARAM = LINK#,
      ATYPE = 1#,
      BTYPE = 2#;
STRING FIELD STRF [0:8];
REAL FIELD CORF [33:15],
           ADDRESSF [18:15],
           NPAR [3:15],
           WH;
SYMBOL FORMAT  % THE FOLLOWING ARE SYNTACTIC CLASS ASSIGNMENTS
      * "RANDOM"    * "ABS"        * "SIN"        * "COS"
      * "EXP"       * "LN"         * "SQRT"       * "MAX"
      * "MIN"       * REALPROCID * REALID       * INTID
      * BOOID       * LABELID
      = IDMAX       * "IF"         * "GO"         * "WHILE"
      * "DO"                       * "RETURN"     * "BEGIN"
      * "END"       * "ELSE"       * ";"          * "UNTIL"
      * "LABEL"     * "REAL"       * "INTEGER"    * "BOOLEAN"
      * "FALSE"     * "TRUE"       * FCONSTANT    * RCONSTANT
```

```
        * "FUNCTION" * "LIST"      * "PLOT"     * "DELETE"
        * "STOP"     = "QUIT"      * "RESET"
        * "TO"       * "THEN"      * "AND"      * "OR"
        * "NOT"      * "="         = "EQL"      * "NEQ"
        * "LSS"      * "LEQ"       * "GEQ"      * "GTR"
        * "("        * ")"         * ","        * ":"
        * "+"        * "-"         * "x"        * "/"
        * "*"        * "."         =  """       =  "&"
        =  "%"       =  "$"        = "#"        = "@"
        = "["        = "]"         = EOF        = NUMERR
        = CLASSMAX;
SYMBOL FORMAT    *          %  INTERPRETER INSTRUCTIONS
    1 = BOF         %    BRANCH ON FALSE
        * BUN       %    BRANCH UNCONDITIONAL
        * CHS       %    CHANGE SIGN
        * ADDP      %    ADD
        * SBOP      %    SUBTRACT
        * MULOP     %    MULTIPLY
        * DVDOP     %    DIVIDE
        * FACTOP    %    RAISE TO POWER
        * LNG       %    LOGICAL NEGATE
        * OROP      %    LOGICAL OR
        * ANDOP     %    LOGICAL AND
        * EQLF      %    =
        * NEQF      %    ≠
        * LSSF      %    <
        * LEQF      %    ≤
        * GEQF      %    ≥
        * GTRF      %    >
        * MKS       %    MARK THE STACK FOR RETURN
        * SBR       %    CALL FUNCTION
        * RTN       %    RETURN FROM FUNCTION
        * ZRN       %    MAKE SPACE FOR VARIABLES IN STACK
        * LITC      %    USE ADDRESS AS CONSTANT
        * OPDC      %    "OPERAND CALL"
        * STD       %    STORE DESTRUCTIVE
        * ISD       %    INTEGER STORE DESTRUCTIVE
        * SND       %    STORE NONDESTRUCTIVE
        * ISN       %    INTEGER STORE NONDESTRUCTIVE
        * BLN       %    BOOLEAN VALUE
        * MAXF      %    FIND MAX OF TWO NUMBERS
        * MINF      %    FIND MIN OF TWO NUMBERS
        * RAND      %    RANDOM NUMBER
        * ABSF      %    ABSOLUTE VALUE
        * SINF      %    SINE FUNCTION
        * COSF      %    COSINE FUNCTION
        * EXPF      %    EXP FUNCTION
        * LNF       %    NATURAL LOG FUNCTION
        * SQRTF     %    SQUARE ROOT FUNCTION
        * RCN ;     %    NON INTEGER NUMBER
```

```
%  THE GETNEXT AND ERROR PROCEDURES
PROCEDURE GETNEXT;
BEGIN
LABEL LS;
        IF INTERPRETING THEN
                BEGIN
                J := L DIV 8;
                CLASS := REAL(STRF(ATSM(J,BASE))(L.[45:3],1));
                L := L + 1
                END
        ELSE
                BEGIN
                IF CLASS = EOF OR NEWLINE THEN
                        BEGIN
                        NEWLINE := FALSE;
                        IF COMPILING THEN
                                BEGIN
LS:                             TWXS2 := FILL(LINENO,3) & ":";
                                TAB := 4; TERPRI;
                                LINENO := LINENO + 1
                                END;
                        READ TWX;
                        END;
                CASE READCON(FALSE) OF
                        BEGIN
                                IF COMPILING THEN GO TO LS
                                     ELSE CLASS := EOF;
                                CLASS := NUMERR; % ILLEGAL NUMBER
                                BEGIN  % NUMBER
                                CLASS := RCONSTANT;
                                IF INREAL < 4096 THEN
                                IF INREAL = J := INREAL THEN
                                        BEGIN
                                        INREAL := J;
                                        CLASS := FCONSTANT;
                                        END;
                                END;
                                IF CLASS := CORF(INSYM) ≤ CLASSMAX
                                        THEN INFO := NIL
                                ELSE
                                        BEGIN
                                        WHL := WH(INFO:=ATSM(CLASS));
                                        CLASS := WHL.CLSS;
                                        ADDRESS := WHL.ADRS;
                                        END;
                                CLASS := 0;
                        END OF CASE STATEMENT;
                END
END OF GETNEXT;
```

```
PROCEDURE ERROR(X); VALUE X; REAL X;
BEGIN
       IF COL < 68 THEN PRINT SPACE(COL + 3) #*#;
       PRINT #ERROR: MISSING #;
       CASE X - 1 OF
              BEGIN
              PRINT #OR ILLEGAL IDENTIFIER IN DECLARATION#;
              PRINT #; OR END#;
              PRINT #LEFT PARENTHESIS#;
              PRINT #RIGHT PARENTHESIS#;
              PRINT #OR ILLEGAL STATEMENT#;
              PRINT #"UNTIL" IN DO STATEMENT#;
              PRINT #OR ILLEGAL USE OF LABEL#;
              PRINT #"THEN" IN CONDITIONAL EXPRESSION OR
                     STATEMENT#;
              PRINT #"ELSE" IN CONDITIONAL EXPRESSION#;
              PRINT #COLON FOLLOWING LABEL#;
              PRINT #LABEL IN GO TO STATEMENT#;
              PRINT #"DO" IN WHILE STATEMENT#;
              PRINT #"=" FOLLOWING ":" IN ASSIGNMENT
                     STATEMENT#;
              PRINT #,#;
              PRINT #OR ILLEGAL BOOLEAN EXPRESSION#;
              END
END OF ERR;
SYMBOL FORMAT *CLASS,GETNEXT,ERROR;
%  MISC. ROUTINES FOR ACCESSING CODE STRING
%         PUT               PLACES CHARACTER IN CODE STRING
PROCEDURE PUT(T,A); VALUE T,A; REAL T,A;
       STR[A DIV 8](A MOD 8,1) := STRING(T.[42:6],1);
%         PUTADR            PUTS ADDRESS IN CODE STRING
PROCEDURE PUTADR(T,A); VALUE T,A; REAL T,A;
       BEGIN
       PUT(T.[36:6],A);
       PUT(T, A + 1)
       END;
%         GET               GETS CHARACTER FROM CODE STRING
REAL PROCEDURE GET(A); VALUE A; REAL A;
       GET := REAL(STR[A DIV 8](A.[45:3],1));
%         GETADR            GETS ADDRESS FROM CODE STRING
REAL PROCEDURE GETADR(A); VALUE A; REAL A;
       GETADR := GET(A) × 64 + GET(A + 1);
%         EMIT              EMITS ONE INSTRUCTION
PROCEDURE EMIT(X); VALUE X; REAL X;
       BEGIN
       PUT(X,L);
       IF L ≥ 798 THEN
              BEGIN
              PRINT #CODE OVERFLOW#;
              LINENO := 0
```

```
              END
          ELSE
              L := L + 1;
          END;
%          EMITADR        EMITS ADDRESS OF INSTRUCTION
PROCEDURE EMITADR(A); VALUE A; REAL A;
          BEGIN
          EMIT(A.[36:6]);
          EMIT(A)
          END;
%          EMITNUM        EMITS NUMBERS
PROCEDURE EMITNUM(C); VALUE C; REAL C;
BEGIN
LABEL FOUND;
          FOR J := 0 STEP 1 UNTIL CMAX DO
          IF CONSTANT[J] = C THEN GO TO FOUND;
          CONSTANT[J := CMAX := CMAX + 1] := C;
FOUND: EMIT(RCN);
          EMITADR(CONSTADR[J]);
          CONSTADR[J] := L - 2
END OF EMITNUM;
%  FORWARD DECLARATIONS
SYMBOL FORMAT AEXP; FORWARD;
SYMBOL FORMAT BEXP; FORWARD;
REAL PROCEDURE EXPRESS; FORWARD;
BOOLEAN PROCEDURE STATEMENT; FORWARD;
PROCEDURE EXECUTE(CODE); VALUE CODE; SYMBOL CODE; FORWARD;
%  EXPRESSION SCANNERS
%          IFEXP          HANDLES CONDITIONAL EXPRESSIONS
SYMBOL FORMAT IFEXP(X); SYMBOL FORMAT X;
BEGIN
REAL T, F;
          [*BEXP, "THEN":8; EMIT(BOF); F := BUMPL; *X, "ELSE":9;
              EMIT(BUN); PUTADR(T := BUMPL,F-2);
          *X, PUTADR(L,T - 2);]
END OF IFEXP;
%          VARIABLE       COMPILES VARIABLES & ASSIGNMENTS
SYMBOL FORMAT VARIABLE(TYPE,ADDRESS,FROM);
          VALUE TYPE,ADDRESS,FROM;
          REAL TYPE, ADDRESS, FROM;
          [[":", "=":13; [*TYPE = BOOID, *BEXP ELSE *AEXP],
              EMIT(REAL(TYPE = INTID) + FROM + STD) ELSE
          *FROM = 2, EMIT(OPDC);],
              EMITADR(ADDRESS);]];
%          PANA           PARENTHESIS AND ARITHMETIC EXPRES
SYMBOL FORMAT PANA ;  ["(":3; *AEXP, ")":4];
%          PRIMARY        COMPILES ARITHMETIC PRIMARIES
SYMBOL FORMAT PRIMARY;
BEGIN
REAL I, N;
```

```
        [REALID * INTID: I := CLASS; J := ADDRESS;
                *VARIABLE(I,J,2) ELSE
        REALPROCID: EMIT(MKS);  N := (T := WFL).NPARAM;
                [*N = 0 ELSE
                "(":3; *AEXP,
                        [*N := N - 1 ≠ 0, ",":14; *AEXP,
                                RETURN ELSE
                        ")":4]],
                EMIT(SBR); EMITADR(T.[18:9]); EMIT(T.[27:6])
                        ELSE
        FCONSTANT: EMIT(LITC); EMITADR(INREAL) ELSE
        RCONSTANT: EMITNUM(INREAL) ELSE
        "ABS" * "SQRT": T := CLASS - (="ABS"); *PARA,
                EMIT(ABSF + T) ELSE
        "MAX" * "MIN": T := CLASS - (="MAX");
                "(":3; *AEXP, ",":14; *AEXP, ")":4;
                EMIT(MAXF + T) ELSE
        "REAL", "(":3; *BEXP, ")":4 ELSE
        "RANDOM", EMIT(RAND) ELSE
        *PARA]
END OF PRIMARY;
%           FACTOR          COMPILES ARITHMETIC FACTORS
SYMBOL FORMAT FACTOR;
        [*PRIMARY, ["*", *PRIMARY, EMIT(FACTOP); RETURN
                ELSE NIL]];
%           TERM            COMPILES ARITHMETIC TERMS
SYMBOL FORMAT TERM;
BEGIN
REAL T;
        [*FACTOR,["×" * "/": T := CLASS - (="×"); *FACTOR,
                EMIT(MULOP + T); RETURN ELSE  NIL]]
END OF TERM;
%           AEXP            COMPILES ARITHMETIC EXPRESSIONS
SYMBOL FORMAT AEXP;
BEGIN
REAL T;
        ["IF", *IFEXP(AEXP) ELSE
        ["+" * "-": T := CLASS ELSE NIL],
                *TERM, IF T = (="-") THEN EMIT(CHS);
                ["+" * "-": T := CLASS - (="+");
                        *TERM,
                        EMIT(ADOP + T); RETURN ELSE
                NIL]]
END OF AEXP;
%           BOOPRIM         COMPILES BOOLEAN PRIMARIES
REAL PROCEDURE BOOPRIM;
BEGIN
REAL T;
BOOLEAN NOTFLAG;
LABEL LF, EXIT;
```

```
        ["NOT", NOTFLAG := TRUE ELSE NIL],
        [BOOID: T := ADDRESS; *VARIABLE(BOOID,T,2) ELSE
          "FALSE" * "TRUE": EMIT(BLN);
               EMIT(CLASS - (="FALSE")) ELSE
          "BOOLEAN", *PANAJ, T := BTYPE ELSE
        "(", T := EXPRESS; ")":4 ELSE
        *AEXP, T := ATYPE;],
                       [*T = ATYPE,
                             ["EQL" * "GTR": T := CLASS-(="EQL");
                                   *AEXP, EMIT(EQLF+T);
                                   T := BTYPE ELSE
                              NIL]  ELSE
                       NIL],
                       [*T = BTYPE,
                             [*NOTFLAG, EMIT(LNG) ELSE NIL] ELSE
                       *T = ATYPE AND NOT NOTFLAG]]: LF;
        RETURN T;
LF:     BOOPRIM := 0
END OF BOOPRIM;
%             BOOSEC          COMPILES BOOLEAN TERMS
REAL PROCEDURE BOOSEC;
BEGIN
REAL T;
LABEL LF;
        [*T := BOOPRIM = BTYPE,
              ["AND", *BOOPRIM = BTYPE, EMIT(ANDOP); RETURN
               ELSE NIL] ELSE
        *T = ATYPE] : LF;
        RETURN T;
LF:     BOOSEC := 0
END OF BOOSEC;
%             EXPRESS         COMPILES BOTH ARITHMETIC AND
%                             BOOLEAN EXPRESSIONS
REAL PROCEDURE EXPRESS;
BEGIN
REAL T, R, S;
LABEL LF, EXIT;
        ["IF",*BEXP, "THEN":8; EMIT(BOF); R := BUMPL;
              T := EXPRESS; "ELSE":9; EMIT(BUN);
              PUTADR(S := BUMPL,  R - 2);
              [*T = ATYPE, *AEXP ELSE *T = BTYPE, *BEXP],
              PUTADR(L,S-2) ELSE
        *T := BOOSEC = BTYPE, ["OR", *BOOSEC = BTYPE,
              EMIT(OROP); RETURN ELSE NIL] ELSE
        *T = ATYPE] : LF;
        RETURN T;
LF:     EXPRESS := 0;
END OF EXPRESS;
%             BEXP            COMPILES BOOLEAN EXPRESSIONS
SYMBOL FORMAT BEXP;
```

```
        ["IF", *IFEXP(BEXP) ELSE *EXPRESS = BTYPE ELSE
                ERROR(15); *FALSE]];
%   STATEMENT SCANNERS
%             COMPOUNDTAIL   TAIL END OF COMPOUND STATEMENT
SYMBOL FORMAT COMPOUNDTAIL;
        [*STATEMENT, [";", RETURN START ELSE "END":2]];
%             STMT           SCANS SOME STATEMENT BEGINNERS
SYMBOL FORMAT STMT;
BEGIN
REAL T,A;
        [SWITCH
        REALID * INTID * BOOID: T := CLASS; A := ADDRESS;
                *VARIABLE(T,A,0) ELSE
        LABELID: T := WHL; WHL.ADRS := L;
                WH(INFO) := ABS(WHL); *T < 0,
                ":":10; RETURN ELSE
        "IF", *BEXP, "THEN":8; EMIT(BOF); A := BUMPL;
                *STATEMENT,
                ["ELSE", EMIT(BUN); T := BUMPL;
                        PUTADR(L,A-2);
                        *STATEMENT, PUTADR(L,T-2) ELSE
                PUTADR(L,A-2);] ELSE
        "GO", ["TO" ELSE NIL], LABELID:11; EMIT(BUN);
                EMITADR(WHL.LINK); LINKF(INFO) := L - 2 ELSE
        "WHILE", A := L; *BEXP, "DO":12; EMIT(BOF); T := BUMPL;
                *STATEMENT, EMIT(BUN); EMITADR(A);
                PUTADR(L,T - 2) ELSE
        "DO", T := L; *STATEMENT, "UNTIL":6; *BEXP,
                EMIT(BOF); EMITADR(T) ELSE
        "RETURN", *AEXP, EMIT(RTN) ELSE
        "BEGIN", *COMPOUNDTAIL]
END OF STMT;
%             RESETLABELS    RESETS FORWARD LABEL REFERENCES
%                            WHEN "UNCOMPILING" (RECOVERING
%                            FROM ERRORS)
PROCEDURE RESETLABELS;
BEGIN
REAL T, A;
SYMBOL S;
        NEWLINE := TRUE;
        GETNEXT;
        FOR S IN PLIST DO
        IF (T := WH(CDR S)).CLSS = LABELID THEN
                BEGIN
                A := T.LINK;
                WHILE A ≠ 4095 AND A > L DO A := GETADR(A);
                T.LINK := A;
                IF T > 0 THEN
                IF T.ADRS > L THEN
                        BEGIN
```

```
                    T := - T;
                    T.ADRS := 0;
                    END;
              WH(CDR S) := T;
              END;
        FOR J := 0 STEP 1 UNTIL CMAX DO
              BEGIN
              A := CONSTADR[J];
              WHILE A ≠ 0 AND A ≥ L DO A := GETADR(A);
              CONSTADR[J] := A
              END;
END OF RESETLABELS;
%           STATEMENT       COMPILES ALL STATEMENTS & HANDLES
%                           RECOVERY FROM ERRORS IN STATEMENTS
BOOLEAN PROCEDURE STATEMENT;
BEGIN
LABEL RECOV, START;
REAL LNR, LO;
        LNR := LINENO - 1;
        LO := L;
START: [*STMT ELSE
          *CLASS ≥ (="END") AND CLASS ≤ (="UNTIL") ELSE
          "RESET":5; LINENO := INTEGER(READN(IWX));
              GO TO RECOV;] : RECOV;
        RETURN TRUE;
RECOV:IF LINENO ≥ LNR THEN
              BEGIN
              L := LO;
              PRINT #RETYPE STARTING AT LINE # LINENO := LNR;
              RESETLABELS;
              GO TO START
              END;
        STATEMENT := BOOLEAN(2);
END OF STATEMENT;
%  DECLARATION SCANNERS
%           ENTRY           RUN TIME SYNTAX CLASS ASSIGNMENT
BOOLEAN PROCEDURE ENTRY(TYPE); VALUE TYPE; REAL TYPE;
BEGIN
        IF CLASS > 0 THEN
              BEGIN
              ERROR(1);
              RETURN FALSE
              END;
        PLIST := (INSYM := MKATOM) . PLIST;
        CDR(INSYM) := INF := CONS[CLASSF: TYPE,
              ADDRESSF: IF TYPE = LABELID OR FORMTOG THEN 0
                                   ELSE
                    IF TYPE = REALPROCID THEN
                                 NEXTAVL(SYMBOL) + 2 ELSE
                                COUNT := COUNT + 1];
```

```
        IF TYPE = LABELID THEN
                BEGIN
                WH(INF) := -WH(INF);
                LINKF(INF) := 4095;
                END
        ELSE
        IF FORMTOG THEN COUNT := COUNT + 1;
        GETNEXT;
        ENTRY := TRUE;
END OF ENTRY;
%            ENTER            APPLIES ENTRY TO LIST OF
%                             IDENTIFIERS
SYMBOL FORMAT ENTER(X); VALUE X; REAL X;
        [*ENTRY(X), [",", RETURN START ELSE NIL]];
%            DECLARATION      HANDLES DECLARATIONS
SYMBOL FORMAT DECLARATION;
        LSWITCH
                "LABEL",       *ENTER(LABELID) ELSE
                "REAL",        *ENTER(REALID) ELSE
                "INTEGER",     *ENTER(INTID) ELSE
                "BOOLEAN",     *ENTER(BOOID) ];
%            PURGE            REMOVES ATOMIC SYMBOLS FROM
%                             OBJECT LIST WHEN LEAVING THE
%                             BLOCK AND CHECKS FOR MISSING
%                             LABELS
PROCEDURE PURGE(L); VALUE L; SYMBOL L;
BEGIN
SYMBOL R;
REAL T,A;
        COUNT := 0;
        NOLABEL := FALSE;
        FOR R IN L DO
                BEGIN
                IF (T := WH(CDR R)).CLSS = LABELID THEN
                        IF T < 0 THEN NOLABEL := TRUE
                        ELSE
                                BEGIN
                                A := T.LINK;
                                T := T.ADRS;
                                WHILE A ≠ 4095 DO
                                        BEGIN
                                        J := GETADR(A);
                                        PUTADR(T,A);
                                        A := J
                                        END;
                                END;
                IF SMTA(R) > 64 THEN
                        REMOB(R)
                ELSE CDR R := NIL;
                COUNT := COUNT + 1
```

```
            END;
END OF PURGE;
%           DECLARE         HANDLES SERIES OF DECLARATIONS
%                           AND PROVIDES FOR RECOVERY FROM
%                           SYNTAX ERRORS
BOOLEAN PROCEDURE DECLARE;
BEGIN
LABEL START,RECOV;
REAL I, LNR, CO;
SYMBOL PLISTO;
START: CO := COUNT;
       T := NEXTAVL(SYMBOL);
       PLIST := NIL;
       LNR := LINENO;
       [*DECLARATION, ";",
            PLISTO := NCONC(PLIST,PLISTO); GO TO START ELSE
        "RESET": LINENO := INTEGER(READN(TWX));
            GO TO RECOV ELSE
        NIL] : RECOV;
       PLIST := PLISTO;
       RETURN TRUE;
RECOV:PURGE(PLIST); NEXTAVL(SYMBOL) := T;
       IF LINENO ≥ LNR THEN
            BEGIN
            COUNT := CO;
            PRINT #RETYPE STARTING AT LINE # LINENO := LNR;
            GO TO START
            END;
       DECLARE := BOOLEAN(2);
       PLIST := PLISTO;
END OF DECLARE;
%           DUMPCODE        TRANSFERS CODE FROM CODE STRING
%                           TO THE LISP "STACK" AT A POINT
%                           WORD OF THE ATOMIC SYMBOL
%                           REPRESENTING THE FUNCTION
PROCEDURE DUMPCODE;
BEGIN
INTEGER LMAX;
REAL T;
       LMAX := (L + 7) DIV 8;
       FOR J := 0 STEP 1 UNTIL CMAX DO
            WHILE T := CONSTADR[J] ≠ 0 DO
                 BEGIN
                 CONSTADR[J] := GETADR(T);
                 PUTADR(J + LMAX, T);
                 END;
       FOR J := 0 STEP 1 UNTIL LMAX - 1 DO
            STRF(CONS) := STR[J];
       FOR J := 0 STEP 1 UNTIL  CMAX DO
            WH(CONS) := CONSTANT[J];
```

```
END OF DUMPCODE;
%           PROCEDUREDEC  HANDLES DECLARATIONS OF FUNCTIONS
%                         TO BE PLOTTED
PROCEDURE PROCEDUREDEC;
BEGIN
REAL R, T, P, FA, LNR;
LABEL START, NOGO, RECOV, EXIT;
SYMBOL PLISTO, FN, S, PINFO, PARAM;
        LINENO := 1;
        CMAX := -1;
        PLISTO := PLIST;
        R := NEXTAVL(SYMBOL);
        PLIST := NIL;
        [*ENTRY(REALPROCID), FN := PLIST; PLIST := NIL;
                        CTR FN := ATSM(R);
                        PINFO := INF; P := NEXTAVL(SYMBOL);
                        COUNT := 0; FORMTOG := TRUE;
                [")", *ENTER(REALID), ")":4; FA := 2048;
                        FOR S IN PLIST DO
                                ADDRESSF(CDR S) := FA := FA + 1 ELSE
                 NIL]]:NOGO;
        NPAR(PINFO) := COUNT;
        PARAM := PLIST;
        PLIST := NIL;
        T := NEXTAVL(SYMBOL);
        FORMTOG := FALSE;
        INPUT(TWXF1,TWXS1,136,/FALSE);
        COMPILING := TRUE;
        GETNEXT;
        LNR := LINENO - 1;
START:COUNT := L := 0;
        ["BEGIN", *DECLARE,
                IF COUNT ≠ 0 THEN
                        BEGIN
                        EMIT(ZRN); EMIT(COUNT)
                        END;
                *COMPOUNDTAIL ELSE
         *STATEMENT] : RECOV;
        EMIT(BLN); EMIT(0); EMIT(RTN);
        PURGE(PLIST);
        IF NOLABEL THEN
                BEGIN
                PRINT #DECLARED LABELS DID NOT OCCUR#;
                GO TO NOGO
                END;
        PURGE(PARAM);
        NEXTAVL(SYMBOL) := P;
        DUMPCODE;
        PLISTO := NCONC(FN, PLISTO);
        GO EXIT;
```

```
RECOV:IF LINENO ≥ LNR THEN
                BEGIN
                PURGE(PLIST);
                NEXTAVL(SYMBOL) := T;
                PRINT #RETYPE STARTING AT LINE # LINENO := LNR;
                NEWLINE := TRUE;
                GETNEXT;
                GO TO START
                END;
NUGO: PURGE(PARAM);
        PURGE(PLIST);
        FORMTOG := FALSE;
        PURGE(FN);
        NEXTAVL(SYMBOL) := R;
EXIT: COMPILING := FALSE; PLIST := PLISTO;
        INPUT(TWXF1,TWXS1,146,/TRUE);
END OF PROCEDUREDEC;
%  INTERPRETER SECTION
%          MKADR          REMOVES 2 CHARACTERS FROM CODE
%                         STRING TO BE USED AS AN ADDRESS
INTEGER PROCEDURE MKADR(A); VALUE A; BOOLEAN A;
BEGIN
REAL T;
        T := CLASS;
        GETNEXT;
        T := T × 64 + CLASS;
        GETNEXT;
        MKADR := IF A THEN
                        IF T > 2048 THEN FR + 2048 - T
                                ELSE FR + T
                        ELSE T;
END OF MKADR;
%          INTERPRET      INTERPRETS THE CODE STRING
SYMBOL FORMAT INTERPRET;
BEGIN
DEFINE
        SA = STACK[I := I - 1]#,
        SB = STACK[I]#,
        SC = STACK[I + 1]#,
        SD = STACK[I := I + 1]#;
ALPHA W;
REAL T;
        LSWITCH
                BOF, I := I - 1;
                        IF BOOLEAN(SC) THEN
                                BEGIN
                                L := L + 1;
                                GETNEXT
                                END
                        ELSE
```

```
                    BEGIN
                    L := MKADR(FALSE);
                    GETNEXT
                    END ELSE
            BUN, L := MKADR(FALSE); GETNEXT ELSE
            CHS, SB := - SB ELSE
            ADDP, SA := SB + SC ELSE
            SBOP, SA := SB - SC ELSE
            MULUP, SA := SB × SC ELSE
            DVDUP, SA := SB / SC ELSE
            FACTOP, SA := SB * SC ELSE
            LNG, SB := REAL(NOT BOOLEAN(SB)) ELSE
            ORCP, SA := REAL(BOOLEAN(SB) OR BOOLEAN(SC))ELSE
            ANDOP, SA := REAL(BOOLEAN(SB) AND BOOLEAN(SC))
                  ELSE
            EQLF, SA := REAL(SB = SC) ELSE
            NEQF, SA := REAL(SB ≠ SC) ELSE
            LSSF, SA := REAL(SB < SC) ELSE
            LEQF, SA := REAL(SB ≤ SC) ELSE
            GEQF, SA := REAL(SB ≥ SC) ELSE
            GTRF, SA := REAL(SB > SC) ELSE
            MKS, SD := SAVEI; SAVEI := I ELSE
            SBR, SD := FR; FR := I;
                  EXECUTE(ATSM(MKADR(FALSE)×64 + CLASS));
                  GETNEXT ELSE
            RTN, T := SB; FR := STACK[FR];
                  SAVEI := STACK[I := SAVEI]; SB := T;
                  *FALSE ELSE
            ZRN, FOR CLASS DO SD := 0; GETNEXT ELSE
            LITC, SD := MKADR(FALSE) ELSE
            OPDC, SD := STACK[MKADR(TRUE)] ELSE
            STD * ISD * SND * ISN:
                  IF BOOLEAN(T := CLASS - STD) THEN
                        SB := INTEGER(SB);
                  NIL, STACK[MKADR(TRUE)] := SB;
                  IF T < 2 THEN I := I - 1;  ELSE
            BLN, SD := CLASS; GETNEXT ELSE
            MAXF, SA := MAX(SB,SC) ELSE
            MINF, SA := MIN(SB,SC) ELSE
            RAND, SD := CONVAL(0) ELSE
            ABSF, SB := ABS(SB) ELSE
            SINF, SB := SIN(SB) ELSE
            COSF, SB := COS(SB) ELSE
            EXPF, SB := EXP(SB) ELSE
            LNF, SB := LN(SB) ELSE
            SQRTF, SB := SQRT(SB) ELSE
            RCN, SD := WH(ATSM(MKADR(FALSE),BASE));]
END OF INTERPRET;
%           EXECUTE        MAKES CALLS ON FUNCTIONS; USES
%                          INTERPRET
```

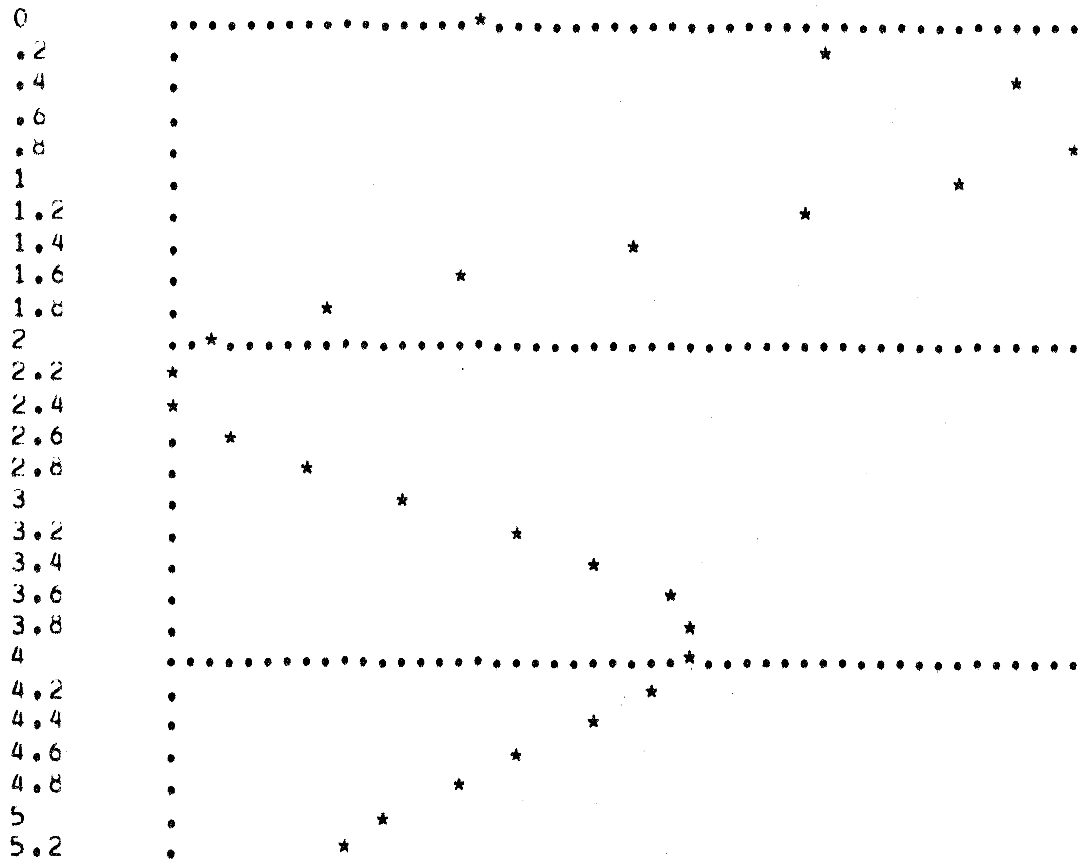
```
PROCEDURE EXECUTE(CODE); VALUE CODE; SYMBOL CODE;
BEGIN
SYMBOL OLDCODE;
REAL OLDL;
        OLDCODE := BASE;
        BASE := CODE;
        OLDL := L;
        L := 0;
        GETNEXT;
        WHILE INTERPRET DO;
        BASE := OLDCODE;
        L := OLDL;
END OF EXECUTE;
%           PLOTTER          PLOTS FUNCTION EXECUTED BY
%                            INTERPRETER
PROCEDURE PLOTTER;
BEGIN
REAL X, Y, Z, N, V, TMAX, TMIN, T, J, K;
SYMBOL CODE, SYM;
LABEL EXIT;
        IF CLASS ≠ REALPROCID OR K := WHL.NPARAM = 0 THEN
                BEGIN PRINT #MISSING OR ILLEGAL FUNCTION#;
                        GO TO EXIT END;
        SYM := INSYM;
        CODE := ATSM(1,INFO);
        IF K ≠ 1 THEN
                BEGIN
                PRINT #TYPE # K-1 # PARAMETER#
                        IF K = 2 THEN # # ELSE #S#;
                J := 1;
                DO STACK[J := J + 1] := READN(TWX)
                         UNTIL J = K;
                END;
        K := K + 1;
        PRINT #ENTER BEGINNING VALUE, INCREMENT, AND FINAL VALUE FOR#,
                #THE PLOT#;
        X := READN(TWX); Y := READN(TWX); Z := READN(TWX);
        N := (Z - X) / Y + 1;
        J := -1; INTERPRETING := TRUE;
        BEGIN
        ARRAY VALUES[0:N];
                TMAX := -(TMIN := TEN[68]);
                FOR V := X STEP Y UNTIL Z DO
                        BEGIN
                        SAVEI := 0;
                        FR := I := K;
                        STACK[1] := V;
                        EXECUTE(CODE);
                        VALUES[J := J + 1] := T := STACK[0];
                        TMAX := MAX(TMAX,T);
```

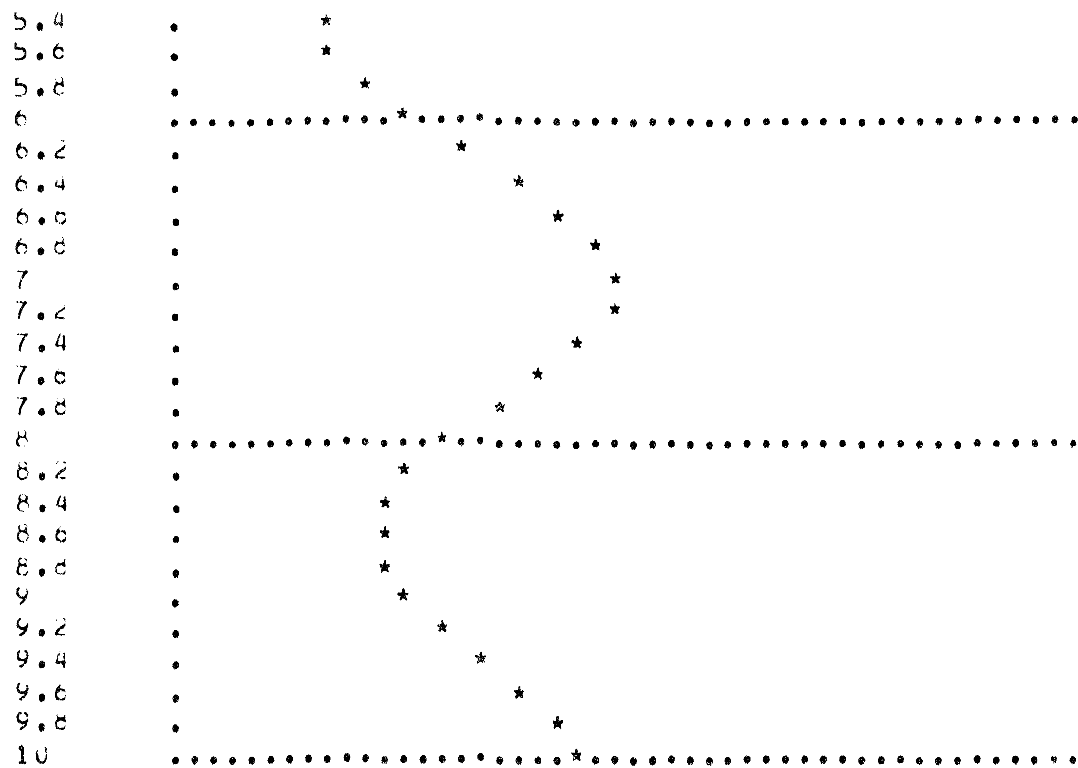
```
                    TMIN := MIN(TMIN,T);
                    END;
                PRINT ,/,/,/,/,/ SPACE(8) #PLOT OF # SYM
                     #(X),   X = # X # TO # Z # IN STEPS OF #
                    Y,/SPACE(16) #RANGE OF # SYM # IS #
                    TMIN # TO # TMAX ,/,/,/;
                IF TMAX ≠ TMIN THEN
                TMAX := 50/(TMAX - TMIN);
                J := -1;
                FOR V := X STEP Y UNTIL Z DO
                    BEGIN
                    TWXS2 := SPACE;
                    PRIN V;
                    T := (VALUES[J:=J+1] - TMIN) × TMAX + 8;
                    IF J MOD 10 = 0 THEN
                        TWXS2(8) := [2:"....."] & TWXS2(8,42)
                    ELSE TWXS2(8) := ".";
                    TWXS2(T,1) := "*";
                    WRITE TWX;
                    END;
                END;
        PRIN ,/,/,/,/,/;
EXIT: NEWLINE := TRUE;
        INTERPRETING := FALSE;
END OF PLOTTER;
%  MAIN PROGRAM SECTION
        PRINT #RECC FUNCTION PLOTTER#;
START:
        PRINT #GO AHEAD.#,/;
              NEWLINE := TRUE; GETNEXT;
[SWITCH
        "FUNCTION", PROCEDUREDEC ELSE
        "LIST", PRINT PLIST ELSE
        "PLOT", PLOTTER ELSE
        "DELETE", IF NULL PLIST THEN
                            PRINT #FUNCTION LIST EMPTY#
                  ELSE
                            BEGIN
                            DL := SMTA(CTR(INF := PLIST));
                            PLIST := CDR PLIST; CDR INF := NIL;
                            PRINT CAR INF, #DELETED#;
                            PURGE(INF);
                            NEXTAVL(SYMBOL) := DL;
                            END ELSE
        "STOP", GO TO EXIT;] : RESTART;
        GO TO START;
RESTART:  PRINT #PLEASE RETYPE#; GO TO START;
EXIT: PRINT #END OF PROGRAM.# ,/,/,/;
END.
```

```
CALL REMPLOT

RECC FUNCTION PLOTTER
GO AHEAD.
FUNCTION F(X);
  1:RETURN SIN(X)×COS(X)/(X+1);
GO AHEAD.
PLOT F
ENTER BEGINNING VALUE, INCREMENT, AND FINAL VALUE FOR THE PLOT
?0,.2 10
```

```
          PLOT OF F(X), X = 0 TO 10 IN STEPS OF .2
                   RANGE OF F IS -.14869 TO .29126
```





GO AHEAD.
STOP
END OF PROGRAM.

IX. GTL INPUT-OUTPUT FUNCTIONS

9.1 INTRODUCTION

In addition to the standard ALGOL Input-Output functions, GTL contains a set of Input-Output functions which facilitates reading and writing the GTL data types. The purpose of this section is to describe in detail the operation of these Input-Output functions and to indicate how they might be used with various kinds of files.

9.2 THE OUTPUT FUNCTIONS

9.2.1 Extended WRITE Statement

The syntax of the array row form of the ordinary Extended ALGOL WRITE statement has been extended as follows: Any string variable (See Subsection 5.1) which is not a formal parameter and which is longer than 8 characters in length may be used in place of an array row. The number of words to be written is specified as in the array row form, instead of the number of characters, since only multiples of 8 characters can be written. For example, if LINE is a 120-character string variable and OUTFILE is a 15-word output file, then

```
WRITE(OUTFILE, 15, LINE)
```

is a legal GTL construct.

9.2.2 The PRINT, PRIN, and TERPRI Statements

The PRINT (PRIN) statement consists of the word PRINT (PRIN) followed by a list of one or more printable items which are to be written on an output file. The output file is specified by the OUTPUT statement (Subsection 9.3, below). The OUTPUT statement also specifies an output string variable in which the printed output is composed,

the size of the output file, and the left and right margins. The PRINT statement will cause each item in the list of printable items to be inserted into the output string beginning at the left margin. If the output string is filled, i.e., the right margin is reached, before all of the items have been printed, then the output string is written onto the output file and output string composition process continues at the left margin. When all of the print items have been inserted into the output string the output string is written onto the output file.

The PRIN statement has the same effect as the PRINT statement except that the output string is not immediately written unless the right margin has been reached. Subsequent PRINT or PRIN statements will continue to fill the partially composed output string instead of restarting at the left margin. The TERPRI statement will cause a partially composed output string to be written after a series of one or more PRIN statements. (The PRINT statement is equivalent to a PRIN statement followed by the TERPRI statement.)

When two or more printable items appear in a PRINT or PRIN statement, they may be separated by one of three following print list separators:

1) One or more spaces, which causes two print items to be printed without intervening spaces,

2) A comma, which causes two print items to be printed with one intervening space, and,

3) A comma followed by a slash (",/"), which causes an implicit call on the TERPRI function starting a new line of print before the next item is printed).

For example,

PRINT I1 I2, I3,/ I4

will cause the following to be printed (assuming I1, I2, I3, and I4 are chosen to represent the symbolic output corresponding to the four printable items),

I1I2 I3

I4

The kinds of items which may be printed are described in detail in paragraphs 9.2.4 through 9.2.15 below.

9.2.3 The FORMAT Option

In addition to the PRINT and PRIN statements described above, there are four optional forms of output functions. Any one of the following may precede the list of printable items:

PRINT FORMAT

PRIN FORMAT

PRINT FORMAT [ae]

PRIN FORMAT [ae]

where ae represents an arithmetic expression. The use of this FORMAT option will cause the items to be printed to be spaced evenly across the line. After a printable item is inserted into the output string variable, spaces are inserted into the output string variable up to the smallest multiple of the spacing factor, the value of ae. If ae is not given, the spacing factor will retain its previous setting (the spacing factor is initially set to 15). In terms of the standard variable TAB

(described in paragraph 9.7.2), the equivalent of

```
PRIN SPACE(ae - (TAB MOD ae))
```

is executed after each printable item is composed and inserted into the output string variable (paragraph 9.2.14). For example,

```
PRINT FORMAT [5] I1 I2 I3 I4
```

will cause the following to be printed (assuming I1, I2, I3, and I4 are chosen to represent the symbolic output corresponding to the four printable items),

```
I1   I2   I3   I4
```

9.2.4 Literal String

A string to be printed, like a quoted string in a format statement, may be enclosed in #'s. The character # itself may be printed by ###. Two or more spaces in the string are reduced to one in the printed output. For example,

```
#THIS IS A LITERAL STRING#
###
# X = #
```

The length of a literal string may not exceed 896 characters. If a literal string will not fit into one line of output it will be divided into two or more strings (the print mechanism will attempt to avoid dividing a string in the middle of an identifier).

9.2.5 String Values

String valued printable items are string designators and string assignment statements (Section V). If the string thus generated will not fit into the output string, it will be divided in the manner described in paragraph 9.2.4 above.

9.2.6 Real and Integer Values

Real and integer valued printable items are real and integer variables, assignment statements, procedures and standard functions. The maximum number of significant figures to be printed is initially set to 5; it may be changed by the NTS function described in paragraph 9.2.16 below.

9.2.7 Alpha Values

Alpha variables, alpha procedures, and string constants (quoted strings containing 7 or less characters) are printable items. They are printed in standard alpha format (up to 7 characters in length).

9.2.8 Boolean Values

Any Boolean expression which does not begin with a conditional expression is a printable item. The Boolean values TRUE or FALSE are printed according to the value of the Boolean expression.

9.2.9 Double Precision Values

Double precision variables and assignment statements (Section III) are printable items. The maximum number of significant figures is initially set to 22; it may be changed by the NTS function described in paragraph 9.2.16 below.

9.2.10 Complex and Double Precision Complex Values

Complex and double precision complex variables and assignment statements (Section IV) are printable items. If the imaginary part of the complex number is zero, then only the real part is printed. If the real part of the complex number is zero and the imaginary part is non-zero, then only the imaginary part is printed. If the imaginary part is printed, then it is preceded by a colon (:). If the complex number is double precision, the remarks given in paragraph 9.2.9 above also apply.

9.2.11 LISP Values

LISP variables, procedures, and assignment statements, i.e., those declared with the type SYMBOL, and the LISP field designators (Section VI) are printable items. The item to be printed must have an S-expression representation; the circular list described in Subsection 6.7, for example, could not be printed.

9.2.12 Reference Values

Variables and procedures of type "reference" (Section VII) are printable items. The contents of the records referenced by the reference values are not printed. Instead, the record class identifier associated with the reference value and the reference value itself are printed.

9.2.13 QMARK

QMARK is a printable item which causes the question mark (the "illegal character") to be printed. It is provided since there would be no other convenient way of inserting a ? into the output string.

9.2.14 SPACE

The SPACE function is a printable item which may be used in one of the two forms:

SPACE

SPACE(*ae*)

where *ae* represents an arithmetic expression. If SPACE alone is used, then one space is printed. If the other form is used, then the value of the arithmetic expression *ae* determines the number of spaces to be printed. If the value of *ae* is negative or zero, then nothing happens. If the number of spaces to be printed extends beyond the right margin of the output string, then the string of spaces is truncated at the right margin, and does not extend onto the next line of print.

9.2.15 SKIP

The word SKIP used in the form

SKIP(*ae*)

where *ae* is an arithmetic expression, is a printable item. It causes spaces to be placed in the output string up to the point indicated by the value of the arithmetic expression. For example, if the output string is 120 characters long,

PRIN SKIP(60)

causes spaces to be filled in up to the sixtieth character position in the output string. If the output string has been filled to a point beyond the position given by the value of the arithmetic expression, or if the value of the arithmetic expression is zero or negative, then nothing happens.

9.2.16 The NTS Statement

The NTS statement may be used in one of the three following forms:

NTS(*, ae)

NTS(**, ae)

NTS(aev, ae)

where ae and aev are arithmetic expressions. The first form of the NTS statement will reset the value of the maximum number of significant figures of a single precision number to be printed to the value of ae (see paragraph 9.2.6). The second form of the NTS statement will reset the value of the maximum number of significant figures of a double precision number to be printed to the value of ae (see paragraph 9.2.9). The third form of the NTS statement will convert the value of the arithmetic expression aev into a string representing that value with a maximum number of significant figures determined by the value of ae. The string thus generated will be contained in the standard string variable OUTSTR (see paragraph 9.6.1) and its length is given by LENGTH (OUTSTR). For example,

NTS(123,5)

will cause the string "123" to be placed in the string designator

OUTSTR(0, LENGTH(OUTSTR))

where, in this case, LENGTH(OUTSTR) is equal to 3.

The string FILL statement (see paragraph 5.3.16) is similar to the third form of the NTS statement and may at times be more convenient.

9.2.17 Conditional PRINT Statement

In addition to the preceding forms of the PRIN and PRINT statements, the following "conditional" forms are allowed:

PRINT IF bexp THEN printlist1 ELSE printlist2

PRIN IF bexp THEN printlist1 ELSE printlist2

where bexp represents a Boolean expression, and printlist1 and printlist2 are lists of printable items defined previously. These constructs are equivalent to

IF bexp THEN PRINT printlist1 ELSE PRINT printlist2

and

IF bexp THEN PRIN printlist1 ELSE PRIN printlist2

respectively. For example,

```
PRINT IF X = 0 THEN #YES# ELSE #NO#
```

prints

```
YES
```

if X = 0 and otherwise prints

```
NO
```

9.3 THE OUTPUT STATEMENT

9.3.1 The Standard Form

The standard form of the OUTPUT statement, which may be used with any output file, is

OUTPUT(outfile, outputstring, filelength)

where outfile represents an output file identifier, outputstring represents a simple string variable, in which the output to be printed is composed, and filelength represents an arithmetic expression the value of which should be the length of the output file in characters. The output file and simple string variable should be declared in the outermost block of the program. The output file is declared by an ordinary ALGOL file declaration. The simple string variable (Section V) is declared in the form

STRING outputstring (n)

where n represents an unsigned integer which determines the length of the string variable in characters. The string variable length should be at least as long as the size of a logical record of the output file. With this form of the OUTPUT statement, the left margin of the output string is set to zero and the right margin is set to the value of the arithmetic expression filelength. For example, given declarations

```
FILE OUT TAPE (2, 56, 10)
     STRING TAP(80)
```

then the statement

```
OUTPUT(TAPE,TAP,80)
```

will cause the print mechanism to print successive logical records onto the TAPE file. An OUTPUT statement need only be executed once during a program, although it may be executed as many times as desired to change files, output string and/or left and right margins. The remote terminal output file (file type REMOTE), which is described in Reference 7, is treated the same way as any other output file in ALGOL. The most important difference is the restriction of the character set which may be printed; the remote terminal character set is given in Appendix B. Another difference is that a carriage return, line feed is generated before every line of printing. The other difference is the action taken when "break" or "output impossible" condition occurs. If either of these conditions occurs, the program is terminated with an error message. See Subsection 9.6 to avoid this action.

9.3.2 The Output Procedure

The name of an untyped procedure may be used in place of the output file identifier in the OUTPUT statement. The statement has the form

OUTPUT(outpro, outputstring, filelength)

where outpro represents the procedure identifier, and outputstring and filelength have the same meanings as defined in paragraph 9.3.1 above. The output procedure will be called whenever the output string has been filled or whenever TERPRI is called: it is assumed that the output procedure will write the output string on some output file. The procedure must be declared in the outermost block of the program, and must have no formal parameters.

For example, given the declarations

```
FILE OUT OUTFILE 16(2,15)
STRING LINE (120)
PROCEDURE OUTPRO;
WRITE(OUTFILE,15,LINE)
```

the OUTPUT statement

```
OUTPUT(OUTPRO, LINE, 120)
```

will have the same effect as the OUTPUT statement described in paragraph 9.7.2. This option is provided since it is sometimes desired to introduce certain kinds of side effects.

9.3.3 Setting Left and Right Margins

There are three forms of OUTPUT statements which allow settings of left and right margins in the output string variable to be filled by the print mechanism:

OUTPUT(_outfile_, _outputstring_, _filelength_, _lmargin_, _rmargin_)

OUTPUT(_outpro_, _outputstring_, _filelength_, _lmargin_, _rmargin_)

OUTPUT(*, _lmargin_, _rmargin_)

where _outfile_, _outputstring_, _filelength_, and _outpro_ have the same meanings as in paragraphs 9.3.1 and 9.3.2 above. _Lmargin_ and _rmargin_ represent arithmetic expressions whose values determine the left and right margins in the output string. The first OUTPUT statement given above is an extension of the OUTPUT statement described in paragraph 9.3.1. The second OUTPUT statement is an extension of the OUTPUT statement described in paragraph 9.3.2.

The third OUTPUT statement (the asterisk form) may be used after the execution of any OUTPUT statement to change left and right margins when there is no change in the output string, and the output file or output procedure. The value of lmargin determines the number of characters to be skipped in the output string before any printable item is inserted into the output string. The right margin determines the maximum number of characters, from the beginning of the output string, which can be placed in the output string before it is written (starting a new line of print). For example, with the declarations

```
FILE OUT OUTFILE 16(2,15)
   STRING LINE(120)
```

the output statement

```
OUTPUT(OUTFILE, LINE, 120, 8, 104)
```

will cause a line to be printed indented 8 spaces and will allow a maximum of 96 characters on a line (i.e., there is a cutoff of 104 characters from the beginning of the string LINE).

Warning: Since the output mechanism does not change the contents of the output string to the left of the left margin and to the right of the right margin, and since any string variable is initially filled with zeroes when declared, the output string variable should be filled with spaces before any print statement is used. This may be accomplished by the assignment statement

```
outputstring:= SPACE
```

For example, referring to the example given above,

```
LINE:= SPACE
```

will prevent zeroes from being printed before the left margin and after the right margin.

9.4 THE READ FUNCTIONS

9.4.1 Extended READ Statement

The syntax of the array row form of the ordinary Extended ALGOL READ statement has been extended as follows: Any string variable (Section 5) which is not a formal parameter and which is longer than eight characters in length may be used in place of an array row. The number of words to be read is specified, as in the array row form, instead of the number of characters since only multiples of eight characters can be read. For example, if CARD is an 80 character string variable and INFILE is a ten-word input file, then

```
READ(INFILE,10,CARD)
```

is a legal GTL construct.

9.4.2 The GTL Read Mechanism

When a GTL read function is called, one or more items of various kinds are read from an input file. The input file is specified either directly or indirectly (see Subsection 9.5). The read mechanism will fill an input string variable (also specified by the INPUT statement) from a logical record from the input file. The scanning process starts from a left margin in the input string variable and continues until the right

margin is reached, at which point the input string variable is refilled from the next logical record. The left and right margins of the input string are also specified by the INPUT statement. The scanning mechanism will scan the input string and extract one or more items depending on the type of read function being used. Blank spaces serve as delimiters only and do not contribute to the value of a read function (except in the case of the SCAN function described below). There are five basic different kinds of read functions provided which are described in paragraphs 9.4.3 through 9.4.7.

9.4.3 The SCAN Function

The SCAN function is an integer valued function having one of five possible values depending on the contents of the input string variable which it is scanning. The values of the SCAN function and their meanings are given in the table below.

Value of SCAN	Meaning
0	one or more spaces scanned
1	an identifier was scanned
2	a digit string was scanned
3	one non-alpha character was scanned
4	end of file has been reached

The SCAN function will scan up to 31 spaces at a time so that a value of zero does not mean that there are no remaining spaces; it simply means that one or more spaces were seen. When an identifier, digit string, or non-alpha character is scanned, the item scanned may be accessed through the standard string variable INSTR (see paragraph 9.7.1). The length of

the item scanned by a call on the SCAN function is given by LENGTH(INSTR), so that the string scanned by SCAN is given by

INSTR(0,LENGTH(INSTR))

9.4.4 The READCON Function

The READCON function is an integer valued function having one of five possible values depending on the contents of the input string variable which it is scanning. The READCON function is called in the form

READCON(bx)

where bx represents a Boolean expression. The value of bx determines whether a multi-character identifier will be read as a string or LISP atomic symbol. If the value of bx is TRUE; then every multi-character identifier is read as a LISP atomic symbol; otherwise, a multi-character identifier is reported to be a LISP atomic symbol only if an atomic symbol representing the identifier already exists. The values of the READCON function and their meanings are given in the table below.

Value of READCON	Meaning
0	end of file
1	number with exponent overflow (the exponent is too large or too small)
2	number
3	LISP atomic symbol
4	multi-character identifier string

As indicated above, READCON can have a value of four only if the value of its argument is FALSE. The value of an item read by READCON can be accessed

through the standard variables (see paragraph 9.7.1) INREAD, INDBL, INSYM, and INSTR. If a number is read, its value is given by INREAL in a single precision arithmetic expression, or by INDBL in a double precision expression. If a LISP atomic symbol is read, then its value is given by INSYM. If a multi-character identifier is read, then its value is contained in the string designator

INSTR(0,LENGTH(INSTR))

9.4.5 The READN Function

The READN function may be used to read numbers only. Its value is the number which is read. If used in a single precision context, its value is a single precision number; if used in a double precision context (Section III), its value is a double precision number. If the item read is not a number, its value is set to zero, and the standard variable INSYM is set to the question mark character (otherwise INSYM is set to zero).

9.4.6 The READ1 Function

The value of the READ1 function, when used in a symbol expression, is a LISP atomic symbol. This function is described further in paragraph 6.10.2.

9.4.7 The READ Function

The value of the READ Function, when used in a symbol expression, is a LISP S-expression. This function is described further in paragraph 6.10.2.

9.5 THE INPUT STATEMENT

9.5.1 The Standard Form

The standard form of the INPUT statement, which may be used with any input file

INPUT(infile, inputstring, filelength)

where infile represents an input file identifier, inputstring represents a simple string variable which the read mechanism will scan, and filelength represents an arithmetic expression the value of which should be the length of the input file (in characters). The input file and simple string variable should be declared in the outermost block of the program. The input file is declared by an ordinary ALGOL file declaration. The simple string variable (Section V) is declared in the form

STRING inputstring(n)

where n represents an unsigned integer which determines the length of the string variable in characters. The string variable length should be at least as long as the size of the logical record of the input file. With this form of the INPUT statement, the left margin of the input string is set to zero (the scanning starts at the beginning of the string variable) and the right margin is set to the value of the arithmetic expression filelength, so that the read mechanism will scan the entire logical record from the input file. For example, given the declarations (for a tape file)

```
FILE IN TAPE (2, 56, 10)
  STRING TAP(80)
```

then the statement

```
INPUT(TAPE,TAP,80)
```

will cause the read mechanism to scan from successive logical records from the TAPE file. An INPUT statement need be executed only once during a program, although it may be executed as many times as desired to change files, input string, and/or left and right margins.

NOTE: The remote terminal input file (file type REMOTE), which is described in reference 7, is treated the same way as any other input file in ALGOL. The most important difference is the restriction of the character set which may be printed; the remote terminal character set is given in Appendix B. Another difference is that a message is sent after every READ statement to give a carriage return, line feed. The other difference is the action taken when "parity", "buffer overflow", and "input too long" occur. If any of these conditions occur, the program is terminated with an error message. (See Subsection 9.6 to avoid this action.)

9.5.2 The Input Procedure

The name of a BOOLEAN procedure may be used in place of the input file identifier in the INPUT statement. This statement has the form

INPUT(inpro, inputstring, filelength)

where inpro represents the BOOLEAN procedure identifier, and inputstring and filelength have the same meanings as in paragraph 9.5.1 above. The input procedure will be called whenever the scanning mechanism has reached the right margin of the input string; it is assumed that the input procedure will refill the input string variable from some input file and will return a value of FALSE unless end of file is detected. The BOOLEAN procedure must be declared in the outermost block of the program, and must have no formal parameters. For example,

```
FILE IN INFILE(2, 10)
STRING CARD(80)
BOOLEAN PROCEDURE INPRO;
BEGIN LABEL EOF, EXIT;
     READ(INFILE, 10, CARD) [EOF]
     GO TO EXIT;
     EOF:  INPRO := TRUE;
     EXIT:
END OF INPRO
```

Then the INPUT statement

```
INPUT(INPRO, CARD, 80)
```

will have the same effect as the first input statement described in paragraph 9.8.1. This option is provided since it is sometimes desired to introduce certain kinds of side effects (see paragraph 9.8.1).

9.5.3 Setting Left and Right Margins

There are three forms of INPUT statements which allow settings of left and right margins in the input string variable to be scanned by the read mechanism:

INPUT(_infile_, _inputstring_, _filelength_, _lmargin_, _rmargin_)
INPUT(_inpro_, _inputstring_, _filelength_, _lmargin_, _rmargin_)
INPUT(*, _lmargin_, _rmargin_)

where _infile_, _inputstring_, _filelength_, and _inpro_ have the same meanings as in paragraphs 9.5.1 and 9.5.2 above. _Lmargin_ and _rmargin_ represent

arithmetic expressions whose values determine the left and right margins in the input string. The first INPUT statement given above is an extension of the INPUT statement described in paragraph 9.5.1. The second INPUT statement is an extension of the INPUT statement described in paragraph 9.5.2. The third INPUT statement (the asterisk form) may be used after the execution of any INPUT statement to change left and right margins when there is no change in the input string variable, and input file or input procedure. The value of lmargin determines the number of characters to be skipped in the input string before beginning the scan of a logical record. The value of rmargin determines the number of characters to be scanned (from the beginning of the logical record) before continuing on the next logical record. For example, with the declarations

```
FILE IN INFILE (2,10)
STRING CARD(80)
```

the INPUT statement

```
INPUT(INFILE, CARD, 80, 8, 72)
```

will cause the read mechanism to scan the first 72 characters from input string CARD (filled from a logical record from INFILE) after skipping over the first eight characters.

9.5.4 Sign-Number Separation

When using the read functions described in Subsection 9.4 (except for the SCAN function), the sign of a number and the number itself are originally read as one item (assuming there are no intervening spaces between the sign and the number).

For example,

+125   -42

appear in the input string variable being scanned; then two numbers, the first positive and the second negative, will be read. It is sometimes useful, however, to be able to read a number and its sign separately. This may be accomplished by inserting a comma followed by a slash followed by a Boolean expression immediately before the right parenthesis in any of the INPUT statements described above. If the value of the Boolean expression is FALSE, then a number and its sign (if any) will be read separately, and vice versa. To be explicit, the following additional forms of INPUT statements are allowed:

INPUT(infile, inputstring, filelength, /bx)

INPUT(inpro, inputstring, filelength, /bx)

INPUT(infile, inputstring, filelength, lmargin, rmargin, /bx)

INPUT(inpro, inputstring, filelength, lmargin, rmargin, /bx)

INPUT(*, lmargin, rmargin, /bx)

where bx represents the Boolean expression, and infile, inputstring, filelength, inpro, lmargin, and rmargin all have the same meanings defined in the previous paragraphs. When the sign separation option is used, the two numbers given at the beginning of this paragraph will be read as four separate items, the first and third items being the LISP atomic symbols + and - (Section VI). The primary purpose for this option is that of facilitating the parsing of arithmetic expressions via the GTL Input mechanism.

9.6 REMOTE TERMINAL INPUT-OUTPUT

9.6.1 The FILE REMOTE Declaration

The declaration FILE REMOTE may be used in a GTL program to declare a remote terminal file. It is a pseudo-declaration which takes the place of the usual pair of file declarations, and the associated input and output string declarations and the INPUT and OUTPUT statements; it is equivalent to the following declarations and statements:

```
FILE IN TWXF1 REMOTE(2,17);
DEFINE  TWXF2 = TWXF1#;
STRING TWXS1,  TWXS2(136)
      .
      .
      .
INPUT (TWXF1, TWXS1, 136);
OUTPUT (TWXF2, TWXS2, 136,0,72)
```

There are, in addition, other forms of the FILE REMOTE declaration which are variants of the following basic form:

FILE REMOTE (<u>file length in characters</u>, <u>outputstring right margin</u>,
    WAIT <u>wait time</u>, <u>break label</u>,
    <u>input time-out label</u>, <u>input overflow label</u>,
    <u>output impossible label</u>,
    <u>abnormal input condition label</u>,
    <u>input end-of-file label</u>, <u>input parity label</u>,
    <u>input buffer overflow label</u>)

where <u>file length in characters</u> is the length of the file in characters, and the length of TWXS1 and TWXS2 in characters, <u>outputstring right margin</u> is the

right margin of TWXS2. The maximum length of TWXS1 and TWXS2 is 136 characters. <u>Wait time</u> is the maximum time the system will wait for a response after an input message is requested (in seconds--must be an unsigned integer--maximum time is 300 seconds). <u>Break label</u> is the label specifying the location of the next statement to be executed after the "break" key is depressed on the remote station; after detecting a "break" condition, the program must READ from the terminal to clear the break--the remote user should just enter a left arrow. <u>Input time-out label</u> is the label preceding the statements which determine the action to be taken if the <u>wait time</u> is exceeded; <u>input-overflow label</u> is the label to which the system transfers if the input message exceeds the length of the input string; <u>output impossible label</u> is the label to which the system transfers if it becomes impossible to write on the remote terminal file. The <u>abnormal input condition label</u> is the label to which the system transfers if it becomes impossible to read from the remote terminal file. <u>Input end-of-file-label</u> is branched to when the user types ?END on the remote terminal; <u>input parity label</u> is transfered to when a parity error is detected on a READ; and <u>input buffer overflow label</u> branches to that label, indicating a buffer overflow has occurred during a READ. Any of the components of the FILE REMOTE declaration may be deleted, with two restrictions: (1) since the GTL compiler determines the meaning of a label by its relative position in the sequence of labels, they may be deleted only from right to left; the absence of a label contained in a sequence of labels may be indicated by placing an asterisk in the corresponding position; (2) if only one unsigned integer is specified, then it is assumed to be the <u>file length in characters</u>; to specify the <u>outputstring right margin only</u>, the <u>file length in characters</u> should

be replaced with an asterisk. For example,

FILE REMOTE (break label, *, input overflow label)

FILE REMOTE (*, 136, *,*,*,*,*, input end-of-file label)

The effect of deleting various components of the FILE REMOTE declaration is indicated in the table shown below. In the case of labels, it is assumed that the condition which affects a transfer to the label in question has occurred.

MISSING COMPONENT	EFFECT ON PROGRAM
file length in characters	set to 136 characters
outputstring right margin	set to 72 characters
wait time	set to 300 seconds
break label	GTL run time error #19
input time-out label	GTL run time error #16
input overflow label	"INPUT TOO LONG. RETYPE"
output impossible label	GTL run time error #9
abnormal input condition label	GTL run time error #17
input end-of-file label	GTL run time error #18
input parity label	"PARITY ERROR. RETYPE"
input buffer overflow label	"BUFFER OVERFLOW. RETYPE"

In the case of input overflow label, input parity label, and input buffer overflow label, the system automatically types the indicated message and then waits for more input from the terminal. The program cannot detect when the system does this, nor does it need to know, because recovery is handled for the user by the system. Under the Time Sharing MCP, all information after outputstring right margin is ignored.

Any labels appearing in the sequence of labels in the FILE REMOTE declaration need not be previously declared; they are declared by default.

The FILE REMOTE declaration must occur in the outermost block of the program, preferably immediately following the BEGIN.

9.6.2 FILE REMOTE Side-effects

A few side-effects occur when the FILE REMOTE declaration is used. They are listed below.

1) The left margin related to output is set to zero, and the right margin is set to 72, or the length of the outputstring right margin, if specified. A non-zero left margin may be set by

OUTPUT(TWXF2, TWXS2, file length in characters, lmargin, rmargin)

where lmargin and rmargin are unsigned integers.

2) The FILE REMOTE mechanism causes the printing device to be positioned at the beginning of a new line before the printing starts.

3) The left margin related to input is set to zero, and the right margin is dynamically set to the number of characters received in one transmission. A non-zero left margin may be set by

INPUT (TWXF1, TWXS1, file length in characters, lmargin)

where lmargin is an unsigned integer.

4) A psuedo end-of-file indication is normally returned after each remote terminal message, thus providing non-programmatic "punctuation" between remote terminal entries. Thus, in the case of SCAN and READCON (paragraphs 9.4.3 and 9.4.4), an end of file is indicated every other time

the function is called. In the case of READN, READ1, and READ (paragraphs 9.4.5 through 9.4.7), INSYM will contain a question mark character on every other read. Thus the following would read an S-expression from the terminal, L assumed to be type SYMBOL:

```
WHILE L := READ EQ QMARK DO;
```

Whenever a real end-of-file occurs ("?END"), it can be accessed by using an end-of-file label in the FILE REMOTE declaration (see paragraph 9.6.1). To have control over the psuedo end-of-file the programmer should execute

INPUT(<u>booid</u>)

where <u>booid</u> is a previously declared Boolean variable. As long as the value of <u>booid</u> is FALSE, the end-of-file condition will be returned; if <u>booid</u> is set TRUE, then the psuedo end-of-file is turned off.

5) If, after declaring FILE REMOTE, and having executed either an INPUT or OUTPUT statement to change the file type, it is desired to change back to the implicitly declared files of FILE REMOTE, then the following INPUT and/or OUTPUT statements may be used.

INPUT(TWXF1, TWXS1, <u>file length in characters</u>)

OUTPUT(TWXF2, TWXS2, <u>file length in characters</u>)

9.6.3 READ and WRITE Statements

Ordinary ALGOL READ and WRITE statements may be used with the remote terminal file declared by the FILE REMOTE declaration if the implicitly declared input and output string variable identifiers TWXS1 and TWXS2 are used in place of the file identifiers. For example,

```
READ(TWXS1, FORMATID, LISTID)

WRITE(TWXS2, FORMATID, LISTID)
```

The effective "file size" is determined by the lengths of the input and output strings.

9.6.4 READ TWX

The use of the statement READ TWX will cause the input string variable to be refilled with a remote terminal message regardless of what portion of the previous message has been scanned. Also, any remaining positions in the input string variable beyond the end of the input message are filled with spaces. For example, the following program segment will transfer successive remote terminal messages to a disk file (DISC) until the first character of the message is found to be an asterisk.

```
READ TWX;
WHILE TWXS1(0,1) ≠ "*" DO
  BEGIN
  WRITE(DISC,9,TWXS1);
  READ TWX
  END
```

9.6.5 WRITE TWX

The statement WRITE TWX will cause the contents of the output string variable to be written onto the remote terminal. It is equivalent to the two statements:

```
TAB := 72; TERPRI
```

For example, the following statements will write the contents of a disk file (DISC) onto the remote terminal:

```
WHILE TRUE DO
  BEGIN
  READ(DISC,9,TWXS2)[EOF];
  WRITE TWX
  END;
  EOF:
```

9.6.6 READN(TWX)

The function READN(TWX) is a special remote terminal version of the READN function (paragraph 9.4.5). This function is used for reading numbers from the remote terminal input message. It will ignore anything in the message which is not a number. If it reaches the end of a message without finding a number, it will print a question mark, followed by a space, and wait for a new message. For example, if the remote terminal message initially contains

2 3 5

then three successive calls on READN(TWX) will yield the three numbers. A fourth call will cause a question mark to be printed, and the system will wait for another number.

If READN(TWX) is used in a double precision context, its value will be a double precision number (see Section 3).

9.6.7 READN(TWXA)

The function READN(TWXA) works like READN(TWX) except that it returns alphanumeric strings and special characters, and ignores numbers. If an alphanumeric string exceeds 7 characters in length, it is truncated to the leftmost 7 characters.

Example:

```
PRINT #CONTINUE #;
IF READN(TWXA) ≠ "YES" THEN GO TO EXIT
        .
        .
        .
```

9.6.8 TWXNUM

The function TWXNUM is a Boolean-valued function which combines the functions of READN(TWX) and READN(TWXA). Its value is TRUE if a number is read, and FALSE otherwise. In either case, the resulting value will be contained in the standard variables INREAL and INDBL.

Example:

```
L:  WHILE TWXNUM DO AR[I := I + 1] := INREAL:
    IF INREAL ≠ "STOP" THEN GO TO L;
```

9.6.9 Conversational READ Statement

A familiar problem in writing remote terminal programs concerns the process of getting data into the program in a conversational fashion. Typically, the program prints a key word and reads the response from the user for each piece of data required. This construct simply helps mechanize this process. The syntax is as follows:

```
READ [V1, V2,---,VN]
```

where V1, V2,---, VN are each simple variables of type SYMBOL, BOOLEAN, ALPHA, REAL, or INTEGER. Strings and subscripted variables are excluded.

For each variable in the list, the name of the variable (up to 7 characters) is printed, followed by "=" and a "?". Then a value is read from the terminal.

The following table shows the variable types and the corresponding value type that is expected:

SYMBOL	atomic symbol (READ1)
BOOLEAN	numeric (READN(TWX))
REAL	numeric (READN(TWX))
INTEGER	numeric (READN(TWX))
ALPHA	alphanumeric (READN(TWXA))

If the proper value type is not entered the program types another "?" and reads from the terminal again.

For example, the following GTL program fragment:

```
BEGIN
ALPHA A; BOOLEAN B; INTEGER I; REAL R; SYMBOL S;
FILE REMOTE;
READ [A, B, I, R, S]
. . .
```

produces the following at run time:

```
A = ?
100 ←
?
ABCDE ←   (note ALPHA expected)
B = ?
0 ←       (1 = TRUE, 0 = FALSE)
I = ?
ABC ←
?
10 ←      (note INTEGER expected)
R = ?
10.2 ←
S = ?
ATOMICSYMBOL ←
. . .
```

9.7 STANDARD VARIABLES AND SYSTEM CONTROL PARAMETERS

9.7.1 The Standard Variables

The GTL language contains a set of six standard variables whose values reflect, and sometimes control, the operation of the GTL Input-Output mechanism. These variables, which are not declared in a GTL program, may be used in any context appropriate for the type of the variable. The names of these variables and their types are illustrated by the following declarations:

```
INTEGER TAB, COL
REAL INREAL
DOUBLE INDBL
SYMBOL INSYM
STRING INSTR(31), OUTSTR(64)
```

If any of these variables were actually declared in a GTL program, such declarations would override the implicit declarations and they could no longer have any special meaning. The standard variables INREAL, INDBL, INSYM, and INSTR are used to return the values of real numbers, double precision numbers, atomic symbols, and strings, respectively, which are read by the SCAN and READCON functions (paragraphs 9.4.3 and 9.4.4). The string variable OUTSTR is used primarily to return the string representation of a number converted by the NTS function (paragraph 9.2.16). The variables TAB and COL are discussed in the next paragraphs.

9.7.2 The Standard Variable TAB

The value of the standard variable TAB is equal to the current number of characters inserted into the output string variable by the output system. The only valid values of TAB are 0 to output string length, inclusive. At the beginning of a line of print, TAB is set to the left margin. When the string of characters representing a printable item is inserted into the output strnng, the value of TAB is increased by the number of characters inserted. Ordinarily, whenever the TERPRI function is called, either implicitly at the end of a PRINT statement, or explicitly, the statement

```
PRIN SPACE (RM - TAB)
```

where RM represents the right margin, is implicitly executed, and TAB is reset to the right margin. An example of the use of TAB is the following procedure which is used in a GTL program which produces flow charts from GTL programs:

```
PROCEDURE CLEAR(A,B); VALUE A,B; INTEGER A,B;
BEGIN
     TAB  := A;
     PRIN SPACE  (B - A);
     TAB  := RMARG;
     TERPRI
END OF CLEAR
```

This procedure has the effect of repeating the previous line of print with spaces filled in between the character positions A and B in the output string variable. The value of RMARG (see paragraph 9.7.4) is the current right margin in the output string variable.

9.7.3 <u>The Standard Variable COL</u>

The value of the standard variable COL is equal to the current number of characters in the input string which have been scanned by the read system. The only valid values of COL are 0 to <u>input string length</u>, inclusive.

When a logical record from an input file has been transferred to the input string variable, COL is set to the left margin of the input string variable. Each time a GTL read function reads an item, COL is increased by the number of characters read. This process continues until COL equals the right margin of the input string variable, at which point a new logical record is read from the input file and COL is reset to the left margin. When an INPUT statement is executed COL is set to the right margin, so that a logical record will be read before the scanning begins. An example of the use of COL is the following procedure which scans a quoted string from a card. In this procedure, which uses GTL string constructs (Section 5), it is assumed that CARD is the input string variable, T is a global integer variable, and that left and right margins are 0 and 72, respectively. It is further assumed that a quote mark has just been read.

```
PROCEDURE SCANQ;
BEGIN
     T := COL - 1;
     WHILE CARD(COL := COL+1,1) ≠ """" AND COL ≤ 72 DO;
     COL := COL + 1
END OF SCANQ
```

The quoted string, including the quote marks, will be found in the string designator

```
CARD(T,COL-T)
```

9.7.4 System Control Parameters

The values of certain control parameters used by the GTL Input-Output system may be accessed by a standard function called CONVAL. The CONVAL

function is used in the form

CONVAL(n)

where n represents an unsigned integer whose value designates the desired control parameter. Some of the values of n which may be used and the corresponding values of CONVAL(n) are listed in the table given below:

n	value of CONVAL(n)
2	maximum number of significant figures in single precision printed numbers
3	maximum number of significant figures in double precision printed numbers
4	sign-separation control (see paragraph 9.5.4)
13	left margin of output string variable (same as LMARG)
14	left margin of input string variable (same as LMARGI)
15	right margin of output string variable (same as RMARG)
16	right margin of input string variable (same as RMARGI)
17	equals 0 if OUTPUT statement has been executed; 1, otherwise
18	equals 1 if INPUT statement has been executed; 0, otherwise
19	number of words in output logical record
20	number of words in input logical record
21	equals 1 if output procedure is being used; 0, otherwise
22	equals 1 if input procedure is being used; 0, otherwise
23	same as LENGTH(OUTSTR)
24	same as LENGTH(INSTR)

The values of the GTL margin control parameters may be accessed through the following standard variables:

Standard Variable	Meaning	CONVAL Number
LMARG	left margin of output string variable	13
RMARG	right margin of output string variable	15
LMARGI	left margin of input string variable	14
RMARGI	right margin of input string variable	16

The asterisk forms of the INPUT and OUTPUT statements can be used to change the left and right margins (refer to paragraphs 9.3.3 and 9.5.3).

9.8 SAMPLE INPUT AND OUTPUT STATEMENTS

9.8.1 Card Reader

In order to read from cards the following declarations could be made:

```
FILE IN INFILE(2,10)
STRING CARD (80)
```

If the entire card is to be scanned, then the following INPUT statement

```
INPUT(INFILE, CARD, 80)
```

could be used. If the input cards are sequenced, then the following declarations could be used to check for the ordering:

```
STRING CARD(72,SEQ(8),OLDSEQ(8))
BOOLEAN PROCEDURE INPRO;
BEGIN LABEL EOF, EXIT;
READ(INFILE, 10, CARD) [EOF];
IF SEQ < OLDSEQ THEN PRINT #SEQUENCE ERROR#, CARD;
OLDSEQ := SEQ;   RETURN FALSE;
EOF:   RETURN TRUE;
EXIT:  END OF INPRO
```

In this case the INPUT statement should be

```
INPUT(INPRO, CARD, 80, 0, 72)
```

9.8.2 Line Printer

In order to print on a line printer, the following declarations could be made:

```
FILE OUT OUTFILE 16 (2,15)
STRING LINE(120)
```

to be used with the OUTPUT statement

```
OUTPUT(OUTFILE, LINE, 120)
```

9.8.3 Remote Terminal Files

In order to use the remote terminal files the following declarations could be used:

```
FILE IN F1 REMOTE(2,17)
DEFINE F2 = F1#
STRING STR1, STR2 (136)
```

Then the following INPUT and OUTPUT statements could be used:

```
INPUT(F1, STR1, 136)
OUTPUT(F1, STR2, 136,0,72)
STR2(72) := SPACE
```

Also the following form could be used,

```
FILE REMOTE
```

with many variations given in Subsection 9.6.

9.8.4 Listing of Input Cards

If a listing of the card input file on the PRINTER file is desired, then the following additional declaration is suggested:

```
BOOLEAN PROCEDURE INPRO;
BEGIN LABEL EOF;
     READ(INFILE,10,CARD)[EOF];
     WRITE(PRINTER,10,CARD);
     RETURN FALSE;
EOF: RETURN TRUE;
END OF INPRO
```

Then, in place of the INPUT statement given in paragraph 9.8.1 the following statement should be executed before using READ or READ1:

```
INPUT(INPRO,CARD,80)
```

These changes will cause each data card read by the program to be printed on the line printer.

APPENDIX A

EXAMPLES OF GTL PROGRAMS

Four examples of GTL programs are listed in the following pages. They are not intended to be examples of practical applications, but merely serve to illustrate some aspects of the GTL language. Other examples are included in Subsections 3.8, 4.9, 6.25, 7.6 and 8.8.

String Processing Example

```
BEGIN
COMMENT THIS PROGRAM MERGES TWO CARD IMAGE TAPES ACCORDING TO
      SEQUENCE NUMBER.

      FIRST TAPE ALREADY HAS SEQUENCE NUMBERS - SEQUENCE NUMBERS
      FOR SECOND TAPE SPECIFIED AT INTERVALS BY CARD IMAGES
      WHICH BEGIN WITH A "$" CONTAINING STARTING SEQUENCE NUMBER
      AND INCREMENTAL VALUE.  IF COLUMN POSITIONS 71 AND 72
      ON UNSEQUENCED CARDS ARE BLANK, "%A" IS INSERTED;

FILE IN TAPE1 (2,56,10);
FILE IN TAPE2 (2,56,10);
SAVE FILE OUT TAPE3 (2,56,10,SAVE 10);
STRING CARD1(72,SEQ1(8)),
      CARD2(72,SEQ2(8), NEWSEQ(INCR(8),SEQ(8)));
LABEL L1,L2,EOF;
L1:   READ(TAPE1,10,CARD1)[EOF];
L2:   READ(TAPE2,10,CARD2);
      IF CARD2(0,1) = "$" THEN
            BEGIN NEWSEQ := CARD2(64,16); GO TO L2 END;
      SEQ2 := SEQ;  COMMENT SET SEQ2 TO CURRENT SEQ NO;
      SEQ := + INCR; COMMENT INCREMENT CURRENT SEQ NO;
      WHILE SEQ1 LSS SEQ2 DO
            BEGIN
            WRITE(TAPE3,10,CARD1);
            READ(TAPE1,10,CARD1);
            END;
      IF CARD2(70,2) = "  " THEN CARD2(70,2) := "%A";
      WRITE(TAPE3,10,CARD2);
      IF SEQ1 = SEQ2 THEN GO TO L1 ELSE GO TO L2;
EOF:
END.
```

Lisp Processing Example

```
BEGIN
COMMENT THIS PROGRAM TAKES THE FIRST OF TWO LISTS,
        REVERSES IT AND ADDS THE SECOND LIST TO IT;
FILE REMOTE;
SYMBOL L1,L2;
SYMBOL PROCEDURE REVANDADD (X,Y); VALUE X,Y; SYMBOL X,Y;
        REVANDADD := IF NULL(X) THEN Y ELSE
             REVANDADD(CDR(X),CONS(CAR(X),Y));
LABEL EOF,START;
PRINT #GO AHEAD#,/;
START: WHILE L1:= READ EQ QMARK DO;
      IF L1 EQ "STOP" THEN GO TO EOF;
       WHILE L2:= READ EQ QMARK DO;
       PRINT #THE NEW LIST IS # REVANDADD(L1,L2),/,/;
       GO START;
EOF:
END.

EXECUTE REVADD
 RUNNING


GO AHEAD
(THIS IS A LIST)$
(THIS IS ANOTHER LIST)$
THE NEW LIST IS (LIST A IS THIS THIS IS ANOTHER LIST)

(THIS (IS A) (COMPLEX (LIST)))$
(THIS IS A SIMPLE LIST)$
THE NEW LIST IS ((COMPLEX (LIST)) (IS A) THIS THIS IS A SIMPLE LIST)

(REVERSE IS THIS)$
(OF A LIST)$
THE NEW LIST IS (THIS IS REVERSE OF A LIST)

STOP


 END REVADD 1.1 SEC.
```

```
BEGIN
COMMENT  THIS PROGRAM, USING LIST PROCESSING, ACCEPTS
      SYNTAX RULES AS INPUT, AND WILL PRODUCE RANDOM GENERATED
      STRUCTURES WHICH ARE SYNTACTICALLY CORRECT.  TO
      INPUT A RULE, THE COMMAND "RULE" IS FOLLOWED FIRST BY
      THE NAME OF THE RULE AND THEN ITS DEFINITION.
      A RULE IS ANY COMBINATION OF EITHER
      RULE NAMES, WHETHER PREVIOUSLY DEFINED OR NOT, OR AN
      "ATOM".  (AN "ATOM" IS AN ATOMIC SYMBOL AS DEFINED IN
      SECTION 6.12).  ALTERNATE RULES MAY BE SEPARATED
      BY A SLASH(/) WHICH MEANS "OR".
      THE SYNTAX OF "RULES"  IS SHOWN BELOW:

            SPECIFICATION   ::=  ATOM / ATOM SPECIFICATION
            RULE ::=  SPECIFICATION / SPECIFICATION "/" RULE
            RULENAME ::=  ATOM
            RULES ::=  RULENAME RULE

      TO GENERATE RANDOM STRUCTURES FROM THE RULE SPECIFICATIONS
      THE COMMAND "GENERATE""INTEGER" "RULE NAME" WILL CAUSE
      "INTEGER" RULE NAMES TO BE GENERATED
      ACCORDING TO THE SPECIFICATION OF THE "RULE NAME".
      TO STOP THE PROGRAM "STOP" SHOULD BE ENTERED.
      AFTER EVERY COMMAND, "GO AHEAD" IS PRINTED AND THE
      PROGRAM WAITS FOR MORE COMMANDS FROM THE TERMINAL;
FILE REMOTE;
SYMBOL R;
REAL X;
LABEL L;
SYMBOL PROCEDURE RULE;
      RULE := IF R := READ1 EQ "/" OR R EQ QMARK THEN NIL ELSE R . RULE;
SYMBOL PROCEDURE RULES;
      RULES := RULE . (IF R EQ "/" THEN RULES ELSE NIL);
PROCEDURE GEN(X);
      VALUE X;
      SYMBOL X;
      IF NUMBERP(X) OR NULL(CDR(X)) THEN
            BEGIN
            IF X NEQL "EMPTY" THEN PRIN X SPACE;
            END
      ELSE FOR X IN RANDOM(CDR X) DO GEN(X);

COMMENT PROGRAM STARTS HERE:          ;

L:    PRINT #GO AHEAD.#,/;
      READ TWX;
      IF R := READ1 EQ "RULE" THEN
            DO CDR(READ1) := RULES UNTIL R EQ QMARK
      ELSE
      IF R EQ "GENERATE" THEN
```

```
            BEGIN
            X := READN(TWX);
            R := READ1;
            FOR X DO BEGIN GEN(R); TERPRI; END;
            END
      ELSE
      IF R EQ "STOP" THEN
            BEGIN PRINT #EOJ#,/,/,/; EXIT; END
      ELSE PRINT #ILLEGAL INPUT#;
      GO TO L;
END.

EXECUTE SENGEN

 -BOJ- OSENGEN
GO AHEAD.
RULE SUBJECT BARRY/SUSIE/RECC/B5500/GTL
GO AHEAD.
RULE VERB LOVES/HATES/RUNS/PROCESSES
GO AHEAD.
RULE ADVERB REALLY/MOSTLY
GO AHEAD.
RULE ACTION VERB/ADVERB VERB
GO AHEAD.
RULE OBJECT SUBJECT
GO AHEAD.
RULE SENTENCE SUBJECT ACTION/SUBJECT ACTION OBJECT
GO AHEAD.
GENERATE 25 SENTENCE
GTL PROCESSES RECC
B5500 REALLY PROCESSES
BARRY REALLY RUNS
GTL PROCESSES
RECC REALLY RUNS BARRY
SUSIE HATES BARRY
GTL HATES SUSIE
B5500 REALLY RUNS
RECC MOSTLY LOVES RECC
GTL PROCESSES
GTL REALLY LOVES
GTL REALLY RUNS
BARRY RUNS
SUSIE REALLY RUNS
BARRY REALLY RUNS
B5500 REALLY RUNS
GTL PROCESSES
BARRY REALLY PROCESSES
BARRY MOSTLY PROCESSES
RECC LOVES
RECC REALLY LOVES B5500
```

```
SUSIE RUNS SUSIE
SUSIE PROCESSES B5500
BARRY REALLY PROCESSES RECC
BARRY REALLY LOVES
GO AHEAD.
STOP
EOJ
```

```
BEGIN
COMMENT A SYNTAX-DIRECTED PARSING PROGRAM USING THE REMOTE TERMINAL.
      THIS PROGRAM PRODUCES REVERSE POLISH FROM ANY ARITHMETIC
      EXPRESSION.  THE COMMAND "POLISH" FOLLOWED BY ANY ARITHMETIC
      EXPRESSION WILL RESULT IN THE PRINTING OF THE POLISH OF
      THAT EXPRESSION.  AN OPTIONAL TRACE FEATURE IS INCLUDED.  THE
      COMMAND "TRACE ON" WILL TURN THIS FEATURE ON, THUS GIVING
      A TRACE OF THE PARSING OF THE ARITHMETIC EXPRESSION INTO
      POLISH.  TO TURN THE TRACE OFF THE COMMAND "TRACE OFF" IS
      USED.  TO STOP THE PROGRAM, THE USER ENTERS "STOP" AND THE
      PROGRAM GOES TO END-OF-JOB.  ANY OTHER INPUT OR
      AN ILLEGAL ARITHMETIC EXPRESSION RESULTS IN AN ERROR
      MESSAGE.  AFTER COMPLETING A COMMAND OR DETECTING AN
      ERROR, THE PROGRAM PRINTS "GO AHEAD" AND WAITS FOR ANOTHER
      COMMAND FROM THE TERMINAL;
FILE REMOTE;
REAL CLASS;           % THE CLASS VARIABLE
BOOLEAN TRACE;        % THE TRACE VARIABLE
LABEL START,ERROR;
SYMBOL FORMAT         % SYNTACTICAL CLASS ASSIGNMENT
      *0
      = VARIABLE
      * NUMBER
      * "("
      * ")"
      * "*"
      * "x"
      * "/"
      * "+"
      * "-"
      * "POLISH"
      * "TRACE"
      * "ON"
      * "OFF"
      * "STOP"
      * EOF;
REAL FIELD CDRF [33:15];
%
%THE "GETNEXT" PROCEDURE
%
PROCEDURE GETNEXT;
      CLASS := CASE READCON(FALSE) OF
      BEGIN
      EOF;            % THE END-OF-FILE CLASS (SECTION 9.6.2)
      NUMBER;         % ILLEGAL NUMBER CLASS
      NUMBER;         % NUMBER CLASS - NUMBER IN INREAL
      CDRF(INSYM);% ATOMIC SYMBOL CLASS
      VARIABLE;       % NOT ON SYMBOL TABLE - GIVE VARIABLE CLASS
      END;
      COMMENT
```

```
       NOW TELL THE COMPILER THE NAMES OF THE GETNEXT PROCEDURE,
       WHICH VARIABLE IS THE CLASS VARIABLE AND
       THAT THE TRACE OPTION IS DESIRED.  NOTE THAT THE ERROR
       OPTION IS NOT BEING USED IN THIS EXAMPLE;
SYMBOL FORMAT *CLASS, GETNEXT, TRACE;
SYMBOL FORMAT AEXP; FORWARD;
SYMBOL FORMAT PRIMARY;
       [VARIABLE: PRIN INSYM SPACE ELSE
       NUMBER: PRIN INREAL SPACE ELSE
       "(", *AEXP, ")" ];
SYMBOL FORMAT FACTOR;
       [*PRIMARY,
       ["*", *PRIMARY, PRIN #* #; RETURN ELSE NIL]];
SYMBOL FORMAT TERM;
       BEGIN
       BOOLEAN TIMES;
       [*FACTOR,
       ["x"*"/": TIMES := CLASS =  ="x";
       *FACTOR, PRIN IF TIMES THEN #x # ELSE #/ #;
       RETURN ELSE NIL]];
       END;
SYMBOL FORMAT AEXP;
       BEGIN
       REAL MINUS;
       [*TERM,
       ["+"*"-": MINUS := CLASS;
       *TERM, PRIN IF MINUS =  ="-" THEN #- # ELSE #+ #;
       RETURN ELSE NIL]];
       END;

       COMMENT PROGRAM STARTS HERE;
       PRINT #RECC POLISH GENERATOR#;
       GETNEXT;
START:[EOF: PRINT #GO AHEAD#,/ ELSE
       "POLISH", *AEXP, TERPRI ELSE
       "TRACE",
             ["ON"*"OFF": TRACE := CLASS = ="ON";] ELSE
       "STOP", PRINT #GOOD BYE#,/,/,/; EXIT;]:ERROR;
       GO START;
ERROR:PRINT #ILLEGAL SYNTAX OR COMMAND#;
       COL := RMARGI; GETNEXT; GO START;
END.


RUN

 -BOJ- OPOLISH
RECC POLISH GENERATOR
GO AHEAD
POLISH (A+B)x(C-D)
```

```
A B + C D - x
GO AHEAD
TRACE ON
GO AHEAD
POLISH (A+B)x(C-D)

          CALL AEXP

                    CALL TERM

                              CALL FACTOR

                                        CALL PRIMARY

                                                  CALL AEXP

                                                            CALL TERM

                                                                      CALL FACTOR

                                                                                CALL PRIMARY
A
                                                                                PRIMARY = 1

                                                                      FACTOR = 1

                                                            TERM = 1

                                                            CALL TERM

                                                                      CALL FACTOR

                                                                                CALL PRIMARY
B
                                                                                PRIMARY = 1

                                                                      FACTOR = 1

                                                            TERM = 1
+
                                                  AEXP = 1

                                        PRIMARY = 1

                              FACTOR = 1

                              CALL FACTOR

                                        CALL PRIMARY
```

```
                    CALL AEXP

                        CALL TERM

                            CALL FACTOR

                                CALL PRIMARY
C
                                PRIMARY = 1

                            FACTOR = 1

                        TERM = 1

                        CALL TERM

                            CALL FACTOR

                                CALL PRIMARY
D
                                PRIMARY = 1

                            FACTOR = 1

                        TERM = 1
-
                    AEXP = 1

                PRIMARY = 1

            FACTOR = 1
x
        TERM = 1

    AEXP = 1

GO AHEAD
POLISH (A+(B*C)/D)+E*F

    CALL AEXP

        CALL TERM

            CALL FACTOR

                CALL PRIMARY

                    CALL AEXP

                        CALL TERM
```

```
                CALL FACTOR
                    CALL PRIMARY
A
                    PRIMARY = 1
                FACTOR = 1
            TERM = 1
            CALL TERM
                CALL FACTOR
                    CALL PRIMARY
                        CALL AEXP
                            CALL TERM
                                CALL FACTOR
                                    CALL PRIMARY
B
                                    PRIMARY = 1
                                    CALL PRIMARY
C
                                    PRIMARY = 1
*
                                FACTOR = 1
                            TERM = 1
                        AEXP = 1
                    PRIMARY = 1
                FACTOR = 1
                CALL FACTOR
                    CALL PRIMARY
D
                    PRIMARY = 1
                FACTOR = 1
/
            TERM = 1
```

```
+
                          AEXP = 1

                      PRIMARY = 1

                  FACTOR = 1

              TERM = 1

              CALL TERM

                  CALL FACTOR

                      CALL PRIMARY
E
                      PRIMARY = 1

                      CALL PRIMARY
F
                      PRIMARY = 1
*
                  FACTOR = 1

              TERM = 1
+
    AEXP = 1

GO AHEAD
TRACE OFF
GO AHEAD
POLISH (A+(B*C)/D)+E*F
A B C * D / + E F * +
GO AHEAD
STOP
GOOD BYE
```

APPENDIX B

REMOTE TERMINAL CHARACTER SET

The following is intended to serve as a convenient summary of the relation between the standard B 5500 character set and the remote terminal set. Other references should be consulted for a complete description (see, for example, Reference 7). In the following it is assumed that the file designated in the INPUT and OUTPUT statements are ALGOL file type REMOTE files, or the GTL FILE REMOTE declaration is used.

All letters of the alphabet and all digits may be printed on a remote station. The character ⊗ (multiplication sign) is printed as \. With the exception of six characters, all of the remaining non-alphanumeric characters may be printed on a remote station. The following characters serve as control characters and will have different effects, depending on which MCP the system is running under, either the Data Communications MCP (DCMCP) or the Time Sharing MCP (TSMCP).

Character	Effect on Output	
	DCMCP	TSMCP
≤	causes carriage return	prints ?
≠	causes line feed	prints ?
≥	causes station disconnect	prints ?
>	activates paper tape reader (X-ON character)	prints ?
←	causes preceding characters to be transmitted to remote station to be printed	prints ?
<	sends rub-out character	prints ?

Whenever the output string variable is ready to be printed, the GTL FILE REMOTE Output system first writes a blank line to cause a carriage return-line feed to be sent to the remote station, followed by the output string.

All letters of the alphabet and all digits may be entered on a remote terminal by depressing the keys marked with these characters. The remote terminal keyboard also contains keys for the following non-alphanumeric characters:

:	"	)
-	#	*
;	$	=
,	%	@
.	&	+
/	(	>

The blank character is entered by depressing the space bar.

The following additional characters may be entered with the keys indicated in the table below:

character	key
[	upper case K
]	upper case M
⊗	upper case L

The ! character causes the preceding portion of the line to be ignored. The ? character should not be entered since it may be interpreted by the system as a control character which is not a part of the intended input. The < character causes a hardware logical backspace each time it is depressed; i.e., if it is depressed n times it will cause n characters to be deleted from the current message. Since a message is transmitted from

a remote terminal in 28 character groups, characters can be deleted only from the current 28 character group. Under the TSMCP, the ' character causes a software logical backspace and is not dependent upon hardware configuration. The GTL FILE REMOTE system dynamically sets the RMARGI on input from a remote terminal to point to the last non-blank character. Thus RMARGI indicates the number of characters sent in the transmission, excluding the ←, which caused the information to be sent. When a GTL read function scans to RMARGI a pseudo end-of-file indication will normally result (see Subsection 9.6).

APPENDIX C

CONVAL FUNCTION

The values of various control parameters used by the GTL LISP and Input-Output systems may be accessed by a standard function called CONVAL. The CONVAL function is used in the form

CONVAL(n)

where n represents an unsigned integer whose value designates the desired control parameter. The values of n which may be used and the corresponding values of CONVAL(n) are listed in the table given below.

n	Value of CONVAL(n)
0	a newly-created random number between 0 and 1
1	value of current random number produced by CONVAL(0)
2	maximum number of significant figures in single precision printed numbers
3	maximum number of significant figures in double precision printed numbers
4	Sign-separation control
5	total number of words collected by garbage collector
6	number of times garbage collector is called
7	time (in seconds) required by last call on garbage collector
8	arithmetic value of the address of the first word in the freelist
9	first subscript of array described in paragraph 6.22.1
10	second subscript of array described in paragraph 6.22.1
11	same as COL

n	Value of CONVAL(n)
12	same as TAB
13	left margin of output string variable (LMARG)
14	left margin of input string variable (LMARGI)
15	right margin of output string variable (RMARG)
16	right margin of input string variable (RMARGI)
17	equals 0 if OUTPUT statement has been executed; 1, otherwise
18	equals 1 if INPUT statement has been executed; 0, otherwise
19	number of words in output logical record
20	number of words in input logical record
21	equals 1 if output procedure is being used; 0, otherwise
22	equals 1 if input procedure is being used; 0, otherwise
23	same as LENGTH(OUTSTR)
24	same as LENGTH(INSTR)
29	normally 0; will be set to 1 after a REMEMBER is executed, meaning that no LISP operation may be performed that causes a new LISP record to be generated when using automatic storage reclamation
30	initially 0; will be set to 1 after the first LISP record is created by the program; when set to 1, the RECALL statement cannot be used when using automatic storage reclamation
31	number of atomic symbols created by GENSYM

APPENDIX D

GTL RUN TIME ERROR MESSAGES

The following is a listing of the error numbers which may be generated during the execution of the GTL program. The form of the message is as follows:

-GTL ERROR Error Number Terminal Reference

Error Number	Meaning
1.	String designator reference beyond string variable boundary.
2.	Value of string expression longer than destination string variable in string assignment statement.
5.	File length specification in INPUT statement greater than input string variable length.
6.	File length specification in OUTPUT statement greater than output string variable length.
7.	Left margin greater than right margin or right margin greater than specified file length in INPUT statement.
8.	Left margin greater than right margin or right margin greater than specified file length in OUTPUT statement.
9.	Remote terminal "output impossible" condition detected--no "output impossible" label provided in GTL program.
10.	Supply of LISP records exhausted ("freelist empty").
11.	Value of Symbol expression is not a number (this error can occur when a Symbol-valued item is used in an arithmetic expression).
12.	Attempt to generate new LISP records after REMEMBER statement is executed (SYMBOL RECLAIM OPTION).
13.	Attempt to execute RECALL statement after run-time generation of new LISP records (SYMBOL RECLAIM OPTION).
14.	Attempt to apply a field designator to a null reference.
15.	Invalid field index.
16.	Remote read wait time exceeded (no label given).
17.	Remote abnormal read condition detected (no label given).
18.	Remote read end-of-file (READ) detected (no label given).
19.	Remote "break" key depressed (no label given).

APPENDIX E

REFERENCES

1. Paul W. Abrahams, Jeffrey A. Barnett, et. al., "The LISP 2 Programming Language and System," AFIPS Conference Proceedings, 29 (1966 Fall Joint Computer Conference), 661-676.

2. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, Lisp 1.5 Programmer's Manual, Cambridge, Massachusetts: The MIT Press, 1962.

3. Edmund C. Berkeley and Daniel G. Bolrow [editors], The Programming Language LISP: Its Operation and Applications, 2nd ed, Cambridge, Massachusetts: The MIT Press, 1966.

4. Burroughs B 5500 Extended ALGOL Language Manual, Burroughs Corporation, Detroit, Michigan, 1962.

5. Jerome Fieldman and David Gries, "Translator Writing Systems," Communications of the ACM, II, No. 2, 77-113 (February, 1968).

6. M. Levin, Lisp 2 Primer, SDC Document TM-2710/101/00, 1966.

7. Users Manual for B 5500 REMOTE TERMINAL OPERATIONS, Rich Electronic Computer Center, Georgia Institute of Technology, Atlanta, Georgia, June 1968.

8. Clark Weissman, LISP 1.5 Primer, Belmont California: Dickenson Publishing Company, 1967.

9. Nicklaus Wirth and C. A. R. Hoare, "A Contribution to the Development of Algol," Communications of the Association for Computing Machinery, 9, No. 6, 413-432 (June 1966).

10272 7201