THE UNIVERSITY OF MICHIGAN
COLLEGE OF LITERATURE, SCIENCE, AND THE ARTS
Computer and Communication Sciences Department

Technical Report

The CESSL Programming Language

Daniel R. Frantz
Ronald F. Brender

with assistance from:

Department of Health, Education, and Welfare
National Institutes of Health
Grant No. GM-12236
Bethesda, Maryland

and

National Science Foundation
Grant No. GJ-29989X
Washington, D.C.

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR
September 1971

Authors' present addresses:

Daniel R. Frantz
Logic of Computers Group
611 Church Street
The University of Michigan
Ann Arbor, Michigan 48104

Dr. Ronald F. Brender
Programming Research and Development
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

TABLE of CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 Background

This report describes a procedure-oriented language developed at the Logic of Computers Group for the IBM 1800. The language is called CESSL (*CE*llular *S*pace *S*imulation *L*anguage) for historical reasons. One of the authors (RFB) developed a simulation system for cellular spaces for which a new language with special constructs was required. The other author (DRF) helped finish the work and extended the language to the point where it now stands as an entity independent of the simulation system. RFB's Ph. D. thesis (1) describes the simulation system and language as of December, 1969. This manual is a considerably expanded description of CESSL and its extensions since then. A separate report (3) describes the simulation system in its present form.

In the two years it has been under development and in use, CESSL has grown and changed considerably. Most of this change has been in adapting it to be a good tool in the operating system in which it is used rather than an elegant, machine-independent language. It is fairly straightforward and standard in its capabilities with the exception of its data structuring facilities which are simple but effective. There are no general string manipulation capabilities, but *ad hoc* features allow some operations (I/O, assignments, and comparisons) to be performed nicely. CESSL is best described as a very handy procedural language which includes a few good ideas that might be of use elsewhere.

In the great family of languages it is closest to MAD/7090 (9) from which it draws inspiration and to which it owes a great debt. Comparing it to FORTRAN and ALGOL/58, it is closer to FORTRAN by not having a block structure,

but it approaches ALGOL in its conditional and loop structures.

1.2 Use

This report is basically a reference manual for CESSL, not a tutorial in programming--the authors assume that readers are familiar with at least one other procedural language. In addition, users should be familiar with the operating system for the IBM 1800.

Chapter Two contains the complete description of the externals of the language. Chapter Three contains the mechanics of using the compiler in TSX. Chapter Four describes the code produced by the compiler and is included for the benefit of those users who are wont to perform debugging at the machine language level. Chapter Five covers a few special topics which refused to be included elsewhere.

1.3 Facilities

The following is a brief introduction to the hardware and software referred to throughout this manual. Complete information can be found elsewhere on these topics--this is meant only to lessen initial reader confusion.

The computing facilities at the Logic of Computers Group (LOCG) consists of two small computers (an IBM 1800 and a DEC PDP-7), each with its own peripherals, connected by a locally designed interface which has the capability of performing core-to-core transfers. See Figure 1.1

The operating system for the 1800 is TSX, supplied by IBM (6,7,8). Part of the system is the "Nonprocess Monitor" which is a batch monitor which reads control and data cards form the card reader, directs the storage of compiled or assembled programs, and starts the execution of user programs.

The Logical File System on the 1800 is a subsystem (actually a collection

of subroutines) which can create and maintain sequential character (or files on the 1810 disks (2).

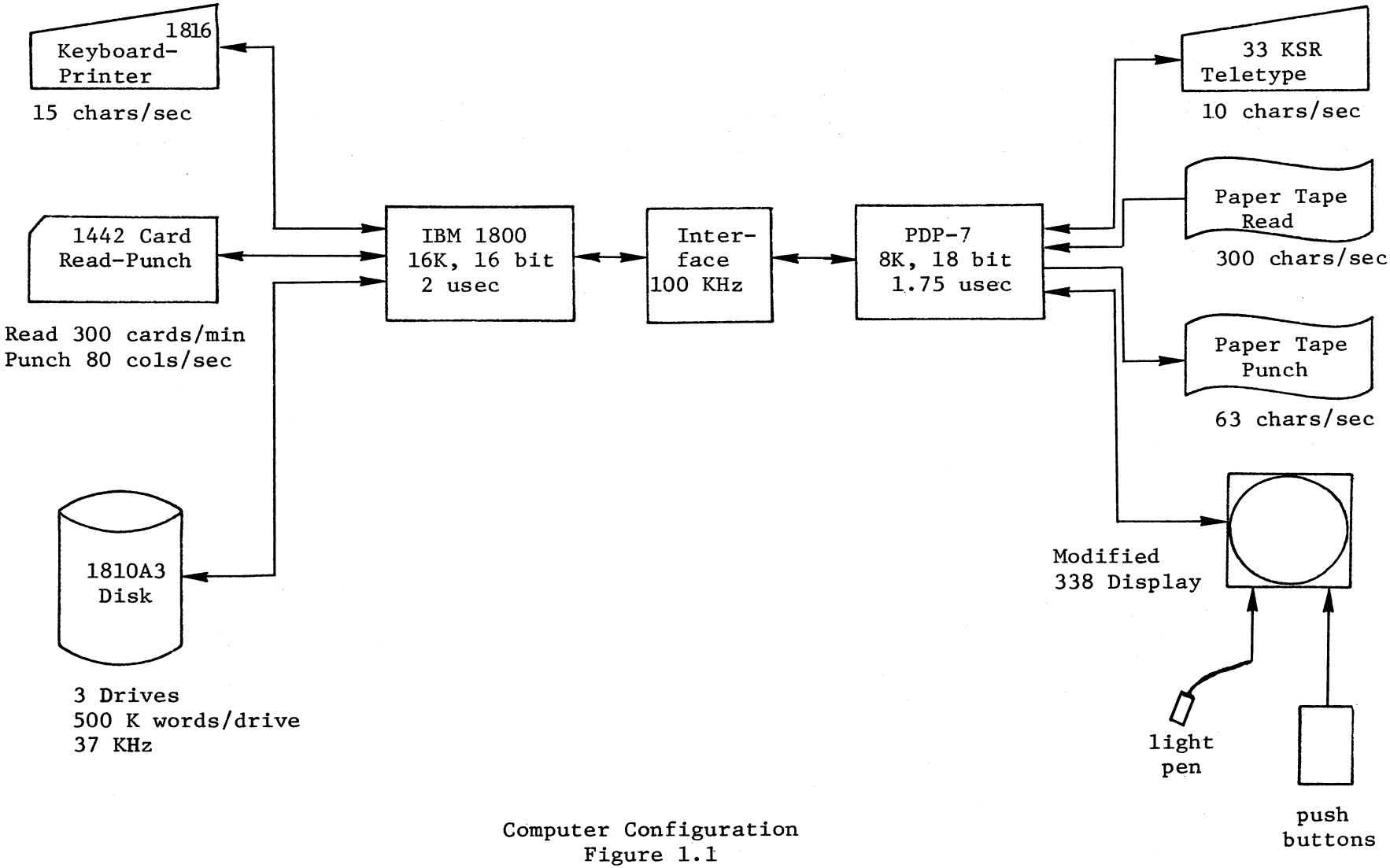
The "copy port" is a logical path between the 1800 and the PDP7 through which characters (i.e. 8 bit bytes) can be passed back and forth between the two machines by means of system software at either end. A common use is for an 1800 program to send a stream of characters via the copy port to a cooperating program in the PDP7 which will then display them on the CRT (338 Display), thereby achieving rapid, albeit volatile, readout.

1.4 Acknowledgments

The authors wish to thank those member of the Logic of Computers Group who have suffered through the development of the compiler and this documentation--which were often contradictory. Indeed, they can truthfully say--as few others in this world can--"But it worked yesterday and I didn't change a thing". Their courage in the use of CESSL is to be praised, although their wisdom may be doubted; their criticisms and suggestions have often been useful.

Special thanks are due to Dennis P. Geller who did much of the support programming for the I/O Statements and Hal A. Rosenblit for the DATA statement.

Finally, thanks to Jan McDougall for typing numerous drafts on our way to increasing the clarity of the presentation, and to Monna Whipp for the final draft.



Computer Configuration
Figure 1.1

CHAPTER TWO

PROCEDURAL ASPECTS OF CESSL

The source statements of the language are read from the input medium (card reader, file storage, etc.) in free format. End of record indications (end of card, carriage return, etc.) are treated the same as a "space" character so that statements may extend over more than one physical record (or "line"). Informally, the basic unit recognized by the compiler is a *construction* which is a string of characters (syntax given later) followed by a semicolon. More than one construction may occur on a given line.

The compiler recognizes *statements* which are a sequence of one or more constructions of a particular type. A *program* is a sequence of statements.

In describing the language we shall use the following conventions and notations. Example sections of source coding will be on separate lines with double indenting to keep them distinguished from the describing text. Source words are always in capital letters and this is also a useful cue. Syntactic descriptions of the language grammar will use a variation of the more common BNF notation.

Syntax will be described in its production (rather than reduction) form, e.g.,

<A> → B <C>

which may be read "the non-terminal symbol <A> may be replaced by the symbol B followed by the non-terminal <C>". Tall square brackets will be used to designate several mutually exclusive possibilities, of which one must be present. Thus,

$$\langle A \rangle \rightarrow \left[\begin{array}{ll} X & \langle L \rangle \\ B & \end{array}\right] \langle C \rangle$$

may be considered a shorthand for the two productions:

$$\langle A \rangle \rightarrow X \ \langle L \rangle \ \langle C \rangle$$

$$\langle A \rangle \rightarrow B \ \langle C \rangle$$

The latter will often be simplified by not repeating the left side, as in:

$$\langle A \rangle \rightarrow X \ \langle L \rangle \ \langle C \rangle$$

$$\rightarrow B \ \langle C \rangle$$

Curly brackets will be used to indicate a sequence that may be repeated an arbitrary (possibly null) number of times. If sub- and superscripts follow the right bracket, they are interpreted as minimum and maximum number of repetitions, respectively. Thus

$$\langle A \rangle \rightarrow X \ \{Y\}$$

describes the same collection of terminal strings as

$$\langle A \rangle \rightarrow \langle A \rangle \ Y$$

$$\rightarrow X$$

and

$$\langle A \rangle \rightarrow \{X\}_1^3 \ Z$$

is shorthand for

$$\langle A \rangle \rightarrow XZ$$

$$\rightarrow XXZ$$

$$\rightarrow XXXZ$$

We will not be concerned here with the subtleties of the differing parsing trees that might result from alternate interpretations of these shorthands. We distinguish between a description grammar which is used to convey the language to users, and an implementation grammar which is used explicitly for

syntax directed parsing. Since we are primarily concerned with describing a language, the above conventions are both a convenience and in some cases more intuitively meaningful than a comparable BNF expression.

We admit that this dichotomy of grammars leaves ample opportunity for conflict and inconsistency between the description and the implementation. But since the compiler currently implemented used syntax directed methods only at the level of expressions and assignment statements, there is no formal implementation grammar for many aspects of the system. Accordingly, we will not be too embarrassed to occasionally use a suggestive non-terminal symbol such as <integer constant> without anywhere giving a syntactic definition of the symbol. The intention will be clear, and will accurately convey much semantic information about what is actually required by the compiler.

2.1 Lexical Format

The following table specifies the *characters* and names for sets of characters recognized by the compiler.

Alphabetics	ABCDEFGHIJKLMNOPQRSTUVWXYZ$'
Numerics	0123456789.
Quote	"
Space	( ), carriage return, and tab
Special Characters	+-*/,:;()=<>!?@
Alphanumerics	Alphabetics and Numerics
Non-alphanumerics	Quote and space and special characters

The other graphics from the input medium may be recognized only in TEXT constants; these are: tab, carriage return, ¢, |, &, ¬, #, % . Note especially that $ and ' are alphabetic characters and period is a numeric.

The basic lexical unit is the atom. Basically, an *atom* is defined by one of the following three cases:

1) Any of the Special Characters (e.g., +,!).
2) Any string of Alphanumerics delimited by Non-alphanumerics, (e.g., ABC, D12, 156, 12.4E5). See below for the special cases REAL constants and hexadecimal INTEGER constants.
3) Any string of characters enclosed in quotes - a quote character may be included in such a string by two quotes in succession (e.g., "AB-1", "A""B").

Note that "space" itself is not an atom. "Carriage return" and "tab" characters are equivalent to space except when they occur in a TEXT constant. Space may always, and must sometimes, be used as a delimiter between atoms. For example, 123 is one atom, while 1 2 3 is three atoms.

Lexically, a *construction* is any sequence of atoms (except semicolon) ended by a semicolon. Note that the semicolon is considered part of the construction. Thus, the following construction consists of 14 atoms:

```
AX = A(5) + B(1, P$T);
```

Note here that $ is an alphabetic character and does not separate atoms. Thus

```
ABC$NE$3
```

is one atom, not three. The same comment applies to prime (').

A *statement* consists of a given number of constructions concatenated together and satisfying certain constraints. The statements of the language will be developed in detail below.

A *numeric atom* is one of the following:

1) An atom consisting only of numeric characters and no period, treated as a decimal INTEGER constant (see Section 2.3 for definition of types and constant values).
2) A question mark (?) followed by an atom consisting only of numeric characters or the letters A-F, and no periods, treated as a hexadecimal integer constant. Note that this is the only legal use of ? outside of TEXT constants.
3) An atom consisting of numeric characters, including period, treated as a REAL constant.
4) A sequence of characters in what is commonly called "E-notation": a sequence of numeric characters followed by E, optionally followed by + or -, followed by one or two numerics. See Section 2.3 for examples.

An *alphabetic atom* is an atom of only alphabetic characters. A *keyword* is a predefined alphabetic atom (see list is Figure 2.8). A λ-atom

(or just plain λ) is an alphanumeric atom (beginning with an alphabetic) which is not a numeric atom and not a keyword. λ-atoms may be used as variable names or labels, defined as entry points, etc.

If the first atom of a construction is a λ-atom and the second a colon, then the λ-atom is implicitly defined as a constant of type LABEL. (See 2.2). Labels may appear only on executable statements.

Numeric atoms may contain a maximum of 15 characters; λ-atoms and TEXT atoms may contain a maximum of 62 characters.

2.2 Attributes

All atoms have attributes associated with them which enable the compiler to interpret statements in the correct manner. Some atoms have predefined attributes, as, for example, +, which has the attribute of "dyadic operator", and "precedence 30", among others. The atom ABC has the attribute "λ-atom" by the definitions above. As a notational convenience, if an atom has an attribute, we say that that atom "is" an attribute, or is in that attribute class. For example, ABC "is" a λ-atom, or it is in the class of λ-atoms.

Atoms which are λ-atoms may have other attributes as well. These attributes are assigned implicitly or explicitly by the statements of the language. The following is a brief summary of the possible attributes of λ-atoms. A further explanation of each attriuute will be found in the following sections.

λ-atoms may have at most one of the attributes: data type name, entry point name, and variable. An example of the predefined data type names is INTEGER. An entry point name is the name of a subroutine entry point, explicitly declared by the user.

Variable λ-atoms (also called "variables") are the usual variables and labels of normal programming languages; that is, they are names that have or may be assigned values for later use. Variable atoms may have other attributes, including any or all of the following: a data type, FUNCTION, formal parameter, and LIBF.

In general, the language allows only one use for an atom; that is, only one set of attributes may be assigned to an atom, and that set applies at every occurrence of the atom. For example, no atom could be both a defined data type name and a label even though the correct use could almost certainly be inferred from the context. However, the dual use of "-" as both a monadic

(one operand) and dyadic (two operand) operator is so pervasive in other programming languages that is has been explicitly accomodated here.

2.3 Data Types

Programs operate on data to achieve the desired results. The description of the data is called the *data type* or *mode*, and every variable or constant must have as an attribute some data type.

There are five predefined data types, called *primitive data types*:

INTEGER

REAL

BOOLEAN

LABEL

TEXT

Each of the above atoms is a keyword whose use in the language is reserved for describing data (see Section 2.4, Declarations). In general, variables and constants of the language may be of any of the above types.

In this implementation, the above descriptions imply the following. An INTEGER is a whole-valued number in the range (-32,768,+32,767). A REAL is a fractional-valued number with an approximate range of $(10^{-39}, 10^{+38})$, and a precision of about seven decimal places (i.e., numbers which differ in the eighth place are identical to the program). A BOOLEAN is a variable which may take on one of two values called TRUE and FALSE (internally one and zero, respectively, in this implementation). A LABEL takes the value of a position in the program; that is, its value is the "address" of a statement of the language. A TEXT *variable* contains two EBCDIC (Extended Binary Coded Decimal Interchange Code) characters.

REAL variables take two words of storage, while all other primitive variables take one word of storage. (This is a restriction of the IBM 1800.)

Constants (i.e., fixed values) of the above types are recognized by their lexical properties as follows:

1) An INTEGER constant is a numeric atom without a period (e.g. 1524, ?1A4).
2) A REAL constant is a numeric atom with exactly one period, or an atom in "E-format" (e.g., 121.3, 11OE4, 123.4E-15).
3) BOOLEAN constants are the atoms TRUE and FALSE.
4) A LABEL constant is any λ-atom that occurs in the label field of an executable statement (i.e., it is the first atom, and the second atom is a colon).
5) A TEXT constant is a string of characters enclosed in quotes. Note that a text constant may contain more than two characters, while a text variable has only two. (In this implementation, text constants are stored two characters per word, with a trailing binary zero inserted to fill out an odd count. Exactly as many words are used for the constant as are needed. E.g., "ABCD" is a constant of two words, and "NOW IS THE TIME" has eight words.)

Other data types may be defined by the user (See Section 2.4.1.2, DEFINE).

It is perhaps appropriate at this point to say a few words about type TEXT and its role in CESSL. The compiler does not directly provide full scale string manipulation facilities (e.g. there is no free storage area for variable length strings, no concatenation or substring selection operators). However, a few conveniences are available. The user may define a data type (Section 2.4.1.2) describing an array of items, each of type TEXT, by a statement of the form:

```
DEFINE textn ARRAY TEXT SIZE n;
```

For each n (an integer) there is a separate data type of this sort definable--any type from this class is called TEXTARRAY (this is used as a meta-type

word--it describes a class of types or a type from the class). A variable which is declared to be a TEXTARRAY has special properties: it acts almost as if it were of type TEXT. It may appear in I/O statements, causing many words to be transferred in or out of memory (instead of the usual one), and it may appear in assignment and comparison statements with TEXT constants longer than two characters, thus effecting a multiple word data movement or comparison. These capabilities (described in the appropriate places), plus the ability to refer to individual items in the array make it possible to perform many useful string manipulations.

2.4 Program Statements

The statements of the program are of three types: Declarations, Executable, and Comments. The declarations (Figure 2.2) assign attributes to atoms or give instructions to the compiler or operating system, and the executable statements (Figure 2.3) specify operations to be performed. Comments are statements which are ignored by the compiler, their sole purpose being to provide documentation. A comment is a statement which begins with an asterisk (*) and ends with a semicolon(;).

2.4.1 Declarations

There are several declaration statements: DECLARE..., DEFINE..., NORMALMODE, SUBSTITUTE, DATA, INCLUDE, INTERNALFUNCTION, ENTRY, ENDFUNCTION, and ENDPROG. In addition, there are several forms of DECLARE and DEFINE. INTERNALFUNCTION, ENTRY, and ENDFUNCTION relate directly to subprogram segments and are discussed in Sections 2.4.3 and 2.4.4. In general, declarations may appear anywhere in a program, the only restrictions being on sequence dependencies in DECLARE and DEFINE, and placement of DATA, ENTRY, and INTERNALFUNCTION as noted in the appropriate sections.

2.4.1.1 DECLARE

The first form of DECLARE assigns the attribute of a particular data type to each of a series of λ-atoms:

DECLARE <type>: <λ-list>;

The <type> must be a λ-atom which is from the primitive data type list above (Section 2.3), or which has previously been defined as a type by the user (below, Section 2.4.1.2 --that section also contains an alternate method of assigning types). For example:

```
DECLARE REAL: A,B,C;
DECLARE INTEGER: X,Y,GEORGE;
DECLARE I10: LONG;
```

would establish A, B, and C as real-valued variables, X, Y, and GEORGE as integer-valued variables, and LONG as a variable of type I10, which must have been already defined by the user.

The second form of DECLARE assigns the attribute FUNCTION to each of a series of λ-atoms:

DECLARE FUNCTION: <λ-list>;

For example:

```
DECLARE FUNCTION: RUNGE, CUBE;
```

FUNCTION attribute is used when it is necessary to pass the *name* of a subroutine in a subroutine parameter list rather than the value of an invokation of that subroutine. Thus any λ-atoms which are external names must conform to the naming conventions of the operating system--in particular, they may have at most five characters in their name. Formal parameters and internal functions, of course, are not external. See Section 2.4.2.1, Assignment statement, for examples.

The third form of DECLARE assigns the attribute LIBF to each of a series of λ-atoms:

DECLARE LIBF: <λ-list>;

For example,

DECLARE LIBF: DISKN, FADD;

LIBF attribute must be assigned to external subroutine names which are entered by a TSX "LIBF" call. User-compiled subroutines are not usually so entered--most LIBF routines are TSX system subroutines. It is not possible to compile such a routine in CESSL.

The fourth form of DECLARE assigns a name to the program and states that this is a "main" program, i.e., not a subroutine:

DECLARE λ NAME;

For example,

DECLARE NOMEN NAME;

The name is used for reference by the operating system so that it must conform to the operating system's naming conventions. In addition, the name must occur as a label on some executable statement. The statement so labeled will be the first statement of the program executed.

2.4.1.2 DEFINE

DEFINE is used primarily to define new data types and secondarily to assign the newly defined types to variables. Two types of composition operations are available to generate data structures more complex than the primitives. The simpler of these is the fixed length array:

DEFINE λ ARRAY <type> SIZE <integer constant>;

The interpretation is that λ is defined as a data type name which identifies a data structure consisting of a fixed number (given by the <integer constant>) of elements, all of which are of type given by the <type>. Thus, to declare A a variable having the structure of an array of five REALs and B a variable having the structure of a square array of seven by seven INTEGERs, the following declarations suffice.

```
DEFINE REAL5 ARRAY REAL SIZE 5;
DECLARE REAL5: A;
DEFINE INT7 ARRAY INTEGER SIZE 7;
DEFINE SQINT7 ARRAY INT7 SIZE 7;
DECLARE SQINT7:  B;
```

The second composition operation provides for the definition of a *block* of contiguous data whose elements may be of diverse types. Blocks are also called component structures or structured variables. The form of the statement is:

DEFINE λ BLOCK < <type list> >;

For example:

DEFINE QQSV BLOCK <REAL, INTEGER, INTEGER>;

specifies that QQSV is a type name referring to a block consisting of a real number followed by two integers.

Either DEFINE operation may be composed with itself or with the other, thereby allowing complex data structures to be constructed in a hierarchical fashion. By convention, the same structure may not be given more than one

name.

Components of a complex data type may be identified by a subscript-list (which is a parenthesized expression list--See Figure 2.4) following the variable name. Subscripts are interpreted from left to right as identifying a lower data type in the hierarchical description of the data structure. Figure 2.1(a) presents a somewhat involved example.

Items in a subscript list may be expressions of type INTEGER or REAL only. If REAL, the value is converted to INTEGER by truncation. All subscript items must have a positive value--there is no zeroth item in an array--and the value may not exceed the number of elements in the array or block defined.

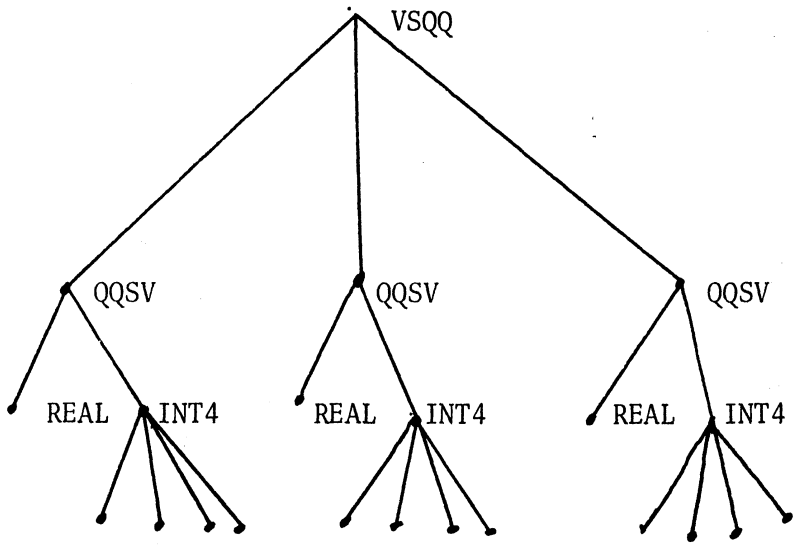
In general, a subscript choosing an element from a BLOCK must be a constant; otherwise the data type selected is unknown and the proper code cannot be produced by the compiler. The SUBSTITUTE statement (Section 2.4.1.5) provides a means of using mnemonic names which "stand for" constants, allowing meaningful subscripts. A variable subscript may be used if the user intervenes to tell the compiler what the resulting type will be by use of the "@" compile-time operator (Section 2.4.2.1).

In order to refer conveniently to primitive elements of a variable's data structure, we shall sometimes speak of the lexigraphical ordering of the fields of a data structure. A *field* is a substructure which has a primitive type. Field f_1 *precedes* field f_2 if in the subscript notation for referring to the respective fields, the subscript designating f_1 is numerically less than the subscript designating f_2 in the first position in which they differ, reading from left to right. In Figure 2.1(a), ABC (1,1) precedes ABC (1,2,3) which precedes ABC (2,1), etc. Since this clearly gives a linear ordering, we shall speak of the first field, second field, etc. One may

```
DEFINE INT4 ARRAY INTEGER SIZE 4;

DEFINE QQSV BLOCK <REAL, INT4>;

DEFINE VSQQ ARRAY QQSV SIZE 3;

DECLARE VSQQ: ABC;
```

ABC	is of type	VSQQ
ABC (2)		QQSV
ABC (2,1)		REAL
ABC (1,2)		INT4
ABC (3,2,1)		INTEGER
ABC (1,2,7)		undefined

2.1(a)



INTEGER

Field	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Word	0	2	3	4	5	6	8	9	10	11	12	14	15	16	17

2.1(b)

Figure 2.1

Example Data Structure Definition and Related Notation

equivalently think of drawing the "structure tree" of a data type, and then numbering the endpoints from left to right as illustrated in Figure 2.1(b). The figure also shows the layout in storage of a variable of this type, reflecting the fact that storage is contiguous and that REAL variables take up two storage words.

As a convenience, the type just defined in the DEFINE statement may be assigned to a list of λ-atoms, as in the DECLARE statement. This is accomplished by replacing the terminal semicolon with a colon and appending a list of λ-atoms. I.e. the full form of the DEFINE statement is:

```
                  [ARRAY <type> SIZE <integer constant>]  [;          ]
DEFINE λ          [                                    ]  [           ]
                  [BLOCK   < <type-list> >             ]  [:<λ-list>;]
```

For example, declaring A to be an ARRAY of five REALs might be performed as follows:

```
DEFINE REAL5 ARRAY REAL SIZE 5: A;
```

Of course, it is still possible to use the DECLARE statement to assign the type to additional variables.

WARNING! As of the publication date of this manual there is a restriction on the structure of data types. This restriction may be lifted sometime in the future, but until then great care must be taken. The reason for it is that in the TSX system REAL variables must be "even-aligned", that is, their addresses in computer memory must be even. It is the user's responsibility to ensure that REAL variables are even-aligned in structured data types.

All structured variables are assigned even addresses for their first words by the compiler. Thus, to ensure even-alignment of REAL fields in a structure, the user need only follow these three steps:

1) Draw the "structure tree" for the data type of every variable,

as in Figure 2.1(b).

2) Starting from the left-most field, label each field with the number of words by which that field is separated from the first field. INTEGER, BOOLEAN, TEXT, and LABEL fields occupy one word, and REAL fields occupy two words. The first field gets zero, and each successive field to the right gets a number either one or two higher depending on the type of the field on the left. This has already been done in Figure 2.1(b).

3) Each REAL field must then be associated with an even number. If it is not, the definition of the DATA type for that variable must be altered. The alteration will usually take the form of an additional INTEGER field or a permutation of the fields.

For eaample,

```
DEFINE BL BLOCK <INTEGER, REAL> ;
```

does not meet the test, while

```
DEFINE BK BLOCK <REAL, INTEGER> ;
```

does.

However,

```
DEFINE BKA ARRAY BK SIZE 5;
```

does not meet the requirements even though the component substructures do, while

```
DEFINE BKB BLOCK <BK, BL> ;
```

meets the requirements although the individual components do not.

The reader should carry out the above three steps for these examples to satisfy himself that he understands the restriction. The compiler does not check for violations.

2.4.1.3 NORMALMODE

The statement

NORMALMODE <TYPE>;

assigns the data type attribute <type> to all variable λ-atoms which are not explicitly assigned some type. For example:

NORMALMODE INTEGER;

assigns INTEGER type to all undeclared variables.

2.4.1.4 DATA

The DATA statement allows variables to be preset to specific values prior to execution of the program. Presetting does not alter the way in which a variable may be used or changed in value. It merely supplies a value which the variable will take on at the time the program is loaded. Variables which do not appear in DATA statements have an indeterminate initial value. The form is:

DATA λ <value>;

The λ-atom specifies the name of the variable to which the value is assigned. The <value> usually specifies a constant, as described below, but may be the name of a variable for the special uses described in Section 5.5. Such uses require great care and are recommended only to those familiar with Chapter Four, Generated Code and Run-time Support.

For primitive type variables, <value> is just a constant. E.g.,

```
DECLARE INTEGER: NUMB;          DATA NUMB 213;
DECLARE REAL: X;                DATA X 2.102;
DECLARE BOOLEAN: A;             DATA A TRUE;
DECLARE LABEL: ABC;             DATA ABC XYZ;
DECLARE TEXT: BLANK, CR;        DATA BLANK " ";
                                DATA CR ?15;
```

The type of the constant should match the type of the variable, although the compiler does not check, allowing special uses, as in the last example. For structured variables (i.e., variables with non-primitive data types), <value> is given by a list of constants (separated by commas) which specifies successive fields of the variable, in the lexigraphic order of Section 2.4.1.2. It is the user's responsibility to ensure that the types of the constants given are correct. (This is especially important in the case of REAL fields

whose storage takes up two computer words whereas all other constants take up only one word.) For example:

```
DEFINE INT3 ARRAY INTEGER SIZE 3: DEF;
DATA DEF 1,2,3;
DEFINE AMESS:  BLOCK  <INTEGER, BOOLEAN, REAL, LABEL, TEXT, TEXT>;
DECLARE AMESS: SLOP;
DATA SLOP 1, TRUE, 4.312, ABC, "TE", "ST";
```

where ABC must occur as a label in the program.

The special case of a TEXT constant longer than two characters (thereby occupying more than one word) is accomodated automatically by the compiler by breaking down the constant into one word (two character) groups and assigning them to successive locations. For example,

```
DEFINE TEXT 10 ARRAY TEXT SIZE 10: ZAPPO;
DATA ZAPPO "NOW IS THE TIME FOR";
```

is equivalent to the more specific:

```
DATA ZAPPO "NO", "W", "IS", ...;
```

As with all text constants, the case of an odd number of characters leaves the last word with a single character in the left half (high order) part of the word and a binary zero in the right half (low order) part. The null TEXT constant "" provides one zero word.

The same constant may be assigned to successive fields by the use of a multiplicative factor as in the FORTRAN DATA statement. That is, a specification of the form n*c, where n is an integer and c is any constant, is equivalent to n c's separated by commas. For example,

```
DEFINE INT100 ARRAY INTEGER SIZE 100: BIGI;
DEFINE REAL100 ARRAY REAL SIZE 100: BIGR;
```

```
DEFINE T4 ARRAY TEXT SIZE 4: DITTO;
DATA BIGI 100*0;
DATA BIGR 25*0.0, 25*1., 25*2., 25*3.;
DATA DITTO 2*"SAME";
```

results in all the elements of BIGI being initialized to zero, the first 25 elements of BIGR being set to 0.0, the next 25 to 1.0, the next 25 to 2.0, the last 25 to 3.0, the first and third elements of DITTO being set to "SA" and the second and fourth to "ME".

A DATA statement which does not provide enough data items to preset all fields of a variable will result in the setting of all unspecified fields to zero. Thus a variable may be set to zero by simply specifying, for example:

```
DATA BIGI;
```

which sets 100 words to zero.

The DATA statement may not be used to preset a value for any variable appearing as the left half of an EQU pair, nor for any variable which has attributes FUNCTION or formal parameter.

The DATA statement may not appear just anywhere in the program as most of the other declarations. In most cases, DATA should not be "flowed into"; i.e. it must be placed at the beginning or end of the program or, if it appears in the middle, it should be after a GOTO or RETURN statement. The compiler will not check for violations of this restriction--crashes will usually occur at run time. This compilation of DATA "in-line" and the use of variable names in a data statement allow special applications of considerable flexibility, as described in Section 5.4.

2.4.1.5 SUBSTITUTE

The SUBSTITUTE declaration directs the compiler to substitute one atom for any occurence of another atom *at parse time*. Its primary use is to allow mnemonic names to be used in place of integer constants (especially when used as subscripts). This declaration is recommended to resolve the awkward choice resulting from block data types. A numeric subscript is non-intuitive but gives a known compile-time data types, while a heuristically chosen and suitably valued variable does not permit a known compile-time data type. Use of a SUBSTITUTE parameter removes the problem. The other use of SUBSTITUTE allows the user to refer to any atom (reserved or otherwise) by another name.

The form of the statement is

SUBSTITUTE {(λ{,}$_0^1$ [<integer constant> / <any atom>])};

It consists of a series of ordered pairs. The first of the pair is replaced by the second wherever encountered. The first must be a previously unused λ-atom; the <integer constant> may be negative. For example,

```
SUBSTITUTE  (XYZ,12)   (ALPHA,0);
X = ABC(XYZ,ALPHA);
```

is equivalent to

```
X = ABC(12,0);
```

Note that the substitution becomes effective at the point of definition and is not retroactive to previous statements. Further, the substitution is actually performed before syntactic parsing and hence, the integer constant is actually used by the parser. This permits the type result of a subscript of BLOCK structure to be known at compiler time.

Another use might be to change the external appearance of the language. For example,

```
DECLARE BOOLEAN: MARY;

SUBSTITUTE (FAIRYTALE,MARY) (HAD,=) (A,NOT$) (LITTLE,FALSE) (LAMB,;)
           (TELLA,WRITE) (PLEASE,;);

MARY HAD A LITTLE LAMB

TELLA FAIRYTALE PLEASE
```

would produce a "TRUE" on the output medium.

Note also that SUBSTITUTE atoms may be chained since the substitution is done before parsing; e.g.,

```
SUBSTITUTE (A,12) (B,-A);
```

is allowable and assigns the value -12 to parameter B. Note that '-A' is actually two atoms, but is interpreted correctly as a single integer constant.

2.4.1.6 EQU

The EQU statement provides a means of declaring the equivalance of two "addresses" at run-time. (It should not be confused with the SUBSTITUTE statement which provides compile-time equivalence of two atoms.) As such it is especially useful to programmers who are familiar with the Assembler for the 1800 and the material in Chapter Four which describes the code produced by the compiler. Others should use it with care. EQU may be used to achieve some of the same effects as the FORTRAN EQUIVALENCE and COMMON statements.

The form of the statement is

EQU {(λ{,}$_{o}^{1}$ <rhs>)} ;

It consists of ordered pairs where the left hand side must be a variable and <rhs> must be an atom acceptable to the Assembler as the right hand side of an "EQU" pseudo-op. To be more specific, <rhs> must be either a λ-atom, an integer constant, or a TEXT constant containing an expression with +,-, and as the only operators, and all λ-atoms in the expression consisting of only five characters or less. The result of an expression must satisfy the Assembler's relocation requirements. The pairs are inserted into the Assembler source program produced by the compiler in the order in which their left hand sides were first detected by the compiler. (This latter condition means that the Assembler EQU statements produced are not necessarily in the same order as the compiler EQU statements encountered. In the case of chained EQU statements the order may be important so that it might be wise to clump them all at the beginning of the program, even before the declarations.) All EQU pairings are put out at the very end of the assembly program so that all other symbols are already defined.

After all that, some examples are in order, especially for those readers

not familiar with assembly languages.

EQU may be used to achieve a FORTRAN-like COMMON by allowing the programmer to assign absolute locations for variables as in the following. (COMMON is assigned from the top of memory--highest address--down.) If there are three INTEGER variables, I,J,K, to be shared by many CESSL programs, the following statement should be included in each program; it will place the variables all in "COMMON". (The top of memory is 7FFF, not FFFF.)

```
EQU (I,?7FFF) (J,?7FFE) (K,?7FFD);
```

Then, when a program refers to any of these variables, it will refer to a location near the top of core, not to a location within itself. REAL variables may be assigned similarly, but must be even-aligned; that is, the last digit of the address should be even. Variables with structured data types may also be assigned in this manner, but those with REAL fields in them must be even-aligned.

A CESSL program may even share COMMON with TSX FORTRAN by assigning the same addresses as FORTRAN does for its COMMON (which can be obtained from a FORTRAN symbol table dump). Care should be taken in mixing with FORTRAN arrays since they run backwards in memory.

To ensure that there is no overlap between a loaded program and COMMON, the programmer should use the "*COMMON N" control card (Section 3.1) to inform the loader of the length of COMMON which he is assigning from this program.

An EQUIVALENCE statement may be approximated by using the appropriate EQU pairings. For example, a user who wishes to refer to a variable as being of type TEXT at one time and type INTEGER at another may do one of two things. He may assign it to have one of the types and then, at every reference in which it is to have the other type, use the "@" compile-time operator to perform a type override. Alternately, he can declare two different variables, each

of the appropriate type, equivalence the two, and use the name with the correct type in the appropriate place. For example:

```
DECLARE TEXT: SCHIZO;
DECLARE INTEGER: PSYCHO;
EQU (SCHIZO, PSYCHO);
```

Thereafter, both names refer to the same location in the computer's memory. As another example, it is sometimes necessary to refer to the same locations alternately in INTEGER and REAL mode. This may be performed as follows:

```
DEFINE INT2 ARRAY INTEGER SIZE 2: I2;
DECLARE REAL: R;
EQU (I2,R);
```

Reference to I2(1) then refers to the first half of the REAL variable and I2(2) refers to the second half.

Equivalences between sections of structured variables may be performed by using a TEXT constant as the right hand side of an EQU pair. For example:

```
DEFINE INT10 ARRAY INTEGER SIZE 10:  I10;
DEFINE INT5 ARRAY INTEGER SIZE  5:  I5;
EQU (I5, "I10+5");
```

References to I5 will then address the last half of the I10 array. Note that the left hand side must always be a simple variable. The right hand side is delivered verbatim to the Assembler, stripped of the quotes, so that all symbols in it must be legal Assembler symbols, containing five or fewer characters. This restriction is not necessary when the right hand side is a λ-symbol since CESSL can do its normal name translation for such symbols.

A use of EQU which bears mentioning is the following:

```
DEFINE CORE ARRAY INTEGER SIZE 16383:  C;
```

```
EQU (C,1);
```

Thereafter, reference to C(I) addresses location I in the computer's memory (except for I equal zero). This may be especially useful for system programmers and other fanatics.

EQU can help in changing the definition of a data type at run-time. See Section 5.2.

2.4.1.7 INCLUDE

The form of this statement is:

```
INCLUDE <integer constant>;
```

For example:

```
INCLUDE 143;
```

This statement tells the compiler that the contents of the named logical file are to be inserted at this point in the statement source stream. This might be useful for oft-used definitions of data types and operators. Lines from a file should not be longer than 81 characters including carriage return. INCLUDE may appear in a file in which case the new file is opened and read. (The INCLUDE statement in a file is treated as an end-of-file marker for the file in which it appears.) If a real end-of-file is read, the reading of source statements reverts to the card reader.

2.4.1.8 ENDPROG

The last statement of every program (main or subroutine) must be

```
ENDPROG;
```

If control "flows into" this statement at execution time (i.e., there is no branch before it) an error comment will be generated (see Section 3.2.3).

<declare part>↠DEFINE λ [ARRAY <type> SIZE <integer constant> | BLOCK < <type-list> >] [; | :<type-list>;]

DECLARE λ NAME;

DECLARE <type>: <λ-list>;

DECLARE LIBF: <λ-list>;

DECLARE FUNCTION: <λ-list>;

SUBSTITUTE {(λ {,}$_0^1$ [<integer constant> | <any atom>])};

EQU $\left\{(\lambda\{,\}_0^1 \text{<rhs>})\right\}$;

NORMALMODE <type>;

INCLUDE <integer>;

DATA λ<value>;

INTERNALFUNCTION λ!;

INTERNALFUNCTION λ!(<λ-list>);

ENDFUNCTION;

ENTRY λ!;

ENTRY λ!(<λ-list>);

ENDPROG;

Figure 2.2. Procedural Declarations of the Language

```
<executable> → IF <exp>;
             → ORIF <exp>;
             → ELSE;
             → ENDIF;
             → ENDIF <any characters except ;>;
             → LOOP λ=<exp>; <exp>; <exp>;
             → ENDLOOP;
             → ENDLOOP <any characters except ;>;
             → <assignment>
             → CONTINUE;
             → EXECUTE <exp>;
             → GOTO <exp>;
             → RETURN;
             → RETURN <exp>;
             → FUNCTIONRETURN;
             → FUNCTIONRETURN <exp>;
             → READ <left-designator-list>;
             → WRITE <exp-list>;
             → WRITEFMT <left designator> ;<exp-list>;
             → WRITEFMT <left-designator>;;
             → INPUTDEV <device>;
             → OUTPUTDEV <device>;
             → PAUSE <integer-expression>;
```

Figure 2.3. Executable Statements of the Language

```
<assignment>   → <left des>  = <exp>  ;
               → <left des> @ "<type>"  = <exp>;
 <left des>    → λ
               → λ (<exp list>)
  <exp list>   → <exp>
               → <exp list>, <exp>
   <exp>       → <exp> θ <exp>
               → φ<des>
               → <des>
   <des>       → <left des>
               → (<exp>)
               → λ!(<exp list>)
               → λ!
               → (<assignment>)
```

Figure 2.4 Syntax of Assignment Statement

Name	Operand Type (X)	Result Type	Function
-	I	I	Unary minus (X)
NEG$	R	R	
ABS$	I	I	Absolute value (X)
	R	R	
NOT$	B	B	Logical complement(X)
BITNOT$	I	I	Bitwise complement(X)
FIX$	R	I	Real-to-integer conversion (X)
FLOAT$	I	R	Integer-to-real conversion (X)
SQRT	I	I	Square root (X)
	R	R	
ATAN	R	R	Arctangent (X)
SIN	R	R	Sine (X)
COS	R	R	Cosine (X)
EXP	R	R	e^X
ALOG	R	R	Natural Log (X)
TANH	R	R	Hyperbolic tangent(X)
LINK	Any	None	TSX LINK call

I = Integer
R = Real
B = Boolean

Figure 2.5 Monadic Operators

Name	Operand Types Left op	Right op	Result Type	Function
+-*/	I	I	I	X+Y, X-Y
P	I	R	R	X*Y, X/Y, X^Y
	R	I	R	
	R	R	R	
MOD	I	I	I	X modulo Y
=	R	I	R	X=Y (with conversion
	I	R	I	for real and integer
	T	I	T	X=Y (no conversion)
	I	T	I	
	TA	TC	TA	X=Y for any two
	A	A	A	similar types
EQ, NE	I	R	B	X=Y, X≠Y
GT, GE	R	I	B	X>Y, X≥Y
LT, LE	T	I	B	X<Y, X≤Y
	I	T	B	
	TA	TC	B	Comparison of TEXTARRAY
	TC	TA	B	and TEXT constant
	A	A	B	Comparison of any similar types
AND, OR XOR	B	B	B	X "and" Y, X "or" Y X "exclusive or" Y
$BITAND$	I	I	I	Bitwise logical
$BITOR$	T	I	T	operations
$BITXOR$	I	T	I	
LS	I	I	I	X"Left Shift Logical" Y
RS	T	I	T	X "Right Shift Logical" Y
@	A	TC		Type override

I = Integer
R = Real
B = Boolean
T = Text
TA = TEXTARRAY
TC = TEXT constant
A = Any type

Figure 2.6

Dyadic Operators

( )	80
RS LS $BITAND$ $BITOR$ $BITXOR$	70
P	60
* / MOD	50
+ -	40
EQ NE GT GE LT LE	30
AND	20
OR XOR	10

Figure 2.7 Precedence Values for the Dyadic Operators

Punctuation	Operators		
	Dyadic		Monadic
;	OR	+	ABS$
=	XOR	-	NEG$
(	AND	*	NOT$
)	EQ	/	BITNOT$
,	NE	MOD	FIX$
	GT	P	FLOAT$
!	GE	RS	SQRT
:	LT	LS	ATAN
<	LE	$BITAND$	ALOG
>	@	$BITOR$	SIN
?		$BITXOR$	COS
			EXP
			TANH
			LINK

Keywords and λ-atoms

ARRAY	ENDPROG	INPUTDEV	PNCHC*
A7E*	ENTRY	INTEGER	READ
BLOCK	EQU	INTERNALFUNCTION	REAL
BOOLEAN	EXECUTE	KBDE*	RETURN
CARDE*	FALSE	LABEL	SIZE
CARDS	FILE	LIBF	SUBSTITUTE
CONTINUE	FGETA*	LOOP	TEXT
DECLARE	FPUTA*	NAME	TRUE
DEFINE	FUNCTION	NORMALMODE	TYCH*
ELSE	FUNCTIONRETURN	ORIF	TYPEWRITER
ENDFUNCTION	GOTO	OUTPUTDEV	T7E*
ENDIF	IF	PAUSE	WRITE
ENDLOOP	INCLUDE	PDP7	WRITEFMT
			$$$

*Name of a system subroutine used for I/O. May be called firectly.

Figure 2.8 Reserved Atoms - Procedural

2.4.2 Executable Statements

The basic executable statements of the language are: assignment, unconditional branch, conditionals, iteration, input/output, subprogram return, PAUSE, CONTINUE, and EXECUTE. The subprogram return statements are discussed in Sections 2.4.3 and 2.4.4.

2.4.2.1 Assignment Statement

Syntax

The basic statement of the language is the assignment statement. The most succinct way to present its acceptable forms is via the productions of a grammar. This description is found in Figure 2.4. The following observations are made about this syntax.

1) The symbols λ, θ, and ϕ are not particular terminal symbols but designators for the class of λ-atoms, dyadic (two operand) operators and monadic (one operand) operators respectively. The lexical parsing actually performs the necessary assignment of an atom to these classes, if appropriate, prior to parsing. (The terms monadic and dyadic are used instead of unary and binary to avoid possible confusion with the concept of "bitwise" operators.) The monadic and dyadic operators are listed in Figures 2.5 and 2.6 respectively.

2) In this descriptive grammar the production

<exp> → <exp> θ <exp>

obviously introduces an ambiguity into the resulting language. In contrast to typical applications, the grammar rules are not used here to establish the relative precedence of a multitude of binary operators. In contexts where an ambiguity exists in parsing an input string, as for example in

A + B * C

the order of association is resolved by an extra-grammatical attribute of all dyadic operators, its (single) precedence value. (The resulting organization has the virtue of making the grammar invariant with respect to the number and relative precedence of dyadic operators, and accordingly new dyadic oeprators can be introduced with absolutely no impact on the syntactic parser.) The precedences of the dyadic operators are given in Figure 2.7. Operations with identical precedences are performed from left to right.

3) Note that the abuse of "-" as both a monadic and dyadic operator has led to exception condition parsing in other languages in which $-X^{Y}$ (-X P Y) is parsed as $-(X^{Y})$. In this language monadic operators are <u>always</u> invoked first so that the parse is $(-X)^{Y}$. The atom NEG$ may also be used to specify "unary minus".

4) This syntax corresponds roughly to normal FORTRAN syntax except for point three above and the last rule which allows embedded assignment statements, i.e., statements of the form:

```
Q = (X = (Y + 3 * (Z=1)));
```

Note carefully that the syntax requires the embedded assignment to be enclosed in parentheses and that the parentheses are part of the assignment. Thus, the statement

```
A(I=4) = 1;
```

is syntactically incorrect, while

```
A((I=4)) =1;
```

is correct.

<u>Semantics</u>

Several points of interpretation need to be made regarding expression evaluation and value assignment.

1) The operators (both monadic and dyadic) have well-defined meanings only when they are applied to operands which are of the proper data types. Figures 2.5 and 2.6 contain the mode-combinations for which the operators are defined, and the mode of the result of applying the operators. For example, the operator "+" is defined for mode-combinations INTEGER/INTEGER, INTEGER/REAL, REAL/INTEGER, and REAL/REAL producing an INTEGER result in the first case and REAL results in the remaining cases. It is not defined for any other data types.

2) For the purpose of this table "=" (assignment) is considered as an operator taking only a specific combination of operands. For the case in which the modes on either side of the "=" are identical (primitive or user-defined), the operation consists merely in a movement of data to the address given by the left hand side. For the combinations INTEGER/TEXT and TEXT/INTEGER, again there is merely data movement. The cases of REAL/INTEGER and INTEGER/REAL cause a conversion from one data type into the other before the movement is performed. For the TEXTARRAY/TEXT constant case, as many words of the constant are moved as can fit into the TEXTARRAY. If the TEXTARRAY is larger than the constant, the entire constant is moved and the remainder of the TEXTARRAY is left untouched.

3) The embedded assignment statement (see point four under syntax) should be used carefully. The embedding is legal only for assignments to variables of primitive types. The "value" of such an embedded statement is the value of the variable <u>at the time of the assignment</u>, <u>not the value of the variable</u>. For example, if I is an integer, the statements:

```
I = 1;
I = (I = I + 2) * (I = 5);
```

would yield a final value for I of 15, not 25. This may be best seen by writing the embedded assignments as separate statements and assigning temporary locations to hold the value of the variable for future use:

```
I = 1;
I = I + 2;
TEMP1 = I;
I = 5;
TEMP2 = I;
I = TEMP1 * TEMP2;
```

In this context it is important to note that the address for the left hand side variable is calculated (if subscripted) before the right hand side expression is calculated. For example, the statements:

```
I = 2;
A(I) = (I=5);
```

result in A(2) and I containing the value 5.

4) The intrinsic (built-in) functions of CESSL are NEG$, ABS$, FIX$, FLOAT$, SQRT, ATAN, ALOG, SIN, COS, EXP, TANH, and LINK. These are treated as monadic operators rather than as subroutine calls in order to achieve object code efficiency with respect to the TSX operating system and to maintain (some small degree of) compatibility with other languages. If the user desires to use the functions SQRT, ATAN, ALOG, SIN, COS, EXP, or TANH as external names (see point 7, below), the function name should be preceded by an "F", e.g., FSQRT. (Consult the TSX Subroutine Library (7) or Figure 4.3 for more details.) Since these functions are treated as monadic operators the user has the option of using them in expressions as follows:

```
A = SQRT B;
```

Of course, he may also parenthesize the argument to match normal usage:

```
A = SQRT(B);
```

and he must do so if the function is to be applied to an expression involving

dyadic operators. For example:

```
A = SQRT -B;   is the same as   A = SQRT(-B);
A = SQRT B+C;  is the same as   A = SQRT(B)+C;
                          not   A = SQRT(B+C);
```

Those operators which have operands or results which are angles (SIN, COS, TANH, and ATAN) expect the angles to be expressed in terms of radians.

The LINK operator takes as an operand the name of a nonprocess coreload in the TSX fixed area of the disk. The effect is to read in and start execution of that core load. By TSX convention, a LINK call can only have effect when included in a nonprocess coreload executed from the fixed area.

5) All arithmetic between operands of types INTEGER and REAL is performed by converting the INTEGER operand to REAL mode and performing the operation as if the mode-combination were REAL/REAL.

6) The relational operators, EQ, NE, GT, GE, LT, and LE, are defined between operands of identical data types and produce a BOOLEAN result. They are also defined for the combinations INTEGER/REAL and REAL/INTEGER, with the appropriate conversion being performed. Comparison for the combinations are made as if both types were INTEGER, and are one word comparisons. For example, the following two expressions produce the same result:

```
I  $GT$  "ABCDE"
I  $GT$  "AB"
```

Comparison of REAL types should be made with great care. Due to the structure of the floating point arithmetic in the IBM TSX system, identical REAL numbers may not produce the correct results using any of the operators. In addition, REAL comparisons are performed internally by a subtraction of the operands and comparison of the result to zero. Two numbers which differ by less than the precision of their floating point representation may produce incorrect

results. Also, two numbers which differ by less than the minimum value of the floating point range may produce incorrect comparisons. For example, the values of the following two expressions are undefined.

```
1.0 $LT$ 1.0 + 1.0E-8
1.01E-36 $LT$ 1.015E-36
```

Comparisons of identical non-primitve data types are performed as if the types were multiple-precision integers, each word having a sign associated. That is, the relation is first tested between the first word of each operand. If this establishes the result, no further testing is done. If it does not, the second words are tested, and so on. For example, given two three word arrays of integers

```
A(1) = 10  B(1) = 10
A(2) =  4    (2) =  2
A(3) =  7  B(3) =  8
```

the following table gives the result of all six relations, each of which requires two comparisons to establish.

```
A $EQ$ B  is FALSE
A $NE$ B  is TRUE
A $GT$ B  is TRUE
A $GE$ B  is TRUE
A $LT$ B  is FALSE
A $LE$ B  is FALSE
```

Since the relations are calculated as if the components were integers, comparisons should not be made between variables containing REAL fields. LABEL fields should be treated with care. TRUE is greater than FALSE in such comparisons.

Comparison of TEXTARRAYs and TEXT constants is performed as a multiple-word compare where the number of words compared is the minimum of the lengths of the two comparands. For example:

```
DEFINE T4 ARRAY TEXT SIZE 4: AXOLOTL;
```

```
IF AXOLOTL $EQ$ "HAROLD" $OR$
   AXOLOTL $EQ$ "MAXIMILIAN";
```

The first comparison is on three words and the second is on four. The user should remember that odd word counts in TEXT constants yields a final word with the last character in the high order (left hand) position, and a binary zero in the low order position.

7) An explicit "operator", the exclamation mark, is used to designate function (subroutine) calls. This convention is similar to that of MAD (9); however a different atom has been chosen to avoid another usage for the period (which is used in MAD). Note that subroutine names may be used without the exclamation mark (deferring the call) as parameters for subroutine calls.

This is achieved by declaring the λ-atom to have attribute FUNCTION. (Implementation restriction: due to the structure of the TSX operating system, subroutines which are built into the SKELETON and subroutines entered via LIBF statements may not be assigned attribute FUNCTION.)

8) A special compile-time dyadic operator, atom "@", is used for the following purpose. The operator accepts as a left operand an argument of any type and as right operand only a constant argument of type TEXT. Where BLOCK data structures are used with variable subscripts the <type> of an expression is not determinable. For example:

```
DEFINE ABLE BLOCK <REAL, INTEGER, INTEGER>: MARY;
X = MARY(I);
```

MARY(I) will be of type INTEGER if I has value 2 or 3, and type REAL if I has value 1. However, the compiler does not support dynamic data types at run time, and the type of every (sub-) expression must be known at compiler time. Thus, the above assignment could not be compiled as it stands. By

using the @ operator followed by a quoted <type> atom, such situations may be resolved. Thus,

```
X = MARY(I) @ "INTEGER";
```

would compile and treat MARY(I) as an integer regardless of the value of I. The operator may also be similarly used to override a known <type> in favor of a different one.

The reader should also be aware of the SUBSTITUTE statement (Section 2.4.1.5) as a means of providing constant, mnemonic subscripts.

The following are additional examples of legal assignment statements:

```
DEFINE INT3 ARRAY INTEGER SIZE 3:I3,J3;
DEFINE QQSV BLOCK <REAL, INT3, BOOLEAN>:Q1,Q2;
DECLARE REAL: R,S;
DECLARE BOOLEAN A,B,C;
DECLARE INTEGER I,J;
DECLARE TEXT T1,T2;
DECLARE LABEL: L1,L2;

R = SQRT R + 1;
R = SQRT(R) + 1;
Q1(1) = R;
Q1(1) = Q1(2,1) $P$ I;
A = B $OR$ Q1(2,2) $GT$ 4 $XOR$ L1 $EQ$ L2;
XYZ: B = Q1 $EQ$ Q2 $AND$ Q1(3);
L1 = XYZ;
I = BITNOT$ J $BITOR$ ("A" $RS$ 8);
T1 = "AB";
Q1 = Q2;
```

2.4.2.2 Unconditional Branch

The form of the unconditional transfer is

```
GOTO <exp>;
```

where <exp> is of type LABEL and may be either a constant or a variable. The effect of the GOTO is that control passes to the statement with the designated label. For example:

```
     DECALRE LABEL:LVAR;
     .
     .
     .
XYZ: CONTINUE;
     .
     .
     .
     GOTO XYZ;      (1)
     LVAR = XYZ;    (2)
     .
     .
     .
     GOTO LVAR;     (3)
```

If the value of LVAR is not changed between statements (2) and (3), statements (1) and (3) have the same effect.

The <exp> may be a formal parameter to a subroutine or internal function. The result of the GOTO then is to transfer control to the statement designated by the calling program as the corresponding actual parameter.

2.4.2.3 Conditionals

The conditional statements are of the following forms:

```
I F<exp> ;
ORIF <exp>;
ELSE;
ENDIF;
```

These are interpreted the same as the compound conditional in the 7090 MAD language (9). In particular, note that exactly one "ENDIF;" must follow the "IF..." to determine the scope of the sequence. The "ORIF..." may be used any number of times between 0 and 14, inclusive, and the "ELSE;" at most once. If "ELSE;" appears there may be at most 13 "ORIF..." statements. These statements delimit mutually exclusive sequences of statements of which the first true condition will enable its corresponding body to be executed. If no previous IF or ORIF statement yielded a true condition, the ELSE body (if any) will be executed.

If IF, ORIF, ELSE, and ENDIF represent their respective statements and α any valid sequence of statements, then the following defines legal uses of the conditional branch:

$$\text{<legal IF>} \rightarrow \text{IF } \alpha \ \{\text{ORIF } \alpha\}_0^{14} \ \{\text{ELSE } \alpha\}_0^{1} \ \text{ENDIF}$$

For example:

```
DECLARE INTEGER: A,B,D,E;
DECLARE BOOLEAN: C;
IF A $GT$ B $OR$ C;
  IF D $EQ$ 0;
    GOTO XYZ;
  ENDIF;
ORIF A $LT$ B;
  IF D $NE$ 0;
    GOTO ZYX;
  ORIF D $GT$ 5 ;
    GOTO HOME;
  ENDIF;
ELSE;
  A = B;
ENDIF;
```

As a convenience for "labelling" ENDIFs, all characters between the keyword ENDIF and the semi-colon will be ignored. Thus it is possible to write:

```
ENDIF ON X CONDITION;
```

2.4.1.4 Loop Statement

The loop statement is of the form:

```
LABEL1:  LOOP λ = <exp>; <exp>; <exp>;
...
LABEL2:  ENDLOOP;
LABEL3:  ...
```

The interpretation of this statement is similar to that of 7090 MAD. The left side of the assignment must be a λ-atom (i.e., cannot be subscripted) of type INTEGER or REAL, and specifies the controlled variable for the loop. The assignment is performed and then the last expression is evaluated. If TRUE, the loop is terminated by a transfer to the first statement after the ENDLOOP statement; otherwise, the loop body is executed. It is possible that the loop body may not be executed at all. The ENDLOOP statement returns control to the loop header statement where the controlled variable is incremented by the first expression and the termination test is performed again. The body of a LOOP may be null. For example:

```
I = 1;
LOOP I = DATA(I); DATA(I);I $EQ$ 0;
   DATA(I) = I-DATA(I)-1;
ENDLOOP;
```

is exactly equivalent to

```
    I = 1;
    I = DATA(I);
    GOTO T1;
T3: I = I+DATA(I);
T1: IF I $EQ$ 0; GOTO T2; ENDIF;
    DATA(I) = I-DATA(I)-1;
    GOTO T3;
T2: CONTINUE;
```

Note that the equivalence is exact implies that the loop control variable and increment may be changed in the middle of the loop, the loop control variable retains its value when the loop is satisfied, and control may be transferred into the body of a LOOP.

As a convenience for "labelling" ENDLOOPs, all characters between the keyword ENDLOOP and the semi-colon will be ignored. Thus it is possible to write something like:

```
ENDLOOP I;
ENDLOOP ON NEIGHBORS;
```

2.4.2.5 Input/Output

Input/output facilities are oriented to the special needs at LOCG: a hands-on, real-time environment with I/O being a secondary consideration. Input is free format, items being separated on the input medium by commas and end-of-record indicators (end of card, carriage return, etc.). Output is either simple (fixed format), or formatted as described below. There are five I/O statements: OUTPUTDEV, INPUTDEV, READ, WRITE, and WRITEFMT.

DEVICES

Input or output is performed on a particular device, which must be designated previous to any I/O call. For example, input may be from CARDS, or from TYPEWRITER, or .... These devices supply a stream of EBCDIC characters to the input routines which interpret them. Output is a stream of EBCDIC characters directed to the device.

The input or output device which is to be addressed is specified by the executable statements:

```
INPUTDEV  <device>;
OUTPUTDEV <device>;
```

The specification of which device is to be addressed by the I/O routines holds until it is changed by another device control statement. Thus, it is possible to read or write numbers from or to different devices by changing the device addressed between the READ or WRITE statements.

If the <device> given above is one of the keywords CARDS, TYPEWRITER, PDP7, or FILE, the I/O routines are automatically connected to the system version of the routines to read or write from these devices. If <device> is none of these,the I/O routines will be connected to the subroutine with the given name. Consult Section 4.5.1 for additional information aoout the required structure of such a routine. Idiosyncrasies of the system routines are listed below.

Input from TYPEWRITER follows the usual TSX conventions, only briefly mentioned here. When input is needed by the program, the carriage is returned, the PROCEED light on the typewriter console is turned on, and the keyboard is unlocked. Editing of an unfinished line is provided by the two keys ER CHR and ER FLD, which delete the last character typed and the whole line, respectively. A line is terminated by the EOF key. Output to TYPEWRITER is straightforward except for the last character of a line. If the last character sent to the typewriter is not a carriage return, it may wait (i.e. be buffered) until another character is sent before being typed out.

Input from CARDS requires only that the card reader be ready. Output to CARDS is buffered until a carriage return character is encountered or until 80 columns of information is accumulated, at which time the card is punched and a new card image started. The card routines do not check the cards to be punched to ensure that they are blank so that it is possible to overpunch.

I/O referring to the PDP7 passes through the "copy port", a character oriented path between the 1800 and the PDP7. Of course, there must be cooperating program running in the PDP7 to supply or dispose of characters. Characters at the 1800 end of the copy port must be EBCDIC so that PDP7 programs must take care to use a "translated" copy port read or write if they wish to deal with ASCII characters.

FILE refers to the LOCG logical file system. A file must be opened for input prior to any READs and opened for output prior to any WRITEs. Files should be closed after use. Consult the approriate LOCG internal memos and subroutine descriptions for additional information on use of the file system from higher level languages.

The system routines used for I/O are KBDE, TYCH, CARDE, PNCHC, A7E,

T7E, FGETA, and FPUTA. These names are automatically defined in all programs to have attribute FUNCTION.

INPUT

The form of the input statement is:

```
READ <left-designator-list>;
```

where a <left-designator-list> is a list of λ-atoms and/or subscripted λ-atoms, separated by commas. For example

```
READ ABC,I,J,QQSV(1), QQSV(2,1,J);
READ I;
```

All subscripted variables in the list have their addresses calculated before any data transfer takes place with the result that variables read in are not used until the end of the statement. For example,

```
I = 3;
READ I, A(I);
```

Values will be read into I and A(3) no matter what the value of I read in. Except for the case of TEXTARRAY (see below), every item in the left-designator-list must be a variable of type INTEGER, REAL, BOOLEAN, or TEXT. Since every item has a known data type, the input routines look for the next item in the input stream and convert it to internal form according to the known type.

Items on the input medium are separated by commas (end-of-line indicators are treated as commas for item separation purposes). When the input routines are reading numbers, all spaces are ignored, and all characters are treated as numbers. Thus stray letters will foul up the reading.

Input for BOOLEAN type should be either an integer 0 or integer 1 for FALSE or TRUE, respectively.

REAL numbers are read in any variation of "E-notation". The following items would be read as the same real number:

1234.0, 1234., 1234, 123400E-02, 12340.E-1, +12.34E+02,....

TEXT input may appear in the input stream in either of two ways: enclosed in quotes (") or not. If the first non-blank character in the input is a quote, the item consists of all characters up to a terminating quote. (A quote may appear as part of the string by doubling it as in a text constant (Section 2.1).) If the first non-blank character in the input is not a quote, then the input is terminated by the first comma or end of the line encountered, with leading and trailing blanks deleted from the string, but with internal blanks maintained.

If the item to be read is of type TEXT, a maximum of two characters will be read. If more characters are given, only the left two are read, and the remainder are ignored. If fewer than two characters are given, characters are left-adjusted with trailing zeros (not blanks). For example, if T is of type TEXT,

```
READ T;
```

will produce the following results for the given input strings (C1, C2 and 40 are the hexadecimal equivalents of the EBCDIC characters A,B, and blank, respectively and b specifies a blank):

```
"AB",   T=C1C2          bA,     T=C100
"A ",   T=C140          bAbb,   T=C100
" A",   T=40C1          A,      T=C100
" ",    T=4040          ,       T=0000
"A" ,   T=C100
"" ,    T=0000
```

The only exception to the rule that the READ-list item must be primitive is if the item in the READ-list is in the class TEXTARRAY (Section 2.3). For example, the following sequence is legal:

```
DEFINE TEXT10 ARRAY TEXT SIZE 10: STRING;
DEFINE T3BY10 ARRAY TEXT10 SIZE 3: STRARRAY;
```

```
READ STRING, STARRAY(2);
```

The input routine will read up to 2*N characters (in this case, up to 20 characters), pack them two per word, and store them in the array starting from the left (i.e., smallest subscript). Surplus characters will be ignored. If fewer characters appear on input than there is room for, the remaining character positions are filled with zeros (not EBCDIC blank). For example:

```
      "AB AB AB AB"          bbABCbbbDEFbABCb,
STRING(1)=C1C2               C1C2
      (2)=40C1               C340
      (3)=C240               4040
      (4)=C1C2               C4C5
      (5)=40C1               C640
      (6)=C200               C1C2
      (7)=0000               C300
      (8)=0000               0000
      (9)=0000               0000
     (10)=0000               0000
```

OUTPUT

There are two kinds of output statements: simple and formatted. They may both appear in a single program for convenience.

The simple output statement is of the form:

```
WRITE  <exp-list>;
```

For example:

```
WRITE I,J,A(I),B(I,J*3/4)+15*EXP(4),"RESULT=",D,15;
```

All the expressions in the expression list must designate a quantity which is one of the following types: INTEGER, REAL, BOOLEAN, TEXT, or TEXTARRAY.

Note that all expressions in a WRITE-list are evaluated before any output is performed. Thus the statements:

```
I = 3;
WRITE I, (I=4);
```

will produce two four's on the output medium, not a three and a four.

INTEGER types are put out in an I6 format (i.e., in a field of width six, right adjusted, with leading blanks). REAL numbers are put out in the form ±x.xxxxxx±yyb where b indicates a blank. BOOLEAN items are put out as either of the two character strings: TRUE, or FALS. (The "E" is left off "FALS" so that a constant field width for BOOLEAN output can be maintained.)

TEXT constants are put out exactly as stated, without the outer quotes. TEXT variables are put out in EBCDIC;binary zeros are ignored (not put out). For example, if T is of type TEXT, the statements

```
T="AB";   Q="Z";
WRITE "T IS:", T, " .", Q;
```

would produce the output line:

```
'T IS:AB .Z'
```

where the primes ('s) are used here merely to delimit the extent of the output.

Note that the output produced has no imposed record limits--only what is sent by the WRITE statement appears on the output medium. To effect a line-oriented output, the user must specifically send an end-of-record (e.g., carriage return) character to the output device. This can be done by executing system subsoutine CRET, which sends a carriage return character to the current output device. It can also be accomplished by the following subterfuge:

```
DECLARE TEXT:  CR;
DATA CR ?15;
.
.
WRITE CR;
```

since 15_{16} is the EBCDIC character for carriage return. The other special EBCDIC characters (e.g., line feed, shift-to-red) can be used in the same way. Also note that the high order character of CR is a binary zero; the character

This program:

```
WRITE "ARRAY OF COST FACTORS:", CR, CR, "  ";
LOOP I=1;1;I $GT$ 20;
     WRITE COST(I), "¢  ";
     IF I $MOD$ 5 $EQ$ 0;
          WRITE CR, "  ";
     ENDIF;
ENDLOOP;
```

Might produce this output:

```
ARRAY OF COST FACTORS;
     10¢       15¢       17¢       14¢       20¢
      1¢       12¢        8¢      100¢      -12¢
      9¢       34¢       56¢       55¢        0¢
    156¢     -212¢       34¢       17¢        0¢
```

Figure 2.9 WRITE Example

routines ignore such (non-)characters.

The one exception to the rule that all expressions in a WRITE list must be primitive is for the case when an expression is a TEXTARRAY (Section 2.3), as for the input case above. Given the definitions and declarations for that example, the following statement would output twenty EBCDIC characters from each of the two specified arrays:

```
WRITE STRING, STRARRAY(3);
```

Given the above definitions for CR, and array COST of integers size 20, and an integer I, the program segment in Figure 2.9 might produce the output given.

FORMATTED OUTPUT

The WRITEFMT statement affords the user some of the power of formatted output which is usually an important feature of other programming languages. This implementation of CESSL was not designed for large scale I/O programs so that appropriately less effort has been expended in formatting development. What is provided is a run-time package which interprets a "format-list" which is compiled or put together *by the user*. Although this lacks the convenience of FORTRAN in the definition of the format list, it gains the considerable advantage of allowing run-time construction and alteration of format specifications (cf. MAD/7090 and PL/I format variables).

The formatted output statement has the form:

```
WRITEFMT  <left-des>; <exp-list>;
```

The <left-des> should specify an INTEGER array which will be interpreted as the format list. It may be simple or subscripted, as in the following examples:

```
DEFINE INT10 ARRAY INTEGER SIZE 10:  LIST1;
DEFINE INT2BY10 ARRAY INT10 SIZE 2:  LIST2;
```

```
WRITEFMT LIST1; A,B,C;
WRITEFMT LIST2(2); A,B,C;
WRITEFMT LIST1(5); A,B,C;
```

In the last statement, LIST1(5) actually designates an INTEGER but since arrays are stored contiguously, it also points to the last five words of LIST1, so that the usage is legal and, perhaps, even meaningful.

Ordinarily, the contents of a format list are initialized via a DATA statement; for special uses they may be changed by the user at run-time.

The form and restrictions of the <exp-list> are exactly the same as for the WRITE statement. BOOLEAN, TEXT, and TEXTARRAY variables are sent to the output device in exactly the same manner. INTEGER and REAL variables are put out according to an interpretation of the format list.

Items in the <exp-list> which are of type INTEGER require a positive number, w, in the format list. W specifies the width of the field in which the output is to appear, right justified with leading blanks (this is equivalent to a FORTRAN Iw specification). Care should be taken to provide a field width wide enough to accomodate the maximum value expected for the integer expression, including a possible minus sign for negative quantities. If too narrow a field is specified, the value will be put out, but in a field of the minimum sufficient width (thereby messing up the programmer's output spacing, but at least retaining the value). For example:

```
DEFINE INT10 ARRAY INTEGER SIZE 10: LIST;
DATA LIST 5, 0;
DECLARE INTEGER: I;
I = 43;
WRITEFMT LIST; I;
```

produces

```
bbb43
```

where b indicates a blank in the output.

The output for items in the <exp-list> which are of type REAL can take two forms. The first is the same as for the WRITE statement, "±x.xxxxxx±yyb". This output is specified in the format list by the appearance of the positive number "999", as in:

```
DATA LIST 999,0;
WRITEFMT LIST; R;
```

where R has been declared REAL.

The second form of REAL output produces something like "xxx.yyyy". This form is specified by two successive positive integers in the format list, respectively the total field width (including sign and decimal point) and the number of digits after the decimal point (This is equivalent to FORTRAN Fw.d). The number is always right justified in the field with leading blanks. If the field width is too narrow the number is sent out in E-format (messing up the spacing, but at least getting it out). For example:

```
DATA LIST 8,4,0;
WRITEFMT LIST; 3.45E1;
```

produces

```
b34.5000
```

Other entries in a format list can be used to control spacing and output EBCDIC characters. A negative word in the range -1 to -299 is treated as a request for as many spaces in the output as the number is negative. (This is similar to the FORTRAN nX coding for spaces.) For example:

```
DATA LIST -4,4,0;
WRITEFMT LIST; 84;
```

produces

```
bbbbbb84
```

A single EBCDIC character may be put out from the format list by encoding it as a word in the list in the following manner. Take the decimal representation of the EBCDIC character, negate it, and add -300. For example, carriage return is an EBCDIC character with a hexadecimal representation of 15, or a decimal representation of 21. Thus, an integer -321 in the format list will produce a carriage return on the output medium. Appendix A lists the EBCDIC characters which might be of interest, their hexadecimal, decimal, and format-list representations. An easier way of outputting the non-control EBCDIC characters (i.e. the printing graphics) is by a sort of quoting. The special value of -993 in a format list declares that all words up to the occurence of the value -995 contain EBCDIC characters. The characters may be packed one or two per word--zero characters are ignored (i.e. not put out). For example:

```
DATA LIST -993, "COST = ",-995, 5,2,0;
WRITEFMT LIST; R;
```

might produce the output:

```
COST = 43.20
```

The final special value which may appear in a format list is zero. It marks the end of the format specification and resets interpretation to the beginning of the list, if necessary.

In summary, there are three kinds of numbers which can appear in a format list: positive, specifying a data conversion format, negative, specifying the output of EBCDIC characters; and zero, specifying a reset of the interpretation pointer to the beginning of the list.

These items in the format list interact with the WRITEFMT statement in

the following manner. A pointer is first set to the address given by the <left-des>. If an item in the <exp-list> is of type BOOLEAN, TEXT, or TEXTARRAY, it is put out just as in the WRITE statement. INTEGER and REAL items in the <exp-list> require a specification from the format list. If the interpretation pointer does not point to a number which can serve the purpose (i.e., positive), the function specified by the negative number is performed (either spacing, or output of one or many EBCDIC characters). The pointer is then incremented to the next word and the test is performed again. If at any time during the pointer advancement a zero value is encountered, the pointer is reset to the value it initially had. After the last item in the <exp-list> has been put out, the pointer is advanced until it encounters a positive or zero number, thus allowing the output of EBCDIC characters beyond the last format specification. This might be especially useful in the output of a carriage return since record control characters must be specified by the user.

Two implications of the above explanation should be emphasized. First, interpretation of the format list occurs only at two times: when an output field specification is needed and at the end of the <exp-list>. No interpretation of the list is performed before the output of an <exp-list> item of type BOOLEAN, TEXT, or TEXTARRAY. For example:

```
DATA LIST -933. "ABC", -995, 4, 0;
WRITEFMT LIST; "DEF", 15;
```

would produce

```
DEFABCbb15
```

not ABCDEFbb15

Second, format lists should be ended with a zero word to ensure that the interpretation at the end of the WRITEFMT statement will terminate.

There are two special forms of the WRITEFMT statement. The first has a

null <exp-list> as in

```
WRITEFMT <left-des> ;;
```

which may be used to obtain interpretation of a format list when there are no items to be put out. The second uses a zero for the <left-des> in the form:

```
WRITEFMT 0; <exp-list> ;
```

which specifies that the format list it be used is the one given in the last executed WRITEFMT statement, retaining the interpretation pointer's position. The use of the latter form is illustrated in the following program segment which produces output almost identical to that of Figure 2.9.

```
SUBSTITUTE (CENT, -374) (QB, -993) (QE, -995) (ENDFMT,0) (CARET, -321);
DATA LIST QB, "  ", QE, 6, CENT, -3, 6, CENT, -3, 6, CENT, -3, 6,
          CENT, -3, 6, CENT, CARET, ENDFMT;
WRITEFMT LIST; "ARRAY OF COST FACTORS:", CR, CR;
LOOP I=1;1; I $GT$ 20;
      WRITEFMT 0; COST(I);
ENDLOOP;
```

This segment also shows the utility of the SUBSTITUTE statement in providing mnemonics for the special values used in a format list. Once a mnemonic is defined, it can be used in several format lists.

2.4.2.6 Miscellaneous: PAUSE, CONTINUE, EXECUTE

The form of the PAUSE statement is:

```
PAUSE <exp>;
```

where the expression must be of type INTEGER. The effect is that the computer goes into WAIT state (all processing stopped), with the expression displayed in the A register. Pressing the START button continues executing at the next statement. Examples:

```
PAUSE 1;
PAUSE N;
PAUSE X+Z(Y);
```

The form of the CONTINUE statement is, simply:

```
CONTINUE;
```

which serves as a convenient place to introduce a label. Nothing else happens.

The form of the EXECUTE statement is:

```
EXECUTE <exp>;
```

which allows direct subroutine calls. For example:

```
EXECUTE ABC!(1,3*SQRT(R));
EXECUTE XYZ!;
EXECUTE (Z=X+Y);
EXECUTE X+Y;
```

Note that the fourth example adds X and Y, but doesn't do anything with the result. The third is equivalent to the assignment statement

```
Z=X+Y;
```

2.4.3 Subroutines

Subroutines (also called external functions) are independently compiled programs which may be "called" from other programs (perhaps with parameters) to produce a result--either a returned, explicit value, or "side-effects" on the parameters.

2.4.3.1 Subroutine Definition

A program is a subroutine if it contains an ENTRY statement. Thus, no program can contain both an ENTRY statement and a DECLARE...NAME statement. The form of a subroutine is:

```
.
.
.
ENTRY -;
.
.
.
RETURN;
.
.
.
ENDPROG;
```

where there may be more than one ENTRY and more than one RETURN statement.

In CESSL, the "name" of a subroutine is its ENTRY point, i.e., the point at which the subroutine begins executing; it is this name which is "external" for loading purposes. A program which is a subroutine may have many entry points (but not more than fourteen in the TSX system), each with its own formal parameter list. The first such named entry point is the "program name" given to TSX.

The form of the ENTRY statement is:

```
ENTRY λ!( <λ-list>);
```

The statement declares this physical point in the program to be the entry point for the given λ-atom. The λ-atom must obey the TSX naming conventions

since it is to be declared external. The <λ-list> specifies the formal parameters for the call. If there are no formal parameters, no list need be given. For example:

```
ENTRY SUB!(A,B,C);
ENTRY XYZ!;
```

If the formal parameters are to be used in the body of the subroutine, their types must be declared, and, of course, should match the types of the actual parameters used in any calls on the subroutine. The types of parameters may be any primitive type or any type defined by the user. (Formal parameters should not appear as the left half of an EQU-statement pair unless the user is thoroughly familiar with material in Chapter Four.)

The advantage of having several entry points to a subroutine is that code, variables, and actual parameters may be shared even across separate call . For example:

```
     DECLARE REAL: X,Y,Z,FACT;
     DECLARE INTEGER: I;
     ENTRY COSIN!(X);
     Y=X+3.141592/2;
     GOTO COM;
     ENTRY SINE!(X);
     Y=X;
COM: Z=0;
     LOOP I=1;2;I $GT$ 10;
          Z=Z+Y $P$ I/FACT!(I);
          ENDLOOP;
     RETURN Z;
     ENDPROG;
```

Note that the equivalence between formal and actual parameters is made at the ENTRY statement when the subroutine is invoked. Successive entries to a program may use formal parameters established by a previous entry. For example:

```
ENTRY ABC!(A,B,);
DECLARE INTEGER: A,B;
REUTRN;
ENTRY XYZ!;
RETURN A*B;
ENDPROG;
```

If ABC is called first and then XYZ is called, the value returned by XYZ is the integer product of the two parameters established by the entry to ABC. If, however, XYZ is called before ABC has ever been called, the value returned will be undefined.

Control may not "flow into" an ENTRY statement; i.e. there should be a GOTO or RETURN before it. A run time error occurs if this happens (see Section 3.2.2).

Subroutines usually return to their callers and, in so doing, may or may not return a value to be used as the direct result of the call. If no value is to be returned, the return statement is simply:

```
RETURN;
```

If the subroutine is to return a value, the statement has the form:

```
RETURN <exp>;
```

The <exp> may be of any type and should match the type expected by the calling program.

RETURN is an executable statement, and there may be more than one RETURN in a program. RETURN is not associated with a particular entry point so that the execution of any RETURN statement produces the same result. In the case of RETURN with a value in a subroutine with many entry points, it is the user's responsibility to ensure that the value returned is of the type expected by the entry point actually invoked.

It is possible to return to a point in the calling program other than whence the call was made. This is achieved by declaring one of the formal parameters to have type LABEL and passing a LABEL constant or variable as the appropriate actual parameter. A GOTO the formal parameter then transfers control to the address passed. Of course, it is not possible to return a value by this method.

2.4.3.2 Subroutine Calls

In CESSL, the subroutine structure is not recursive; in particular, a subroutine may not "call" another entry point in the same program.

In a call on a subroutine, the matching between actual and formal parameters takes place on a "call-by-reference" basis; that is, the information passed is an address. This is essentially equivalent to call-by-name for actual parameters which are variables, and call-by-value for actual parameters which are expressions. For example,

```
X=SUB!(A,B*3+D,C(2*E));
```

is a call-by-value for "B*3+D", and call-by-name for "A" and "C(2*E)". Call-by-name allows for "side-effects", i.e., a variable in the parameter list in the calling program may be changed by the subroutine.

CESSL-compiled programs may call subroutines prepared by other language processors, and may be called as subroutines by such programs. Care must be taken in the other programs to match the calling sequence protocol expected by CESSL programs (See Section 4.6.1). In general, parameters are compiled as a list of addresses after the CALL statement; this is compatible with TSX FORTRAN. TSX FORTRAN arrays, however, run backwards in storage, while CESSL arrays run forward, so that care must be exercised in this respect. It is not possible to include FORTRAN EXTERNAL references or CESSL FUNCTION references in parameter lists of subroutine calls from one language processor to another (See Section 4.6.2 for an explanation and an exception). In addition, FORTRAN compiled calls expect--and subroutines may return--only INTEGER or REAL values.

2.4.4 Internal Functions

Internal functions are strictly local subroutines included in a larger program. They are very similar to the construction of the same name in MAD/7090 and are similar to ALGOL/60 procedures. As subroutines they may take formal parameters and may or may not return a value. They may have many return points, but unlike external functions, they may have only one entry point. The advantage they have over external functions is that they may reference variables in the containing program as well as the parameters of the particular invocation.

The form of an internal function is

```
INTERNALFUNCTION λ!(<λ-list>);
    <code>
ENDFUNCTION;
```

The λ-list is the formal parameter list (in the case that there are formal parameters). *The λ-atoms so declared may be used only for formal parameters of internal functions - they may not be used for any other purpose.* These parameters must be declared as having some type. If the function returns a value, the function name must be declared to be of the appropriate type.

The return statement is:

```
FUNCTIONRETURN <expression>;
```

where the expression need not appear if no value is to be returned. The returned expression must be of the same type as the function name. Statements not legal in the scope of an internal function definition are INTERNALFUNCTION, ENTRY, and ENDPROG.

Internal functions are invoked exactly as external functions. Internal function names are implicitly declared to have attribute FUNCTION so that they may appear in subroutine parameter lists as FUNCTION names without further declaration.

The user must insure that branches do not occur into or out of the scope of the definition (this implies that IFs and LOOPs must be closed in the definition). The internal function may occur anywhere in the physical layout of the program, with the restriction that control may not "flow into" the entry point of the function; a run-time error will occur if this happens. Similarly, control may not "flow into" the ENDFUNCTION statement. There may be no more than 64 internal functions in any one program.

Example:

```
INTERNALFUNCTION NORMAL!(X);
DECLARE REAL:NORMAL,X;
FUNCTIONRETURN EXP(-(X*X));
ENDFUNCTION;
     .
     .
     .
VALUE=NORMAL!(2*ABCISSA)+1.0;
```

Another example:

```
INTERNALFUNCTION XYZ!(P1,P2,P3);
DECLARE INTEGER:P1,P2;
DECLARE BOOLEAN:P3,XYZ;
IF P3;
      FUNCTIONRETURN P1 $GT$ P2;
ELSE;
      FUNCTIONRETURN P1 $LT$ P2;
ENDIF;
ENDFUNCTION;
```

And another:

```
INTERNALFUNCTION CALC!;
DECLARE REAL:CALC;
     FUNCTIONRETURN A*EXP(-X*Y)+SIN(Y);
ENDFUNCTION;
```

2.5 Program Format

A program is a sequence of statements ended by ENDPROG. The compiler reads source lines from the card reader immediately after the compiler control cards (see Section 3.1). Use of the INCLUDE statement will direct the compiler to take source lines from a logical file. No more input is read after the ENDPROG statement is encountered. The next card in the card reader should be a TSX control card, e.g. "*STORE", "// XEQ", etc.

A program may be either "main" or "subroutine" (the latter includes what FORTRAN calls subprograms and functions), specified by the inclusion of statements "DECLARE...NAME" or "ENTRY ...", respectively. Both may not appear, although up to fourteen ENTRY statements may be included in one subroutine. The symbol in the "DECLARE...NAME" statement must appear as a label somewhere in the program to specify where execution of the main program should begin.

Main programs which are in non-process coreloads normally return control to the system by an

```
EXECUTE EXIT!;
```

statement. They may also end by transferrring control to another coreload by use of the LINK operator as in:

```
EXECUTE LINK OTHER;
```

Subroutines normally terminate by RETURNing to their callers although they may also call EXIT or LINK.

CESSL programs may be included in TSX Process and Interrupt coreloads, from which the appropriate exits must be taken, calls on TSX subroutines VIAQ and INTEX, respectively.

Figure 2.10(a) shows a complete program deck and Figure 2.10(b) shows the output produced from a compilation and run of the program.

```
// JOB   111102222030333
// XEQ CESSL   FX
*COMPILE BASIC

*  GRADING PROGRAM.

  THIS PROGRAM READS CARDS WITH THE FOLLOWING FORMAT
        NAME, SCORE1,SCORE2,SCORE3,SCORE4,SCORE5
  WHERE THE NAME IS AT MOST 20 CHARACTERS LONG AND THE SCOREN ARE
   INTEGERS FROM 0 TO 100. AT MOST 100 AND AT LEAST 2 CARDS CAN BE
   READ.  THE END OF THE LIST IS INDICATED BY A BLANK CARD.

  TWO LISTS ARE PRODUCED IN THE OUTPUT.  THE FIRST IS AN ALPHABETIC
  LISTING OF ALL THE NAMES WITH THE MEAN OF THE ASSOCIATED SCORES.
  THE SECOND IS A RANKING OF THE MEANS.  ALSO PRINTED IS THE MEAN
   AND MEDIAN OF ALL THE INDIVIDUAL MEANS.    ;



        DECLARE GRADE NAME;

*  EACH STUDENT IS KEPT IN A BLOCK CONSISTING OF HIS NAME AND HIS MEAN;

        DEFINE T10 ARRAY TEXT SIZE 10;
        DEFINE STDT BLOCK <T10,INTEGER>: TEMPSTDT;
        SUBSTITUTE (NAM,1) (MEAN,2);



*  STUDENTS IS THE ARRAY WHERE ALL STUDENTS NAMES AND MEANS ARE KEPT
   AND RANK IS AN ARRAY WHERE JUST THE MEANS ARE KEPT.  WE USE RANK AS
   A SEPARATE ARRAY TO EASE SORTING AND PRINTOUT;

        DEFINE STDTS ARRAY STDT SIZE 100: STUDENTS;
        DEFINE I100 ARRAY INTEGER SIZE 100: RANK;

        DECLARE INTEGER: NSTDT, I, J, ACCUM, T, SCORE;
        DECLARE TEXT: CR;



GRADE:  INPUTDEV CARDS;
        OUTPUTDEV TYPEWRITER;
        CR=?15;
        NSTDT=1;                                  *START AT THE BEGINNING;
```

Sample Program: Source Cards

Figure 2.10(a)

```
READNAM: READ STUDENTS(NSTDT,NAM);
         IF STUDENTS(NSTDT,NAM,1) $EQ$ 0;    *IF ZERO, CARD WAS BLANK;
            NSTDT=NSTDT-1;                   *SO ADJUST COUNT;
            GOTO ENDREAD;                    *AND TERMINATE READING;
         ENDIF;

*  THE INPUT HAS BEEN TRIMMED OF TRAILING BLANKS AND HAS ZEROS IN
   THE UNUSED PART.  WE WANT TO PRINT IT OUT IN CONSTANT FIELD
   WIDTH, SO WE MUST PAD WITH BLANKS;

         LOOP I=10;-1;I $EQ$ 0;
            IF STUDENTS(NSTDT,NAM,I) $EQ$ 0;  *IF WHOLE WORD IS 0;
               STUDENTS(NSTDT,NAM,I)="  ";    *MAKE IT 2 BLANKS;
            ORIF STUDENTS(NSTDT,NAM,I) $BITAND$ ?FF $EQ$ 0;
                                             *IF RIGHT HALF IS ZERO;
               STUDENTS(NSTDT,NAM,I)=STUDENTS(NSTDT,NAM,I) $BITOR$ ?40;
                                             *PUT IN A BLANK;
            ENDIF;
         ENDLOOP;

         ACCUM=0;
         LOOP I=1;1;I $GT$ 5;                *READ IN FIVE SCORES;
            READ SCORE;
            ACCUM=ACCUM+SCORE;          *TOTAL ALL SCORES;
         ENDLOOP;

*  CALCULATE MEAN AND ASSIGN TO STUDENT AND RANKING ARRAYS.  DO
   THE ARITHMETIC IN FLOATING POINT TO KEEP PRECISION AND THEN
   ROUND OFF TO INTEGER SCORE;

         RANK(NSTDT)=(STUDENTS(NSTDT,MEAN)=ACCUM/5.+0.5);
         NSTDT=NSTDT+1;
         GOTO READNAM;

*  ALPHABETIZE NAMES BY SIMPLE SORT.  ALSO RANK THE SCORES;

ENDREAD: LOOP I=1;1;I $EQ$ NSTDT;
            LOOP J=I+1;1;J $GT$ NSTDT;
               IF STUDENTS(I,NAM) $GT$ STUDENTS(J,NAM);
                  TEMPSTDT=STUDENTS(I);
                  STUDENTS(I)=STUDENTS(J);
                  STUDENTS(J)=TEMPSTDT;
               ENDIF;
               IF RANK(I) $GT$ RANK(J);
                  T=RANK(I);
                  RANK(I)=RANK(J);
                  RANK(J)=T;
               ENDIF;
            ENDLOOP;
         ENDLOOP;
```

Sample Program: Source Cards

Figure 2.10(a)

```
*  NOW OUTPUT THE ALPHABETIZED LIST WITH MEANS AND THE RANKING;

        WRITE "NAME OF STUDENT        AVE            RANKED SCORES",
              CR,CR;
        ACCUM=0;
        LOOP I=1;1;I $GT$ NSTDT;
           WRITE STUDENTS(I,NAM),STUDENTS(I,MEAN),"           ",
                 RANK(I),CR;
           ACCUM=ACCUM+RANK(I);
        ENDLOOP;

        WRITE CR,CR,"MEAN SCORE:  ",FIX$(FLOAT$ ACCUM/NSTDT+0.5),
              CR,"MEDIAN SCORE:";

*  CALCULATION OF MEDIAN DEPENDS ON WHETHER THERE ARE AN ODD
   OR AN EVEN NUMBER ITEMS;

        IF NSTDT $MOD$ 2 $EQ$ 0;
           WRITE (RANK(NSTDT/2)+RANK(NSTDT/2+1))/2;
        ELSE;
           WRITE RANK(NSTDT/2+1);
        ENDIF;
        WRITE CR,CR;
        EXECUTE EXIT!;
        ENDPROG;
// XEQ GRADE
*CCEND
WASHINGTON GEORGE,50,25,75,0,100
BURR AARON, 90,91,92,93,94
NEWMAN ALFRED E.,0,1,2,3,4
STASSEN HAROLD,44,55,66,77,88
ARNOLD BENEDICT, 39,43,47,48,53
EGGS BENEDICT, 71,35,64,58,79
EGGS HAMMOND, 47,58,69,36,25
ATLAS CHARLES, 99,69,79,59,89
ATLAS HAMMOND, 88,68,78,98,58
GRANGER STEWART, 57,70,13,2,47
MACMILLAN HAROLD, 43,43,41,43,42
FRODO THE HOBBIT, 100,99,98,100,98
MERLIN THE MAGE, 52,78,46,79,82
FLOWER THE SKUNK, 12,15,17,56,32
THAT MAN FROM RIO, 72,76,74,78,83
DICKENS CHARLES, 14,95,47,62,58
BROWN CHARLIE, 0,0,0,0,0
EINSTEIN ALBERT, 100,100,100,100,100

// END
```

Sample Program: Source Cards

Figure 2.10(a)

```
 // JOB   111102222030333
 // XEQ CESSL   FX
*COMPILE BASIC



***  END OF PROGRAM  ***


SINGLE OCCURRENCE SYMBOLS:
STDT
STDTS
I100

0965
0A34
16:24, 08/09/71

     NO ERRORS IN ABOVE ASSEMBLY.
GRADE
DUP FUNCTION COMPLETED
 // XEQ GRADE
 *CCEND

CLB, BUILD GRADE


ROC  CARDN 0011  LEV.0


CLB, GRADE LD XQ
NAME OF STUDENT        AVE          RANKED SCORES

ARNOLD BENEDICT         46                 0
ATLAS CHARLES           79                 2
ATLAS HAMMOND           78                26
BROWN CHARLIE            0                38
BURR AARON              92                42
DICKENS CHARLES         55                46
EGGS BENEDICT           61                47
EGGS HAMMOND            47                50
EINSTEIN ALBERT        100                55
FLOWER THE SKUNK        26                61
FRODO THE HOBBIT        99                66
GRANGER STEWART         38                67
MACMILLAN HAROLD        42                77
MERLIN THE MAGE         67                78
NEWMAN ALFRED E.         2                79
STASSEN HAROLD          66                92
THAT MAN FROM RIO       77                99
WASHINGTON GEORGE       50               100


MEAN SCORE:      57
MEDIAN SCORE:    58


 // END
```

Sample Program: Output

Figure 2.10(b)

CHAPTER THREE

OPERATING PROCEDURES

3.1 Control Cards for the Compiler

The compiler is a program stored in the fixed area of the disk in the TSX system. It is called by the Nonprocess Monitor control card

```
1  4   8        16
// XEQ CESSL     FX
```

CESSL starts reading source cards from the card reader. The user may direct the compiler to start reading card images from a logical file by use of the INCLUDE statement (c.f., section 2.4.1.7). Source text is read until an ENDPROG statement is recognized.

The direct output from the compiler is Assembler source text. After it has produced this Assembler source, CESSL automatically calls in the (LOCG modified) TSX Assembler (4) to finish the process of producing a relocatable object program. After the whole process has been completed, the object program resides on the disk in the temporary area, and can be called for execution with a nonprocess monitor // XEQ control card, loaded to the user or fixed area with a DUP *STORE or *STORECI operation, or punched as a binary deck with a DUP *DUMP control card operation.

Note, however, that CESSL allows names with greater than five characters, while the Assembler restricts names to five characters or less. CESSL produces legal Assembler source text, so that it must map long names into legal short names. It does this by assigning a name of the form "@XXXX" to names of greater than five characters, where XXXX is a hexadecimal number. This number is printed with the name when the *LIST SYMBOL TABLE option is chosen, so that the Assembler listing and symbol table print can then be interpreted in terms of the original source program.

Before the start of compilation, the user can specify certain options by means of control cards which must immediately follow the // XEQ CESSL FX control card and precede the source program. The control cards may be in any order, except for *COMPILE BASIC or *COMPILE CELL SPACE, only one of which may appear, and which signal the beginning of the source program.

Control cards are in standard TSX control card format, i.e., an asterisk in column one, and the option specified in columns 2-72. Spaces on control cards are ignored.

As far as possible, CESSL uses control cards which are identical to FORTRAN and Assembler control cards for the same options. In particular, the Assembler control cards used actually control the action of the assembly phase. Any unrecognizable control cards will cause an error message to be printed and will then be ignored.

The control cards and their meanings follow.

*LIST SOURCE PROGRAM

The source program is listed as it is read in from cards or from a logical file.

*LIST SYMBOL TABLE bN

After the ENDPROG card has been read, the compiler's knowledge about the symbols in the user's program is listed. Such a listing is probably useful only to those performing machine level debugging. This listing alone does not provide the relative addresses of these symbols in the object program. That information is prepared by the Assembler and may be obtained by use of the *PRINT SYMBOL TABLE control card. "b" indicates at least one space. N, if it appears, must be an integer constant (decimal or hex); it controls the type of information dumped. Four kinds of information may be listed, separately or in combination. The categories and numbers associated are:

Variables	1
Data Types	2
INTEGER, REAL, and TEXT constants	4
LABEL constants	8

Combinations may be obtained by summing the associated mumbers. For example, "*LIST SYMBOL TABLE 1" would yield just the variables, "*LIST SYMBOL TABLE 9" would yield variables and LABEL constants, and "*LIST SYMBOL TABLE 15" or "*LIST SYMBOL TABLE ?F" would get everything. If no number is given, the default is 11_{10}, i.e. data types, LABEL constants, and variables are dumped.

Each symbol dumped occurs on a separate line of the listing. For most symbols, the first item on the line is the "@-equivalent" for that symbol (see above and Section 4.1). Then comes the symbol itself, followed by information about it, depending on its category.

The category "variable" includes two sub-categories, lambda-symbols and substitute symbols. If the lambda-symbols have any special attributes, they are indicated by single letters:

E The symbol is an entry point. It either appeared in an "ENTRY" statement or it is one of the CELLSPACE "entry points".

P The symbol is a formal parameter. It appeared in the formal parameter list of an "ENTRY" or "INTERNALFUNCTION" statement.

F The symbol has FUNCTION attribute. It appeared in a "DECLARE FUNCTION:","ENTRY", or "INTERNALFUNCTION" statement.

I The symbol is the name of an internal function.

After any of these letters, the type of the symbol, if any, is printed. Finally, if the symbol is defined by an EQU, the characters "EQU:" are printed followed by the defining atom. This latter printout is useful

in determining the order in which EQU symbols are given to the Assembler since symbols are listed in the table in the order in which they will be defined. SUBSTITUTE atoms appear merely as the atom, an equal sign, and the substitution atom.

The category "data type" is also made up of two sub-categories, arrays and blocks. For both, two numbers are printed, respectively the number of components in the data type and the total number of words required for a variable of this type. For arrays, the next symbol printed is the name of the component type. For blocks, the next symbols given are the names, in order, of the component types.

For the categories of constants, just the name of the constant is given. All LABEL constants appear in the program. Note that not all of the other constants are necessarily used in the program, but if they are, the @-equivalences for them are given by these table entries. In addition, INTEGER constants appear as literals, so their @-equivalences do not appear in the assembly source program produced.

If the number 16 (hex 10) appears on the control card, the entire symbol table will be dumped, uninterpreted, largely as hex constants. This is mainly for the benefit of the compiler writer, and is mentioned here only as a warning to inaccurate keypunchers who will receive much more information than they ever wished to see.

Figure 3.1 contains the symbol tables produced by CESSL and the Assembler for the sample program of Figure 2.10. Figure 4.1 also contains a CESSL symbol table.

If output is directed to the typewriter, "Z" tabs should be set.

***PRINT SYMBOL TABLE**

The symbol table produced by the Assembler is listed. This table gives

the relative locations of the symbols in the object program. Symbols with absolute values are preceded by the letter A. See Figure 3.1.

*LIST

The Assembler listing is printed. Note, however, that CESSL puts out a LIST OFF pseudo-op at the beginning of the generated code, and a LIST ON pseudo-op at the end of the code, so that the active program code will not be listed. Only the data type, variable, and constant allocation statements and the Transition Control Block (if this is a Cell Space compilation) will be listed in Assembler format. Note that when Assembler output is printed on the typewriter, "Z" tabs must be set (see LOCG internal memo).

*LIST OVERRIDE

The entire Assembler listing will be printed, i.e., the LIST OFF pseudo-op will be ignored.

*COMMON bN

N = the length of COMMON in words. "b" indicates at least one space. This allows the use of CESSL programs which assign COMMON themselves or the linking of FORTRAN and CESSL programs where the FORTRAN programs use COMMON. (Note that the main program of a core load must specify the length of COMMON.) The length may be given in decimal or hexadecimal, as in:

```
*COMMON     523
*COMMON     ?20B
```

*SWITCHES

This is mainly a debugging aid for the compiler-writer, although some options may be of interest to others. This control card directs

the compiler to read the data switches on the computer console and to perform special actions according to the setting of the switches. The particular actions and their associated switches may be found in the documentation for the internals of the compiler.

*COMPILE BASIC

This control card specifies that only the basic compiler is wanted, with none of the special checks and definitions for the simulation language. It also signals the end of the control cards; i.e., the next card reader is the start of the source program.

*COMPILE CELL SPACE

This control card specifies that this compilation should use the special definitions and make the special checks for the CELL SPACE simulation system (3). In particular, if the program is a "main" program, the Transition Control Block and a call to CLSPC is compiled into the program. If the program is a subroutine, neither of the above will be compiled into it, but it will have the special definitions available to it. This control card also signals the end of the control cards; i.e., the next card from the card reader is the start of the source program.

3.2 Compiler messages

Compiler messages inform the user of the progress attained in translating his program. The compiler has three distinct phases during which messages may be issued: first pass (reading the source program and parsing), second pass (producing assembly source code), and assembly. Messages referring only to the Cellular Space Simulation System are included herein for completeness sake.

3.2.1 Normal messages

The compiler will always print out the control cards read on the first pass. If the source program is not listed and if no parsing errors occur (see below), the next messsge printed is

*** END OF PROGRAM ***

indicating that the ENDPROG statement has been recognized. Additional errors may be discovered at this time. If there are any user-defined symbols (other than constants) which occurred only once in the program, the compiler types:

SINGLE OCCURRENCE SYMBOLS:

and lists them. Such occurrences are not considered fatal errors; the list is provided merely as an aid to the programmer in detecting misspellings.

The symbol table is printed at the end of the first pass if it was requested by the "*LIST SYMBOL TABLE" control card.

The next two lines normally printed are hexadecimal numbers which appear for the following reason. CESSL has a limited storage space for symbol and constant definitions and no provision for table overflow. The numbers give the amount of memory remaining for such use in the two passes of the compiler.

Programs will not compile if either number dips much below 100_{10} (see 3.2.2). An average of about ten words are needed for each new symbol or constant added, so that users may be able to judge the feasibility of expanding programs.

After the second pass of the compiler, the LOCG Assembler is called. It types the date and time, (optionally) the assembly listing and symbol table, and terminates (hopefully) with the message

NO ERRORS IN ABOVE ASSEMBLY

The Assembler transfers control to the TSX Disk Utility Program to store the compiled program in the temporary area on the disk.

```
 // JOB   111102222030333
 // XEQ CESSL   FX
*LIST SYMBOL TABLE
*PRINT SYMBOL TABLE
*COMPILE BASIC



***  END OF PROGRAM  ***


SINGLE OCCURRENCE SYMBOLS:
STDT
STDTS
I100

VARIABLES:
@3A01  TEMPSTDT            STDT
@39C8  STUDENTS            STDTS
@39B4  RANK                I100
@39AA  NSTDT               INTEGER
@39A2  I                   INTEGER
@399A  J                   INTEGER
@3990  ACCUM               INTEGER
@3988  T                   INTEGER
@397E  SCORE               INTEGER
@3975  CR                  TEXT
@3887  EXIT

SUBSTITUTE ATOMS:
NAM=1
MEAN=2

DATA TYPE DEFINITIONS:

ARRAYS:
@3A1A  T10                  10     10  TEXT
@39D4  STDTS               100   1100  STDT
@39BE  I100                100    100  INTEGER

BLOCKS:
@3A10  STDT                   2     11  T10, INTEGER

LABEL CONSTANTS:
@3A23  GRADE
@3961  READNAM
@3955  ENDREAD

0911
0A34
```

Symbol Table: Compiler

Figure 3.1(a)

```
12:03, 08/13/71

                                SYMBOL TABLE
     ACCUM 0244    CR    0247  A FALSE 0000    GRADE 0000    I     0242
     I100  0746    J     0243    LABEL 0739    L#001 0754    L#002 0755
     L#003 0756    L#004 0757    L#005 0758    L#006 0759    L#007 075A
     L#008 075B    L#009 075C    L#010 075D    L#011 075E    L#012 075F
     L#013 0760    NSTDT 0241    RANK  0694    REAL  0736    SCORE 0246
     STDT  073E    STDTS 0743    T     0245    TEXT  0738  A TRUE  0001
     T10   073B    #DE01 074C    #RETR 074A    #TEM  0754    #00A0 0235
     #00A1 0224    #00A2 0235    #0010 0030    #0011 0030    #0020 003C
     #0021 0036    #0022 0083    #0030 0081    #0031 005E    #0032 0081
     #0040 0092    #0041 008C    #0042 00A8    #0050 00D2    #0051 00CC
     #0052 016E    #0060 00EC    #0061 00E6    #0062 016C    #0070 013B
     #0071 013B    #0080 016A    #0081 016A    #0090 0189    #0091 0183
     #0092 01CA    @3A01 06F8    @3B25 073A    @3B45 0737    @3B5A 0735
     @38A5 0727    @38B6 0721    @38C8 0708    @38FB 0706    @3896 072E
     @39C8 0248    @3904 0704    @3934 0703    @3955 00C6    @3961 0010
     NO ERRORS IN ABOVE ASSEMBLY.
GRADE
DUP FUNCTION COMPLETED
 // END
```

Symbol Table: Assembler

Figure 3.1(b)

3.2.2 Compiler error messages

Any messages but the above report error conditions. A program which contains an error will not assemble correctly (at least one line of the Assembler source produced will have an "ILLG" pseudo-op which will be printed as an error line) and will not be sent to the disk.

Error messages are of five types: problems with the compiler itself, pass one errors (syntax), pass two errors (semantics), Assembler and Loader.

3.2.2.1 Compiler failure

As stated above there is a limited amount of memory for the compilation process. If at any time there is overflow of the table space, CESSL types

```
CORE FULL - TERMINATE
```

and pauses. This is an immediately fatal error; compilation cannot be continued. After the user presses CONSOLE INTERRUPT control is returned to the non-process monitor to start the next job.

Another internal problem may result in the message

```
DISK ERROR - COMPILATION TERMINATED
```

This appears at the end of the first pass and is fatal (the second pass is not performed). Errors of this type should be reported immediately to the staff, for it indicates an internal failure.

The message

```
I05  //BLANK CARD
RESTART
```

is not a CESSL message; it comes from the TSX Non-process Monitor, and causes immediate termination of the job. It may occur on the first pass of the compiler and is caused by the reading of a TSX monitor control card

(slashes in the first two columns and blank in column three). This in turn is usually caused by the absence of ENDPROG as the last statement of the program.

3.2.2.2 Syntax Errors

There are a lot of ways the user can go wrong--more ways, in fact, than he can go right. The following errror messages detail the wrong ways that the compiler will detect. When CESSL detects an error, it will print the source line currently being processed (if it has not yet been listed), and one of the two forms:

*** MESSAGE

*** MESSAGE: ATOM

where ATOM refers to the particular atom which is misplaced, in error, or at which the error was detected. The error itself may be on the line listed or on the previous one. Quite often, the occurrence of one error will generate others, sometimes mysteriously, so that clearing up one condition may solve a whole series of error comments.

The user is reminded that one of the two most common errors is neglecting to end statements with semicolons, leading to the incorrect parsing of two statements. The other is forgetting to separate two atoms with a special character or space--especially atoms with $, which is an alphabetic, not special, character.

Unless otherwise noted, errors are fatal (i.e., the program cannot be stored on the disk or executed).

Procedural -- during pass one

LEXICAL ERROR-ATOM TOO LONG: ATOM

The atom printed is the first 62 characters of an atom which is too long. This error is not fatal in itself--the remainder of the atom is ignored--but it may lead to parsing errors.

ILLG CHARACTER IN #: ATOM

ATOM is a numeric constant which contains an illegal character, i.e. it is a decimal integer with a character other than 0-9, a hexadecimal integer with a character other than 0-9, A-F, or a real number with a character other than 0-9, . (period), or "E".

STACK UNDERFLOW

Compiler error; see a staff member.

STACK OVERFLOW

The expression being parsed is too complicated for the compiler; it must be broken down into less complex parts.

PARSING ERROR DETECTED AT ATOM: ATOM

A parsing failure was detected at ATOM or at the immediately preceding atom.

STATEMENT MUST END WITH ; NOT: ATOM

The parsing of a statement was complete up to ATOM, and the compiler didn't expect any more. It wants a semi-colon.

NOT LAMBDA ATOM: ATOM

ATOM must be a lambda atom; however, it has already been defined as something else, e.g., operator, data type, etc.

NOT INTEGER CONSTANT OR NOT OF ACCEPTABLE MAGNITUDE: ATOM

ATOM should have been an integer constant, or is out of bounds for the current context (mainly Cell Space).

(NEEDED FOR GROUPING NOT: ATOM
) NEEDED FOR GROUPING NOT: ATOM

A SUBSTITUTE statement is ill-formed.

NOT DEFINE KEYWORD: ATOM

In a statement of the form "DEFINE ... ATOM ...;"

ATOM is not a legal keyword.

NOT DECLARE KEYWORD: ATOM

In a statement of the form "DECLARE ... ATOM ...;"

ATOM is not a legal keyword.

SHOULD BE "SIZE": ATOM

The "DEFINE ... ARRAY ..." statement is ill-formed.

BLOCK DEF MUST START WITH < NOT: ATOM
> MISSING

The "DEFINE ... BLOCK ..." statement is ill-formed.

NOT A TYPE: ATOM

The syntax requires that ATOM have the attribute type, and it does not.

INELIGIBLE ATOM: ATOM

1) A DECLARE or DEFINE statement attempted to assign an attribute to ATOM; ATOM has either had that attribute already assigned, or may not have that attribute assigned to it. 2) The atom in a DATA statement is not λ or has already occurred in a DATA statement. 3) The right hand side of a SUBSTITUTE statement is of the form -λ where λ is not an INTEGER or REAL constant.

ILLEGAL REPEAT COUNT

The DATA statement is ill-formed.

EXTERNAL NAME TOO LONG: ATOM

In the TSX operating system, external names (subroutine entry points and main program names) must be five or fewer characters.

ENTRY POINT NAME NOT LAMBDA SYMBOL: ATOM

In an "ENTRY ATOM!" or "INTERNALFUNCTION ATOM!" statement, ATOM must be a lambda symbol.

ENTRY SYNTAX ERROR AT: ATOM

An ENTRY or INTERNALFUNCTION statement is ill-formed.

TOO MANY ENTRY POINTS (14 MAX)

A maximum of fourteen entry points are allowed for subroutines compiled in the TSX system.

KEYWORD IN RESTRICTED CODE: ATOM

INTERNALFUNCTION definitions may not contain certain keywords. See Section 2.4.4.

MAY OCCUR ONLY IN FUNCTION DEF: ATOM

The statements FUNCTIONRETURN and ENDFUNCTION may occur only in an internal function definition.

ALREADY ASSIGNED: ATOM

There are two "DECLARE ATOM...;" statements.

LABEL ILLEGAL, IGNORED

The statement just processed was not an executable statement so that the label it contained was meaningless or dangerous. This is not a fatal error in itself, but does result in the label not being defined, which may cause problems in a GOTO statement later on.

ITERATION VARIABLE NOT SIMPLE

Iteration variables must be unsubscripted.

ENDLOOP WITHOUT LOOP

An ENDLOOP occurred when no LOOP statement was outstanding. This may be the result of an error in a previous LOOP statement.

ORIF, ELSE, OR ENDIF WITHOUT IF

One of the named statements occurred when no IF statement was outstanding. This may be the result of a syntax error in a previous IF statement, causing it to be ignored.

TOO MANY IF BLOCKS

Only 14 ORIF and ELSE statements are allowed.

IF STARTING IN LOOP SCOPE MUST END IN SCOPE

IF-ENDIF groupings may not straddle an ENDLOOP statement; i.e., a "Legal IF" statement starting in a LOOP must end in that loop.

IF/LOOP NESTING DEPTH EXCEEDED

The combined nesting level maximum (about 30) for IFs and LOOPs has been exceeded. The program should be reorganized.

Procedural-end of pass one

ENDLOOP MISSING
ENDIF MISSING

A LOOP or IF statement was not closed by an ENDLOOP or ENDIF statement. As many such messages will appear as there are statements left unclosed. The messages will appear in the inverse order in which the associated header appeared. E.g., if both ENDLOOP MISSING and ENDIF MISSING appear, in that order, then the IF statement left unclosed occurs earlier in the program than the LOOP statement left unclosed.

INTERNALFUNCTION DEFINITION NOT CLOSED

The ENDFUNCTION statement was left off some internal function.

ONLY ONE OF NAME & ENTRY ALLOWED

A program is either main (NAME) or a subroutine (ENTRY). It may not be both.

LABEL NOT DEFINED: ATOM

ATOM was declared in a "DECLARE ATOM ...;" statement and must be a label in the program but has not appeared as such.

Simulation--during pass one

COORDINATE MUST START WITH <, NOT: ATOM
COORDINATE MUST END WITH >, NOT: ATOM
NEED COMMA, NOT: ATOM

A coordinate in a DEFINESIZE or DEFINEBHD statement is ill-formed.

TOO FEW COORDINATES

At least three coordinates are needed to define the space in the DEFINESIZE statement.

TOO MANY COORDINATES

At most fourteen coordinates may occur in the DEFINESIZE statement.

MUST BE CELL: ATOM

The proper form of the statement is "DEFINE CELL SAMEAS ...;". No other atom but CELL may be used in this manner.

Simulation--end of pass one

ATOM "CELL" NOT DEFINED AS DATA TYPE

The user must define "CELL" as a data type by a DEFINE ... CELL ... statement.

ONLY ONE OF INITIALVALUE AND INTIALENTRY ALLOWED

The initial state of a cell may be calculated by an initial entry point or be given a value from an array, but not both.

STATEMENT NOT ALLOWED IN SIM SUB: ATOM

The program has been defined as a subroutine (i.e. ENTRY appeared) thereby making illegal the use of the keyword ATOM.

3.2.2.3 Semantic Errors

Pass two errors are those which are discovered when the compiler must produce code corresponding to the source statements. Except as stated, all second pass errors are fatal.

The first such error which can be discovered is

ERROR: PROGRAM WITHOUT A NAME

Which signifies that neither a "DECLARE...NAME;" nor an "ENTRY" statement occurred in the program. At least one of the two must appear.

Whenever any other error message in the second pass appears, it is immediately preceded by the identification:

AT STATEMENT XXXXX+NNNN

Where XXXX is the last label (i.e. label constant, internal function, DATA statement, or entry point) defined, or "(BEGIN)" in case no label has occurred, and NNNN is the number (hexadecimal) of statements beyond the label.

The compiler may run out of space during the second pass, at which time it will print

MACRO EXPANSION OVERFLOW

and terminate processing of the current statement. The compiler will continue to process the remaining statements in order to provide as much debugging aid as possible.

Another compiler error which can occur is:

STATEMENT TOO LONG

Signifying that the statement being processed is too long for the compiler. It should be broken down.

The remaining semantic errors are:

ILLEGAL SUBSCRIPTION OF PRIMITIVE TYPE

The statement attempts to subscript a variable which is of primitive type.

ILLEGAL SUBSCRIPT TYPE

The only legal subscript types are INTEGER and REAL. Variables or expressions of any other types will produce this error.

DATA VARIABLE HAS NO TYPE: NAME

The variable in a DATA statement has not been DECLAREd to have a data type.

ILLEGAL DATA VARIABLE: NAME

The λ-atom in a DATA statement must be a true variable. It may not have any of the attributes: entry point name, internal function name, function, or formal parameter; nor may it appear as the left hand side of an EQU pair.

TOO MUCH DATA (HEX): N

Too many data items were specified for the variable in a DATA statement. This is a non-fatal error, merely informing the user of a potential problem and storage waste. N is the (hexadecimal) overflow.

ITEM HAS NO TYPE

In a READ, WRITE, or WRITEFMT list, either a variable has not been declared or a subscription has left a result of undefined type.

I/O VARIABLES MUST BE PRIMITIVE
I/O VARIABLES MAY NOT BE LABEL

With the exception of TEXTARRAYs, all input/output variables

must be of primitive type, except LABEL.

CAN'T READ INTO CONSTANTS

A constant may not appear in a READ statement.

CONS ON LEFT OF =

A constant appeared on the left of the = (equal sign).

TEXT IS NOT A TYPE NAME

In an expression of the form ABC@"DEF", DEF is not a type name.

TOO MANY INTERNAL FUNCTIONS

See Section 2.4.4.

ILLEGAL OPERATOR-DATA TYPE COMBINATION: OP TYPEA TYPEB

An expression or statement involving the operator or keyword OP occurs in the context of the two types TYPEA and TYPEB. Either type may be "(NULL)", indicating that the operator is monadic, or that the lambda atom has no type attribute associated. Consult Figures 2.5 and 2.6 for the legal operator-type combinations. If a dyadic operator has a (NULL) type as an operand, either a variable has not been DECLARE'd or a subscription was incorrect (e.g. too many subscripts, out of bounds, or variable subscription a BLOCK type).

For example,

+ INTEGER BOOLEAN

The addition of INTEGER and BOOLEAN types is not defined.

SQRT LABEL (NULL)

The square root of a LABEL is not defined.

GOTO INTEGER (NULL)

The keyword GOTO requires a LABEL as an operand.

GOTO (NULL) (NULL)

The atom in a GOTO statement must be of type LABEL. It has not been defined as such, probably because the label has been forgotten.

(NULL) INTEGER

The first operand has no type associated. It has probably not been DECLARE'd, or a subscription operation resulted in an undefined result type.

#IFF TYPEA (NULL)
#IFT TYPEA (NULL)

These monadic "operators" are generated by the compiler for IF and LOOP statements, respectively, to control the conditions therein. They require a BOOLEAN operand--anything else is an error.

The LE and LT operators will never appear in this context--they are changed by the compiler into GE and GT, respectively, with the order of the two operands switched.

3.2.2.4 Assembler Errors

Errors reported by the Assembler in the final reduction of the program to relocatable binary form are of two types: errors which are a result of errors already reported, and others.

After any compiler-reported error it is imperative to ensure that the resulting program does not get transferred to the disk for permanent storage. This is accomplished by inserting a line of the form

ILLG

into the Assembler source program. The Assembler of course will cough, print

an error message, and make the appropriate magic signals to TSX to halt such a transfer.

It is possible that additional Assembler errors resulting from compiler-reported errors will appear. These should be ignored until all the error conditions are cleared up.

An error that the Assembler will report and which has not been previously noted by the compiler concerns the format of REAL constants. This message is printed as a line of the form:

```
XXXX 00000000 0  S  @YYYY DEC    NNNNNNNN
```

as, for example

```
01A4 00000000 0 S  @3124 DEC     4.5.3E+2
```

In this case, there are too many decimal points in the REAL constant.

An error associated with the DATA statement results in the printing of the following message form:

```
XXXX 0000    0  U        DC      NNNNN
```

This usually results from using the lambda symbol NNNNN as an item in a DATA list when NNNNN has the attribute FUNCTION. While such a specification would be nice, it can't be done in TSX.

The other error detected only by the Assembler is on a user defined EQU statement. Possible violations include: incorrect order of EQU statements, illegal operators in the right hand side (only +, -, and * are allowed), and the inclusion on the right hand side (if defined by a TEXT constant) of symbols longer than five characters.

If after clearing up all the compiler-reported errors, REAL constant errors, DATA/FUNCTION errors, and EQU-definition errors, there are still Assembler errors, consult a staff member--there is probably a problem with the compiler.

When output from the Assembler is directed to the typewriter "Z" tabs must be set for Assembler error messages to print properly. (See internal LOCG memo.)

3.2.2.5 Loader Error

The only way the compiler can contribute to a loading error is if the user has used an external name (i.e. subroutine, main program, or core load) longer than five letters. CESSL translates such names to an "@-symbol", i.e. a symbol of the form "@XXXX", where XXXX is a hexadecimal number. The "@-symbol" will be given to the loader, not the original name (which the loader couldn't use anyway). It is unlikely that the loader will find a subroutine with such a name (although possible by the system naming conventions). The usual error message would be

R03 @XXXX LEV.2

resulting in an abort of the Core Load Builder and the JOB.

3.2.3 Run-time Messages

The following are run-time errors directly connected with CESSL programs. Additional, TSX, errors are possible, of course.

These errors are handled by calls to the LOCG error handling program, LOCER. The form of an error message is:

MESSAGE N1 N2 N3 N4 N5

where N2...N5 are four digit hexadecimal numbers giving additional information about the error. The entire message is typed in red.

SUBSCRIPT OUT OF BOUNDS 0003 N2 N3 N4 N5

A subscript was negative, zero, too large, or a subscription of a primitive type. N2 gives the address of the call on the subscription subroutine. N3 is the base address of the variable being subscripted. N4 is the position in the subscript list of the offending subscript (e.g., the first or fifth subscript). N5 is the value of the offending subscript.

It is most often N5 which gives the most information about the error. For example, in the case of a loop stepping through an array and not stopping in time, the first subscript error usually has a value one more than the size of the array.

A subscript error is not fatal--execution will continue. The address used in case of error is the address which would have resulted if the value in error had been one (1). For example, if in the expression ABC(1,5,3), the value 3 is in error, the address (element) returned as the result of the subscription is ABC(1,5,1). If the error had been in the 5, the address used would be ABC(1,1), which will be equivalent to ABC(1,1,1).

ILLEGAL FLOW IN CESSL PROGRAM 0001 N2 N3 N4 N5

The program "flowed" into a statement illegally, i.e., there was no branch instruction where there should have been. Programs may not flow into ENDPROG , ENTRY, INTERNALFUNCTION, or ENDFUNCTION statements. The error is fatal, and is immediately followed by an op-code error, thus allowing the TSX

system to take over the error handling and do what it wants.

N2 is the address of the statement "flowed into:.

N3, N4, N5 ay be ignored.

CALL ON DEV. INDEP. I/O WITHOUT SETUP 0002 N2 N3 N4 N5

The program called on the device independent I/O routines without having set up an input or output device. N3 always contains the address from which the call was made. If N2 is one (0001) the call was on the input routines; N4 and N5 are then the addresses from which SEFIN and SETIN, respectively, were last called, probably zero. If N2 is two (0002), the call was on the output routines; N5 contains the address which last called subroutine CRET--if non-zero, this is the culprit. The error is fatal and an op-code error immediately follows, allowing the TSX error handling routines to take over the kick-off and clean up process.

CHAPTER FOUR

THE GENERATED CODE AND RUN-TIME SUPPORT

It is a matter of fact that the language described in this manual will be used on a small computer, in a "hands-on" manner. In such an environment it is only natural that a great deal of the debugging procedure inherent in the programming process will be done on-line; that is, it will be done by sitting at the console, establishing break-points in the program, looking at special locations during the running of the program, and all those other magic things that programmers are wont to do.

To aid this process, Chapter Four attempts to explain the type of code CESSL generates for the IBM 1800 computer, and in particular, for the IBM TSX operating system. In order to make use of this chapter, at the very least, the reader should be familiar with the IBM 1800 order code (5) and the IBM TSX Assembler(4).

Decisions in implementation were often strongly influenced by the limitations of the operating system. Insofar as possible, these decisions will be marked as such in what follows for two reasons: to enable other installations to pick out the essential ideas of implementation for adaptation; and to absolve the authors from the horrendous implications of decisions forced on them.

This chapter is not meant to be (and in fact, is far from being) encyclopedic, but is meant rather as a guide. Since the language is still under development it is possible that some of the techniques discussed in this chapter will be changed somewhat, although the overall framework will probably remain undisturbed.

4.1 The Intermediate Assembly Program

As has already been stated, the direct output from CESSL is IBM TSX Assembler source code. After production of this code, the Assembler completes the task of reducing the source to relocatable binary form on the disk. In reality, the Assembler used is a version of the TSX Assembler which has been modified by members of the Logic of Computers Group Computing Staff (see internal memo). These modifications are, in the main, the addition of a literal processor and extended mnomonics similar to those for the IBM 1800 MPX Assembler. The nature of the changes should be obvious in themselves.

Since the full Assembler is available, users have the ability to select Assembler options: PRINT SYMBOL TABLE, LIST, LIST OVERRIDE, and COMMON. By using LIST OVERRIDE, the user can obtain a full assembly listing of the program generated by CESSL. In order to make use of that listing, however, additional information is needed.

Symbols and Constants

CESSL allows symbols to be of any length up to 62 characters. The assembler, however, requires symbols to be of length 5 or less. CESSL creates a five character synonym for every symbol which is longer than five characters. This synonym is then used in the creation of Assembler source text. Symbols of 5 characters or fewer are used in the generated program as is, for the convenience of the u er in reading the code. The user may obtain a listing of all synonyms in the symbol table produced at the end of the compiler (selected by the option "LIST SYMBOL TABLE"). Synonoms have the form "@XXXX" where XXXX is a hexadecimal number unique for that symbol during a particular compilation. Synonyms for symbols may differ in different compilations.

References to TEXTand REAL constants in the Assembler source text are also of the form "@XXXX", where the equivalences may be found in the compiler

symbol tables.

Symbols which the compiler must create for internal use have either pound-sign (#) or at-sign (@) as their first character. These symbols fall into three classes: transfer labels, subroutine call labels and special symbols. Special symbols are those which the compiler builds into every program for internal use such as #TEM, an array of temporary locations used in evaluating expressions. Transfer labels are of the form #XXXY, where XXXY is a hexadecimal number, and are used in loop and conditional statements, as explained in Section 4.4. Subroutine call labels are of the form "@0XXX" where XXX is a hexadecimal number; they are used as labels on a subroutine call, as explained in Section 4.6.

Assembler Program Format

The program produced by CESSL has the following format:

```
HEAD
LIST    OFF
.
.
.
code
.
.
.
LIST    ON
variable allocation
data type definitions
REAL & TEXT constants
special symbols
temporary allocation
EQU-defined symbols
LTORG
END
```

HEAD is a MAIN statement or the ENT statements for the program (for main programs and subroutine, respectively). The LIST OFF-LIST ON pair is provided so that users may obtain a listing of only the variable and constant allocations if they so desire (by choosing the option "LIST").

The "code" is explained in the rest of this chapter. Labels within the code section are assigned by an EQU statement. For example,

```
ABC EQU  *
```

Variable allocation is a series of BSS pseudo-ops which reserve locations for the variables of the program. If, however, a variable has appeared in a DATA statement, the values are outputted by DC or DEC statements at the point in the program at which the DATA statement occured. All the single word variables are allocated first, then all the variables requiring an even number of words, and finally, all the remaining variables. Variables which are formal parameters are allocated last, one word each. For all variables except those of modes INTEGER, TEXT, LABEL, and BOOLEAN, the allocation is to an even location. For example:

```
DEFINE INT3 ARRAY INTEGER SIZE 3:  I3,J3;
DECLARE INTEGER:  I,J;
DECLARE REAL:  R,S;
DECLARE LABEL:  G;
```

produces the following assembler statements.

```
I    BSS     1
J    BSS     1
G    BSS     1
R    BSS  E  2
S    BSS  E  2
I3   BSS  E  3
J3   BSS  E  3
```

Data type definitions consist of the information to be used by the run-time subscription routine. Each type name (including primitives) appears as a label on a list of DC statements. The structure of this list is explained in Section 4.3 and Figure 4.5.

Constant allocation is performed in two parts. All TEXT and REAL constants are referred to by their "@" equivalences throughout the program; for example, when the compiler must output a reference in the code to the TEXT constant

"ABC", it may put out the name "@3456". The first part of the constant allocation is performed by CESSL by putting out a series of @-labels and the constants associated. For example,

```
@3456   DC     /C1C2     TEXT constant "ABC"
        DC     /C300
@347A   DEC    1.E-6     REAL constant 1.E-6
@348D   DEC    5.1       REAL constant 5.1
```

The second part of the constant allocation is essentially performed by the Assembler. In the "code", constants of modes INTEGER, BOOLEAN, and LABEL are referenced as Assembler literals of the form ='q', where q is the constant; for example,

```
LD  L ='1'      The INTEGER constant, 1
LD  L ='ABC'    The LABEL constant, ABC
LD  L ='TRUE    The BOOLEAN constant, TRUE
```

CESSL puts out a "LTORG" statement at the end of the program so that the Assembler will dump the literals. Literals are changed by the Assembler's preprocessor to be symbols of the form L#XXX, so that this changed form is what is printed as a result of the "*LIST OVERRIDE" control card.

The special symbols defined by the compiler are

```
TRUE  EQU   1   Constant definition
FALSE EQU   0   Constant definition
#RETR DEC       Return linkage (Section 4.6)
#DEC1 DEC   1   For BOOLEAN expression evaluation (Section 4.2.3)
```

Temporary allocation consists of an array of locations which are used in the evaluation of expressions (See Section 4.2).

The allocation is made as follows:

```
        BSS  E  /XXXX
#TEM    BSS  E  0
```

where XXXX is a hexadecimal number big enough to provide the maximum number of temporaries needed for any expression. Thus, for example, "Temporary one" is addressed at location

#TEM-/0001

All symbols which appeared in EQU statements are put out at the very end so that they may refer to any of the previously defined symbols.

Figure 4.1 gives a short CESSL program and the assembly program produced by the compiler, in which the various sections are annotated.

The Assembler source program produced by CESSL may be listed by use of the "*SWITCHES" control card and by putting Data Switch 12 up. This listing shows exactly what is sent to the Assembler, without the literal-interpretation.

Index register allocation

In the TSX system, there is a creature known as a "short subroutine call", named LIBF. The call is "short" because it only takes one word of storage, instead of the two words needed for regular calls (using CALL statements). The only subroutines which are callable in this fashion are system routines which are likely to be called from many places in a program, such as the floating point arithmetic subroutines. Thus the one word call may effect a storage savings. However, in order to perform one word calls, it is necessary to have reference to an index register so that a base register/displacement type addressing structure may be used. In TSX, XR3 is used for this task, and both the assembler and the loader assume that XR3 contains the proper value, the address of a "transfer vector". CESSL-compiled programs always maintain XR3 as a pointer to a transfer vector, so that subroutines called by CESSL programs can assume that it is correct.

Index register one is used for subscript calculation and index register two is used for strictly local code sequences.

CESSL subroutines do not save index registers.

```
// JOB   111102222030333
// XEQ CESSL   FX
*LIST SYMBOL TABLE ?F
*SWITCHES
*COMPILE BASIC
*  "UNPAK" UNPACKS BYTES FROM ARRAY "FROM" TO ARRAY "TO".
   "PACK" PACKS BYTES FROM ARRAY "FROM" TO ARRAY "TO"

   THE LONGER ARRAY IS EXPECTED TO BE 80 LONG AND THE SHORTER 40 LONG.
   THE TWO ARRAYS MAY BE THE SAME ON ANY CALL, I.E. THE PACKING OR
   UNPACKING MAY BE DONE "IN PLACE";


        DEFINE I80 ARRAY INTEGER SIZE 80: FROM,TO;
        DECLARE INTEGER: I;


ENTRY UNPAK!(FROM,TO);
        LOOP I=40;-1;I $EQ$ 0;
           TO(2*I) = FROM(I) $BITAND$ ?FF;
           TO(2*I-1) = FROM(I) $RS$ 8;
        ENDLOOP;
        RETURN;


ENTRY PACK!(FROM,TO);
        LOOP I=1;1;I $GT$ 40;
           TO(I) = FROM(2*I-1) $LS$ 8+TO(2*I);
        ENDLOOP;
        RETURN;
ENDPROG;
// END
```

Compiler Output Example: CESSL Source

Figure 4.1(a)

```
 // JOB   111102222030333
 // XEQ CESSL   FX
*LIST SYMBOL TABLE ?F
*SWITCHES
*COMPILE BASIC



***  END OF PROGRAM  ***


SINGLE OCCURRENCE SYMBOLS:
I80

VARIABLES:
@3A1A  FROM          P     I80
@3A11  TO            P     I80
@3A09  I                   INTEGER
@39FF  UNPAK        E F
@39AF  PACK         E F

DATA TYPE DEFINITIONS:

ARRAYS:
@3A24  I80              80    80  INTEGER

LABEL CONSTANTS:

CONSTANTS:
@3A2D  0
@39E8  40
@39DE  -1
@39D4  2
@39CB  /FF
@39C3  1
@39BB  8

0A33
0C06
```

Compiler Output Example: Assembler Source

Figure 4.1(b)

```
       ENT      UNPAK
       ENT      PACK
       LIST     OFF
*** AT STATEMENT  (BEGIN) +0004
       CALL     $FLOW
UNPAK  EQU      *
       NOP
       STX   L1 #RETR
       LDX   I1 UNPAK
       LD     1 /0000
       STO   L  FROM
       LD     1 /0001
       STO   L  TO
       MDX    1 1+/0001
       STX   L1 #RETR+1
*** AT STATEMENT  UNPAK+0001
       LD    L  ='40'
       STO   L  I
       B     L  #0010
*** AT STATEMENT  UNPAK+0001
#0011  EQU      *
       LD    L  I
       A     L  ='-1'
       STO   L  I
*** AT STATEMENT  UNPAK+0001
#0010  EQU      *
       LDX   L2 #DE01
       LD    L  I
       SKP      +-
       MDX    2 1
       LD     2
       BNZ      #0012
*** AT STATEMENT  UNPAK+0002
       LD    L  ='2'
       M     L  I
       XCH
       STO   L  #TEM-/0001
       LIBF     $UBSC
       DC       I80
       DC       TO+/8000
       DC       #TEM-/0001+/8000
       STX   L1 #TEM-/0002
       LIBF     $UBSC
       DC       I80
       DC       FROM+/8000
       DC       I+/8000
       LD     1
       AND   L  ='/FF'
       STO   I  #TEM-/0002
```

Compiler Output Example: Assembler Source
Figure 4.1(b)
(Continued)

```
*** AT STATEMENT  UNPAK+0003
      LD    L  ='2'
      M     L  I
      XCH
      S     L  ='1'
      STO   L  #TEM-/0001
      LIBF     $UBSC
      DC       I80
      DC       TO+/8000
      DC       #TEM-/0001+/8000
      STX   L1 #TEM-/0002
      LIBF     $UBSC
      DC       I80
      DC       FROM+/8000
      DC       I+/8000
      LD     1
      SRA      8
      STO   I  #TEM-/0002
*** AT STATEMENT  UNPAK+0004
      B     L  #0011
#0012 EQU      *
*** AT STATEMENT  UNPAK+0005
      B     I  #RETR+1
*** AT STATEMENT  UNPAK+0006
      CALL     $FLOW
PACK  EQU      *
      NOP
      STX   L1 #RETR
      LDX   I1 PACK
      LD     1 /0000
      STO   L  FROM
      LD     1 /0001
      STO   L  TO
      MDX    1 1+/0001
      STX   L1 #RETR+1
*** AT STATEMENT  PACK+0001
      LD    L  ='1'
      STO   L  I
      B     L  #0020
*** AT STATEMENT  PACK+0001
#0021 EQU      *
      LD    L  I
      A     L  ='1'
      STO   L  I
*** AT STATEMENT  PACK+0001
#0020 EQU      *
      LDX   L2 #DE01
      LD    L  I
      CMP   L  ='40'
      MDX    2 1
      NOP
      LD     2
      BNZ      #0022
```

Compiler Output Example: Assembler Source
Figure 4.1(b) (Continued)

```
*** AT STATEMENT  PACK+0002
      LIBF    $UBSC
      DC      I80
      DC      TO+/8000
      DC      I+/8000
      LD   L  ='2'
      M    L  I
      XCH
      S    L  ='1'
      STO  L  #TEM-/0001
      STX  L1 #TEM-/0002
      LIBF    $UBSC
      DC      I80
      DC      FROM+/8000
      DC      #TEM-/0001+/8000
      LD    1
      SLA     8
      STO  L  #TEM-/0003
      LD   L  ='2'
      M    L  I
      XCH
      STO  L  #TEM-/0004
      LIBF    $UBSC
      DC      I80
      DC      TO+/8000
      DC      #TEM-/0004+/8000
      LD   L  #TEM-/0003
      A     1
      STO  I  #TEM-/0002
*** AT STATEMENT  PACK+0003
      B    L  #0021
#0022 EQU     *
*** AT STATEMENT  PACK+0004
      B    I  #RETR+1
*** AT STATEMENT  PACK+0005
      CALL    $FLOW
```

Compiler Output Example: Assembler Source

Figure 4.1(b) (Continued)

```
       LIST    ON
I      BSS     /0001
FROM   BSS     /0001
TO     BSS     /0001
@3B5A  DC      /0000
REAL   DC      /0000
@3B45  DC      /0000
TEXT   DC      /0000
LABEL  DC      /0000
@3B25  DC      /0000
180    DC      /0050
       DC      /8000+@3B5A
       DC      /0001
TRUE   EQU     1
FALSE  EQU     0
#RETR  DEC
#DE01  DEC     1
       BSS  E  /0005
#TEM   BSS  E  0
       LTORG
       END
15:42, 08/13/71

     NO ERRORS IN ABOVE ASSEMBLY.
UNPAK PACK
DUP FUNCTION COMPLETED
 // END
```

Compiler Output Example: Assembler Source

Figure 4.1(b) (Concluded)

4.2 Expression Evaluation

The contortions necessary to evaluate an expression are fairly easy to understand once the reader has a general idea of how subscripts and formal parameters are handled. For a complete understanding of these topics, refer to sections 4.3 and 4.6, respectively. For the purposes here, the following explanation may suffice.

If A is a formal parameter, after the entry sequence has been executed the location with the name A will contain the address of the actual parameter to the call. Thus, all references to the formal parameter are done on an indirect basis.

Subscript calculations are handled via subroutine call; on return from the subroutine, index register one (XR1) has the address of the subscripted item. For example, if A is an array of integers, the subscript calculation for A(4) would return the address A+3 in XR1. Thus, references to that item are through XR1. Now, if XR1 is needed for some other reason (e.g., another subscription or a subroutine call intervenes), the current contents of XR1 must be saved in a temporary location. Thereafter, references to that item are performed indirectly through the temporary.

CESSL does not perform optimization of expression evaluation to any great extent. There is no attempt to detect common subexpressions (even for subscripted variables) nor is there any larger program analysis done. "Peep-hole" optimization is performed to the extent of register allocation and temporary use. It is therefore very easy to follow the code produced by CESSL for strictly arithmetic operations. For example, in the completely integer expression:

```
I = J+4-L
```

the code produced is:

```
LD    L   J
A     L   ='4'
S     L   L
STO   L   I
```

Note that all direct references to variables and constants are long; that is, there is no attempt to set up a base register and displacement scheme. It just isn't practical for a compiler on the IBM 1800. The displacement address space is not large enough nor are there enough index registers to make it practical. Temporary locations are not reused within one statement for the sake of simplicity.

4.2.1 Arithmetic

INTEGER and BOOLEAN arithmetic is straightforward and requires no explanations. See Figure 4.2 for examples. Mixed mode (REAL and INTEGER) operations proceed by first converting INTEGER to REAL.

All REAL operations are performed using the TSX standard precision floating point subroutines (cf. IBM 1800 TSX System Subroutine Library (7)). These routines are called via LIBF statements and may take a parameter which is the address of a REAL number. For example,

```
LIBF    FADD
DC      A
```

will add the REAL number at location A to the floating accumulator (FAC).

Since these routines were designed for use with FORTRAN, there are alternate entries to most of them allowing the routines to help in subscription. Such entries have names which end in X. The effect is that the contents of index register one is added to the parameter address following the call, with the resulting sum being used as the actual address. It is not entirely fortuitous that in CESSL the subscript calculator returns the address in XR1, for then the call may be made on the X'ed routine with a parameter of zero.

The floating point routine will add the zero to XR1 and get the address to use. For example,

```
(subscript call)
LIBF    FADDX
DC      0
```

will result in the floating addition of the REAL number given by the address contained in XR1. Calls to the floating point routines are made in such a manner as to try to optimize the code for the subscripts calculated. See Section 4.6 for additional information about parameters to subroutine calls.

4.2.1 Intrinsic Functions

The intrinsic functions (ABS$, FIX$, FLOAT$, SQRT, ATAN, ALOG, SIN, COS, EXP, TANH, LINK) are treated in CESSL as monadic (one operand) operators in order to provide local optimization with respect to the calls on the TSX subroutines to calculate these functions. For example, there are three subroutines which may be called to calculate the absolute value of a number, depending on its mode (INTEGER or REAL) and its location (in memory or in the FAC).

Examples of intrinsic function calls may be found in Figure 4.3.

4.2.3 Relational Operators

The relational operators are EQ, NE, GE, GT, LE, AND LT. The mixed mode case of REAL and INTEGER is handled by first converting the INTEGER to REAL, and then doing the comparison as in the REAL case. The other primitive/primitive cases are handled as if they were integers.

The result of a relation is a BOOLEAN result, i.e., either a zero or a one. In order to optimize this process (which is a little on the gruesome side) the compiler automatically defines in every program the symbol #DE01 as follows

```
#DE01   DEC    1
```

which assembles as two words, the first of which (at address #DE01) is a zero and the second of which (at address #DE01+1) is a one. Before a comparison is made, XR2 is set to the address of #DE01, i.e., it points at the zero. The comparison is then made and if the relation is TRUE, XR2 is bumped by one to point to the one. After the comparison, a load via XR2 is performed, obtaining either zero or one in the Accumulator for further use.

The cases of LT and LE are handled exactly as GT and GE respectively, with operands reversed.

INTEGER comparisons are performed using the "CMP" machine instruction and selecting the appropriate case. For examples, See Figure 4.4.

REAL comparisons are performed by floating-point subtraction of the two operands and comparison of the result to zero. This algorithm uses the TSX library routine LDFAC (LIBF call). LDFAC returns the high order bits of the mantissa (which is in twos complement form). Since all floating point numbers are normalized, this part of the mantissa is sufficient to give the less-than-zero, zero, and greater-than-zero conditions which are needed for the comparisons.

Comparison of similar, nonprimitive types and of the combinations TEXT/TEXTARRAY and TEXTARRAY/TEXT is performed by a call on a closed subroutine. There are three parameters to the subroutine, given in the index registers. XR1 contains the address of the left hand operand, XR2 contains the address of the right hand operand, and XR3 contains the length (in words) of the variables. (In the T/TA and TA/T cases this length is the minimum of the lengths of the two.) On return from the routine, the accumulator is either zero or one, and XR3 has been restored to the address of the transfer vector. The subroutines are named, reasonably enough, EQ, NE, GT, and GE. The relations LT and LE are computed by calls on the subroutines GT and

GE ,respectively, with the operands reversed.

4.2.4 Assignments

Assignment statements for similar primitive types are trivial. The expression (right hand side) is brought into the appropriate register (AC or FAC) and the result is stored from that register. REAL types are handled by the floating point routines and the other primitive types are handled as if they were integers.

Conversion code is generated when an integer expression is assigned to a REAL variable and vice-versa. The only other legal, primitive, mixed mode assignment is TEXT/INTEGER and INTEGER/TEXT, which is treated as if both types were INTEGERs.

The assignment of an expression of nonprimitive type to a similar type or of TEXT to TEXTARRAY is performed by the subroutine $MV21, which takes three arguments in the index registers. XR1 is the address of the left hand side, XR2 is the address of the right hand side, and XR3 is the length (in words) of the variables. On return from $MV21, the data movement has been performed and XR3 has been restored to the address of the transfer vector.

Embedded assignment statements are assignment statements within expressions; for example,

```
I=(J=4);
```

The embedded assignment is legal only for primitive data types. The assignment is made but the result of that parenthized expression is fixed with regard to its participation in the remainder of the current statement. In other words, the value of that embedded assignment is not the value of the variable of the assignment, it is the value of the expression of the assignment. Operationally, this may require that an extra temporary be assigned to hold

the value of the expression. For example, the above statement translates into

```
LD    L    ='4'
STO   L    J
STO   L    L
```

but for more complicated expressions we have

```
I=(J=4)*(K+M);

LD    L    ='4'
STO   L    J
STO   L    #TEM-/0001
LD    L    K
A     L    M
M     L    #TEM-/0001 .... Note use of temporary, not J.
XCH
STO   L    I
```

```
DECLARE INTEGER:  I,J,K;
DECLARE REAL:  R,S;
DECLARE BOOLEAN:  A,B,C;
DEFINE INT3 ARRAY INTEGER SIZE 3:  I3,J3;
DEFINE REAL2 ARRAY REAL SIZE 2:  R2,S2;
```

INTEGER Arithmetic

```
I = J+K;
     LD    L   J
     A     L   K
     STO   L   I
```

```
I = (J+4)*(K-3);
     LD    L  J
     A     L  ='4'
     STO   L  #TEM-/0001
     LD    L  K
     S     L  ='3'
     M     L  #TEM-/0001
     XCH
     STO   L  I
```

```
I = I3(J);
     LIBF      $UBSC
     DC        INT3
     DC        I3
     DC        J+/8000
     LD     1  0
     STO   L   I
```

```
I = K*I3(J);
     LIBF      $UBSC
     DC        INT3
     DC        I3
     DC        J+/800
     LD    L   K
     M      1  0
     XCH
     STO   L   I
```

Mixed Mode Arithmetic

```
R = S+I;
     LD    L   I
     LIBF      FLOAT
     LIBF      FADD
     DC        S
     LIBF      FSTO
     DC        R
```

```
I = R*I;
     LD    L   I
     LIBF      FLOAT
     LIBF      FMPY
     DC        R
     LIBF      IFIX
     STO   L   I
```

REAL Arithmetic

```
R = R-S;
     LIBF      FLD
     DC        R
     LIBF      FSUB
     DC        S
     LIBF      FSTO
     DC        R
```

```
R = S2(I);
     LIBF      $UBSC
     DC        REAL2
     DC        S2
     DC        I+/8000
     LIBF      FLDX
     DC        0
     LIBF      FSTO
     DC        R
```

Figure 4.2 Arithmetic

REAL Arithmetic (continued)

```
R = (S+1.0)/4.;
      LIBF    FLD
      DC      S
      LIBF    FADD
      DC      @XXX1
      LIBF    FDIV
      DC      @XXX2
      :
      :
      @XXX1 DEC    1.0
      @XXX2 DEC    4.
```

```
R = S/(R+4.);
      LIBF    FLD
      DC      R
      LIBF    FADD
      DC      @XXX2
      LIBF    FDVR
      DC      S
      LIBF    FSTO
      DC      R
```

BOOLEAN Arithmetic

```
A = B $AND$ C;
      LD   L  B
      AND  L  C
      STO  L  A

A = NOT$ B;
      LD   L  B
      EOR  L  ='1'
      STO  L  A
```

```
A = (B $OR$ C) $AND$ (C $XOR$ A);
      LD   L  B
      OR   L  C
      STO  L  #TEM-/0001
      LD   L  C
      EOR  L  A
      AND  L  #TEM-/0001
      STO  L  A
```

Non-primitive Assignment

```
I3 = J3;
      LDX  L1  I3
      LDX  L2  J3
      LDX  L3  0
      ORG      *-1
      DC       /0003
      CALL     $MV21
```

```
R2 = S2;
      LDX  L1 R2
      LDX  L2 S2
      LDX  L3 0
      ORG     *-1
      DC      /0004
      CALL    $MV21
```

Figure 4.2 Arithmetic (concluded)

```
DECLARE INTEGER:  I,J,K;
DECLARE REAL:  R,S,T;

R = SIN(S);
     CALL    FSIN
     DC      S
     LIBF    FSTO
     DC      R

R = SIN(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FSINE
     LIBF    FSTO
     DC      R

R = EXP(S);
     LIBF    FEXP
     DC      S
     LIBF    FSTO
     DC      R

R = EXP(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FXPN
     LIBF    FSTO
     DC      R

I = SQRT(J);
     LD   L  J
     CALL    XSQR
     STO  L  I

R = ABS$ S;
     CALL    FABS
     DC      S
     LIBF    FSTO
     DC      R

I = ABS$(J);
     CALL    IABS
     DC      J
     STO  L  I

R = COS(S);
     CALL    FCOS
     DC      S
     LIBF    FSTO
     DC      R

R = COS(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FCOSN
     LIBF    FSTO
     DC      R

R = ALOG(S);
     CALL    EALOG
     DC      F
     LIBF    FSTO
     DC      R

R = ALOG(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FLN
     LIBF    FSTO
     DC      R

R = SQRT(S);
     CALL    FSQRT
     DC      S
     LIBF    FSTO
     DC      R

R = ABS$(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FAVL
     LIBF    FSTO
     DC      R

R = TANH(S);
     CALL    FTANH
     DC      S
     LIBF    FSTO
     DC      R

R = TANH(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FTNH
     LIBF    FSTO
     DC      R

R = ATAN(S);
     CALL    FATAN
     DC      S
     LIBF    FSTO
     DC      R

R = ATAN(S+T);
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FATN
     LIBF    FSTO
     DC      R

R = SQRT(S+T)+S;
     LIBF    FLD
     DC      S
     LIBF    FADD
     DC      T
     CALL    FSQR
     LIBF    FADD
     DC      S
     LIBF    FSTO
     DC      R

I = SQRT(J+1)+4;
     LD   L  J
     A    L  ='1'
     CALL    XSQR
     A    L  ='4'
     STO  L  I
```

Figure 4.3 Intrinsic Functions

DECLARE INTEGER: I,J,K; DECLARE REAL: R,S;

INTEGER (and other non-REAL primitive types)

```
I $EQ$ J
     LDX  L2 #DEO1
     LD   L  I
     CMP  L  J
     NOP
     B       *+1
     MDX   2 1
     LD    2 0

I $GE$ J
     LDX  L2 #DEO1
     LD   L  I
     CMP  L  J
     B       *+1
     B       *+1
     MDX   2 1
     LD    2 0

I $GT$ 0
     LDX  L2 #DEO1
     LD   L  I
     SKP     -Z
     MDX   2 1
     LD    2 0
```

```
I $NE$ J
     LDX  L2  #DEO1
     LD   L   I
     CMP  L   J
     NOP
     MDX   2  1
     LD    2  0

I $GT$ J
     LDX  L2  #DEO1
     LD   L   I
     CMP  L   J
     MDX   2  1
     NOP
     LD    2  0

I $GT$ (J+K)
     LD   L   J
     A    L   K
     LDX  L2  #DEO1
     CMP  L   I
     B        *+1
     MDX   2  1
     LD    2  0
```

REAL

```
R $EQ$ S
     LIBF     FLD
     DC       R
     LIBF     FSUB
     DC       S
     LIBF     LDFAC
     LDX  L2  #DEO1
     SKP      +-
     MDX   2  1
     LD    2  0

R $NE$ S
     LIBF     FLD
     DC       R
     LIBF     FSUB
     DC       S
     LIBF     LDFAC
     BZ       *+2
     LD   L   ='1'
```

```
R $GT$ S
     LIBF     FLD
     DC       R
     LIBF     FSUB
     DC       S
     LIBF     LDFAC
     LDX  L2  #DEO1
     SKP      -Z
     MDX   2  1
     LD    2  0

R $GE$ S
     LIBF     FLD
     DC       R
     LIBF     FSUB
     DC       S
     LIBF     LDFAC
     LDX  L2  #DEO1
     SKP      -
     MDX   2  1
     LD    2  0
```

Figure 4.4 Relation Evaluation

4.3 Subscription

Run time subscription requires reference to the data type of the subscripted variable. All data types have their description available at run time in the form of "dope vectors". That is, each named type is a label (in the program) on a list of words describing that type. Figure 4.5 gives the form of the dope vectors.

Subscripts are calculated by a call on subroutine $UBSC (LIBF type call). Threr are 2+n parameters to the call, where n is the number of subscripts. The first parameter is the address of the dope vector for the subscripted variable. The second parameter is the base address of the variable. If bit zero of this word is set, the word is actually a pointer to a location containing the base address of the variable (this is a convenience for formal parameters in a subroutine). The remaining n parameters are addresses of the subscript values. The last parameter has bit zero set. Thus the form of the call is

```
LIBF   $UBSC
DC     ADDRESS OF TYPE DESCRIPTION
DC     BASE ADDRESS
DC     ADDRESS OF SUBSCRIPT 1
DC     ADDRESS OF SUBSCRIPT 2
.
.
.
DC     ADDRESS OF LAST SUBSCRIPT +/8000
```

Consider Figure 2.1 in which the declarations are:

```
DEFINE   INT4  ARRAY  INTEGER  SIZE  4;
DEFINE   QQSV  BLOCK   REAL,<INT4>;
DEFINE   VSQQ  ARRAY  QQSV SIZE 3;
DECLARE  VSQQ: ABC;
```

Then A(J,2,3) is of type INTEGER and is at A+4. The call would be:

For PRIMITIVE TYPES:

word 1: zero (0)

For ARRAYs:

word 1: Number of components in the ARRAY (i.e., SIZE).

word 2: Address of dope vector for components + /8000 (i.e., bit zero set).

word 3: Number of words for each component.

Thus, word 1 times word 3 gives the number of words allocated for this array.

For BLOCKs:

word 1: Number of components in the BLOCK = N.

word 2: Address of dope vector for the first component.

word 3: Displacement to the first component in the block.

Words 2 and 3 are repeated N times, once for each component in the block.

Figure 4.5

Format of Dope Vectors (Run Time Type Descriptors)

```
LIBF   $UBSC
DC     VSQQ
DC     ABC
DC     J
DC     ='2'
DC     ='3'/8000
```

And the tables would be:

```
VSQQ   DC   3              Three components
       DC   QQSV+/8000     Array of type QQSV
       DC   6              Each component is 6 long

QQSV   DC   2              Two components
       DC   REAL           BLOCK, 1st component of type REAL
       DC   0              DISP of comp 1 = 0
       DC   INT4           2nd component of type INT4
       DC   2              Disp of comp 2 = 2

INT4   DC   4              Four components
       DC   #INTG+/8000    Array of type INTEGER
       DC   1              Each component 1 word long

#INTG  DC   0              INTEGER
#REAL  DC   0
```

The value returned by $UBSC (in index register one) is the address which is the result of the subscription. Thus, in the above example, if I is an integer, the statement

```
I = ABC(J,2,3);
```

would compile into the above call and be followed by

```
LD    1
STO L  I
```

Errors may occur at any point in the data structure if a subscript is out of bounds (i.e., less than one, or greater than the number of components in that structure). In this case an error message is printed (see Section 3.3), and the value returned in XR1 is the address which would have resulted if the subscript in error and all following subscripts had been one (1). For example, the address returned for ABC(1,2,7) would be ABC+2.

Since the subscripts are parameters to the subroutine call, and since

they may be expressions or formal parameters in the containing program, they reader should consult Section 4.6.2 (subroutine calling sequences) for all special problems concerned with fixing up parameter lists.

4.4 Loops and Conditionals

Both LOOP and IF (along with the associated statements ENDLOOP, ORIF, ELSE, and ENDIF) require the generation of special labels for handling the branches and conditional branches implicit in the statements. These labels are of the form "#XXXY" where XXX and Y are hexadecimal numbers. The numbers XXX are assigned in sequence - from 000 to 7FF - to LOOP and IF groups as they appear in the program; i.e., the first such statement is assigned sequence 000, the second 001, etc. Thus there may be a maximum of 2048 LOOP and IF statements in a single program. The number Y is used to specify parts of the construction as explained below.

LOOP and ENDLOOP

Upon detection of a legal LOOP statement the next unique sequence number, XXX, is assigned for all references to this loop construction. The loop statement is of the form

```
LOOP  <variable> = <exp1>; <exp2>; <exp3>;
```

The code produced is specified exactly as follows;

```
     <variable> = <exp1>;
     GOTO #XXX0;
#XXX1:  <variable> = <variable> + <exp2>;
#XXX0:  If <exp3>; GOTO #XXX2; ENDIF;
```

The code for the loop body is then compiled, and upon detection of the ENDLOOP statement matching the above LOOP statement, the following code is emitted.

```
     GOTO #XXX1;
#XXX2:  CONTINUE;
```

For example, consider the following program segment.

```
LOOP X=1; A+B; X $GT$ 5;
<stuffing>
ENDLOOP;
```

Let X, A, and B be declared as INTEGER mode, and let the current sequence number be "01A". Ignoring the stuffing, the assembly code produced is (See

Section 4.2.3 for explanation of relational operator code expansion):

```
          LD      L     ='1'
          STO     L      X
          B       L     #01A0
#01A1     LD      L     A
          A       L     B
          A       L     X
          STO     L     X
#01A0     LDX     I2    #DE01
          LD      L     X
          CMP     L     ='5'
          MDX       2   1
          NOP
          LD        2   0
          BNZ           #01A2
          .
          .
          <stuffing>
          .
          .
          B       L     #01A1
#01A2     EQU           *
```

IF, ORIF, ELSE, and ENDIF

Upon detection of an IF statement (just the IF part) the next unique sequence number, XXX, is assigned for all references to this IF construction. The label #XXX0 is assigned as the location to which to transfer to "get out" or terminate the IF construction; that is, all code sections which may be conditionally executed end with a GOTO #XXX0 statement. Each ORIF, the ELSE (if it appears), and the ENDIF statement are then assigned a label of the form #XXXY where Y is the serial appearance number of that statement, with Y=1 for the first ORIF statement. If the BOOLEAN expression in the IF or ORIF statement is FALSE, and if the current serial number Y is n, a branch to serial n+1 is performed. Thus it is possible to compile each statement as it appears without knowing what kind of statements follow.

Specifically, the statements are compiled individually as follows. The IF statement is of the form:

```
        IF    <exp> ;
```

and generates something like

```
    (calculate <exp>, bring into AC)
    BZ   #XXX1
```

ORIF is of a form similar to IF and generates (e.g., if n-4)

```
        B      L    #XXX0
#XXX4   EQU         *
        (calculate <exp>, bring into AC)
        BZ     L    #XXX5
```

ELSE generates

```
        B      L    #XXX0
#XXXn   EQU         *
```

ENDIF generates

```
        #XXXn  EQU    *
        #XXX0  EQU    *
```

Thus, the program segment:

```
        IF A; <stuffing 1>;
        ORIF B $OR$ C; <stuffing 2>;
        ELSE; <stuffing 3>;
        ENDIF;
```

translates into (assuming the next unique label number is 444):

```
        LD   L  A
        BZ      #4441
        <stuffing 1>
        B    L  #4440
#4441   EQU     *
        LD   L  B
        OR   L  C
        BZ      #4442
        <stuffing 2>
        B    L  #4440
#4442   EQU     *
        <stuffing 3>
#4443   EQU     *
#4440   EQU     *
```

4.5 Input/Output

I/O in CESSL is fairly straightforward since there are no implicit or explicit loops allowed in the I/O statements. For the simple statements, READ and WRITE, exactly one subroutine is called for each element in the list. For WRITEFMT the action is only slightly more complex.

Device independence is achieved through the support subroutines. The calls produced by CESSL are oblivious to the type of device being used.

4.5.1 INPUTDEV and OUTPUTDEV

The setup for an I/O device is performed by calling one of the special system routines SFOUT or SEFIN, for output and input respectively, with a parameter specifying a particular subroutine which will handle the transfer. If the user chooses one of the keywords TYPEWRITER, CARDS, PDP7, or FILE for the standard device handling, CESSL will choose the appropriate routine as follows:

	Subroutine	
Device	Input	Output
TYPEWRITER	KBDE	TYCH
CARDS	CARDE	PNCHC
PDP7	A7E	T7E
FILE	FGETA	FPUTA

For example, the statements

```
INPUTDEV CARDS;   OUTPUTDEV TYPEWRITER;
```

will translate into:

```
CALL  SEFIN       CALL  SFOUT
CALL  CARDE       CALL  TYCH
```

If the user does not select one of the standard device handling routines, the setup statement assumes that the name given is a subroutine name and passes information to SFOUT, SEFIN, STOUT, or SETIN accordingly. Which of these routines is called depends on the attributes associated with the name used. See Section 4.7 for a fuller explanation of subroutine calls using

external names as parameters.

For example, if CANDY has been defined as an internal function and FUDGE is merely an external name,

```
INPUTDEV FUDGE;   OUTPUTDEV CANDY;
```

translates into

```
CALL SEFIN          CALL    SFOUT
CALL FUDGE          BSI  L  CANDY
```

It is also possible to use a formal parameter declared as a FUNCTION, as in:

```
INPUTDEV PARA;      OUTPUTDEV PAR;
```

in which case the translations are:

```
LD   L   PARA       LD   L    PAR
STO      *+2        STO       *+2
CALL     SETIN      CALL      STOUT
DC                  DC
```

The routines STOUT and SETIN are similar to SFOUT and SEFIN, but are used for just this purpose.

It is possible to set up the devices directly by calling SETIN and STOUT (rather than using the CESSL statements INPUTDEV and OUTPUTDEV).

For example, using the above defined FUDGE, CANDY, and PARA:

```
EXECUTE SETIN!(PARA);
   LD     L   PARA
   STO        *+2
   CALL       SETIN
   DC

EXECUTE SETIN!(FUDGE):
   B          *+2
   CALL       FUDGE
   LD         *-2
   STO        *+2
   CALL       SETIN
   DC

EXECUTE STOUT!(CANDY);
   CALL       STOUT
   DC         CANDY
```

Runtime

A call on one of the routines STOUT or SFOUT must be made (directly or by OUTPUTDEV) before executing a WRITE or WRITEFMT statement. Similarly, a call on SETIN or SEFIN must be made before executing a READ statement. These setup routines establish the subroutine which is to be used for the transfer of characters into or out of the data conversion routines. The data conversion routines are: CTXTL, RDINT, RDFLT, PINTG, PREAL, PINTC, PFLTC, PBOOL, and PTXTW.

The output routines all call subroutine PRNTE to send characters to the current output device. PRNTE uses the last "device" set up by SFOUT or STOUT as a subroutine to call to handle a character. Obviously, if none has been set up, there is an error condition, for which consult Section 3.3. The subroutine called by PRNTE may expect a single EBCDIC character in the Accumulator, right adjusted, with high order bits zero. It is up to the routine to do something with the character (buffer it or translete and send on, or something) and then return.

The input routines all call CHARE, requesting it to return a single character from the input "device" last set up by SEFIN or SETIN. If none has been set up, there is an error condition. The subroutine CHARE calls is expected to return a single EBCDIC character in the Accumulator, right adjusted, with leading zeros.

The routines called by PRNTE and CHARE need not be "real" I/O routines--they may do anything they wish to the characters they receive, or may get the characters they send from anywhere, even an internal buffer.

4.5.2 READ

The elements of a READ list must be <left-designator>s which means that they must be variable names (i.e., lambda atoms) or they must be subscripted

variable names. The allowed modes of the elements are: INTEGER, BOOLEAN, REAL, TEXT, or an ARRAY of type TEXT.

The READ statement (aside from the subscription calculation needed) compiles into a series of subroutine calls, one call for each item on the list. (Consult Section 4.6.2 for some of the vagaries of subroutine calls with parameters.) Three subroutines handle all the input: RDINT (for reading INTEGER and BOOLEAN numbers), RDFLT (for reading REAL numbers), and CTXTL (for reading TEXT and TEXTARRAY). RDINT and RDFLT each take one parameter, the address of the variable which is to receive the input. CTXTL takes two parameters. The first is the address of the TEXT variable or the start of the TEXTARRAY. The second is the maximum number of words which may be filled from the input stream. CTXTL will read characters into the variable, packing them two per word up to the maximum. Surplus characters from the input will be ignored. If not enough characters appear in the input stream, the characters which appear will be left-adjusted and the remaining character positions will be filled with zeros.

for example,

```
DEFINE TXTRY ARRAY TEXT SIZE 4:  TXT4;
DECLARE INTEGER:  I,J;
DECLARE REAL:  R;
DECLARE BOOLEAN:  B;
DECLARE TEXT:  T;
.
.
.
READ I,B,R,T,TXT4;
```

produces:

```
CALL  RDINT
DC    I
CALL  RDINT
DC    B
CALL  RDFLT
DC    R
CALL  CTXTL
DC    T
```

```
DC    ='1'
CALL  CTXTL
DC    TXT4
DC    ='4'
```

Subscription for the *whole* list takes place before any of the calls are issued (See Section 4.3). For example, using the above declarations, the following:

```
DEFINE INT4 ARRAY INTEGER SIZE 4:  ZAP;
.
.
.
READ ZAP(I), ZAP(J),TXT4(I);
```

produces this code:

```
LIBF      $UBSC
DC        INT4
DC        ZAP
DC        I+/8000
STX   L1  #TEM-/0001
LIBF      $UBSC
DC        INT4
DC        ZAP
DC        J+/8000
STX   L1  #TEM-/0002
LIBF      $UBSC
DC        TXTRY
DC        TXT4
DC        I+/8000
LD    L   #TEM-/0001
STO       *+2
CALL      RDINT
DC
LD    L   #TEM-/0002
STO       *+2
CALL      RDINT
DC
STX   1   *+2
CALL      CTXTL
DC
DC        ='1'
```

Thus the address of any element in a list is the address calculated *before* any reading is performed. The statements:

```
I = 4;
READ I, A(I);
```

will result in reading values for I and A(4),no matter what the value for

I read in. This is no restriction since the input is stream-oriented rather than line-oriented, i.e., it is not the case that a new line on the input medium is obtained for every READ statement.

4.5.3 WRITE and WRITEFMT

The elements of a WRITE list must be expressions of type INTEGER, REAL, BOOLEAN, TEXT, or TEXT ARRAY. All expression evaluation is performed before the values are written out. The output is a series of subroutine calls, one for each element on the list.

The subroutines are PINTG (for INTEGER mode), PREAL (for REAL mode), PBOOL (for BOOLEAN mode), and PTXTW (for TEXT and TEXTARRAY variables and constants). PINTG, PBOOL, and PREAL each take one parameter, the address of the value to be written out. PTXTW takes two parameters: the first is the address of the text variable; the second is the address of a word containing the numbers of words in the variable.

The example, using the same declarations as in the previous section,

```
WRITE "RESULT", I,B,R,T;
    CALL   PTXTW
    DC     @XXXX  Address of TEXT constant "RESULT"
    DC     ='3'
    CALL   PINTG
    DC     I
    CALL   PBOOL
    DC     B
    CALL   PREAL
    DC     R
    CALL   PTXTW
    DC     T
    DC     ='1'
```

WRITEFMT is similar to WRITE in the output list, with the difference being in a call to set up the format list and in different subroutines called for the output.

The format list is set up by a call on subroutine WRITC with a single parameter, the address of the format list to be used on this statement.

Thereafter, there is one call per element in the write list. BOOLEAN, TEXT, and TEXTARRAYs are handled exactly as in the WRITE statement, calling the same routines. The routine PINTC is used for expressions of mode INTEGER; routine PFLTC is used for expressions of mode REAL. After all such calls there is one additional subroutine called, SPECR, with no parameters. This is done so that trailing items in the format list will be interpreted and outputted. For example,

```
WRITEFMT LIST; "RESULT", I,B,R,T;
   CALL     WRITC
   DC       LIST
   CALL     PTXTW
   DC       @XXXX
   DC       ='3'
   CALL     PINTC
   DC       I
   CALL     PBOOL
   DC       B
   CALL     PFLTC
   DC       R
   CALL     PTXTW
   DC       T
   DC       ='1'
   CALL     SPECR
```

The special case of a zero for the format list designator is handled by not making the call on WRITC.

4.6 Subroutines

The basic calling sequence assumed by CESSL for subroutine that it compiles and calls that it makes is as follows. Subroutines are entered via "BSI" statements (the loader normally produces a BSI Long instruction for the "CALL" pseudo-op), and parameters to the call (if any) are specified by a list of addresses immediately following the BSI statement. For example,

```
EXECUTE SUB!(A,B,C);
  CALL   SUB
  DC     A
  DC     B
  DC     C
```

A subroutine may return values to its caller either by "side-effects" on the parameters or by an explicitly specified value. Primitive values are expected to be found in the appropriate general register (AC for INTEGER, BOOLEAN, TEXT, and LABEL, FAC for REAL). Non-primitive returns are handled differently. The calling program sets index register one (XR1) to the address of a temporary block of storage (in the calling program) which is large enough to hold the expected return, i.e., it is a block of storage of the data type of the returned value. The called subroutine saves XR1 when it is entered, and moves the value to the block in the calling program before it returns. Because of the requirements of REAL data types, the temporary block assigned by the calling program should be even-aligned when a REAL is part of the data type of the returned value.

4.6.1 CESSL Compiled Subroutines

A CESSL program with an ENTRY statement is a subroutine. All entry names are collected into a list of Assembler pseudo-op "ENT" statements at the beginning of the program, in the order in which they appeared. The first such name is the TSX "program" name.

The code generated for an ENTRY statement must take into account the fact that there may be many ENTRY and RETURN statements in the same program, and that RETURN has no reference to a specific ENTRY. For this reason, a subroutine, at whatever point it is entered, establishes the internal variable "#RETR+1" as the address through which a RETURN will branch to return to the calling program. This is done as follows for an entry point name SID:

```
SID   EQU     *
      NOP
      LDX  I1 SID
      STX  L1 #RETR+1
```

This code picks up the contents of SID, the entry point actually involved (which is the address after the BSI instruction in the calling program), and stores it in "#RETR+1". The RETURN statement may then simply emit the code

```
B   I   #RETR+1
```

Index Register One is saved in location #RETR in case the return statement specifies a non-primitive value as the result of the subroutine:

```
STX  L1  #RETR
```

The actual parameter addresses are obtained and stored in internal locations which have the name of the formal parameter, as in the following:

```
ENTRY SUB!(A,B,C);
  SUB   EQU     *
        NOP
        STX  L1  #RETR
        LDX  I1  SUB
        LD    1  /0000
        STO  L   A
        LD    1  /0001
        STO  L   B
        LD    1  /0002
        STO  L   C
        MDX   1  1+/0002
        STX  L1  #RETR+1
```

(Note that #RETR+1 ends up with the return address, i.e., the address after the parameter list.) Thus, formal parameters in subroutines are locations which contain the addresses of the actual parameters to the subroutine.

All references to the formal parameters are performed on an indirect basis. For example, if the above parameters had been declared to have integer type,

```
A = B + C;
```

would compile into

```
LD   I   B
A    I   C
STO  I   A
```

referring in each case to the actual address in the main program, a call-by-name matching of parameters.

Note that in the case of a LABEL parameter the pick up is

```
LD   I1   /XXXX
```

since the actual parameter is the address of an address. For example, if ABC is a LABEL constant:

```
EXECUTE SUB!(ABC);
   CALL    SUB
   DC      ='ABC'
```

Since it is a run-time error to "flow into" an ENTRY statement, the following call is compiled into the program *before* the ENTRY code:

```
CALL    $FLOW
```

$FLOW is an LOCG system subroutine which will print an error message of the appropriate type (See Section 3.3).

4.6.2 Subroutine Calls

This section describes the code produced by CESSL for normal subroutine calls, i.e., subroutines which expect a parameter list of the sort described in 4.6.1.

All parameters to a call are calculated before any calling sequence adjustments are performed, in order from left to right, with temporary locations assigned for the results of any expressions or subscriptions.

There are several conditions under which the calling sequence needs adjustment:

1) The subroutine called is a formal parameter (i.e., it was passed to the current program as a parameter with attribute FUNCTION). In this situation a "BSI Indirect" through the formal parameter name is compiled instead of a "CALL" on that name.

2) A parameter to the call is a subscripted variable. The problem is that the address which has been calculated by the subscripting routine is in XR1 or in a temporary location. This address must be moved from either of these two locations to its proper position in the address list after the CALL statement, i.e., it must be stored "in-line". Whenever this is the case, a new symbol of the form "@OXXX" is created, where OXXX is a hexadecimal number, and assigned as a label to the call statement.

   References to positions in the parameter list can then be made by using the address "@OXXX+1+n" where n is the position in the list; e.g., for the first parameter n is 1. Example:

```
EXECUTE SUB!(1,A,B(I));

        LIBF    $UBSC
        DC      TYPEB
        DC      B
        DC      I+/8000
        STX  1  @0012+1+3
@0012   EQU     *
        CALL    SUB
        DC      ='1'
        DC      A
        DC
```

In this case, the subscript calculation left the variable address in XR1, from which it was stored in-line as the third parameter. Sometimes XR1 must be reused before its value can be stored in-line (all parameters are calculated before any of the subroutine call setup

is performed). In this case, the address is stored in a temporary location, from which it must be stored in-line.

```
    EXECUTE SUB!(1,A(I),B(J);
       LIBF       $UBSC
       DC         TYPEA
       DC         A
       DC         I+/8000
       STX  L1    #TEM-/0001
       LIBF       $UBSC
       DC         TYPEB
       DC         B
       DC         J+/8000
       LD   L     #TEM-/0001
       STO        @0013+1+2
       STX   1    @0013+1+3
@0013  EQU        *
       CALL       SUB
       DC         ='1'
       DC
       DC
```

3) A parameter to the call is a formal parameter in the calling program. For example,

```
    ENTRY SUBA!(A,B);
    EXECUTE SUBB!(A);
```

As stated in Section 4.6.1, formal parameters in a subroutine are implemented as locations containing the addresses of the actual parameters to that subroutine. Thus, in order to pass the actual address to the next subroutine, the CESSL program picks it up from the formal parameter name location and stores it "in-line" to the call. The same created symbol technique as in 2) is used for addressing the parameter list. For the above example,

```
       LD    L   A
       STO       @0014+1+1
@0014  EQU       *
       CALL      SUBB
       DC        0
```

4) A parameter to the call has the FUNCTION attribute. There is a distinctly distasteful problem here caused by the structure of the

TSX operating system--in particular, the loader. The only way a program may signal to the TSX loader that it wishes to refer to an external symbol is by use of a CALL statement (LIBF statements are essentially the same). A CALL statement assembles into "BSI L", a two word instruction. Parameters, however, are supposed to be one word addresses. Thus the following type of code is generated.

```
DECLARE FUNCTION:  YECH;
EXECUTE BARF!(A,YECH);
     B          *+2
     CALL       YECH
     LD         *-2
     STO        @0015+1+2
@0015 CALL      BARF
     DC         A
     DC
```

This succeeds in picking up the second word of the CALL statement (which at run-time is the address of the routine as filled in by the loader), and storing it in-line to the call. Note that the technique for addressing the parameter list is that explained in case 2 above. This code is not very efficient in terms of space if it must be repeated many times, but since the authors do not expect that FUNCTION attribute will enjoy much use, little effort was expended to improve this method. The real culprit is TSX.

Of course, if the parameter is a formal parameter in the calling program, it will be handled as in case 3. If the parameter is an internal function, its address may be included in the address list directly, without the shenanigans.

The reason that CESSL FUNCTION names and FORTRAN EXTERNAL names may not normally be mixed is the different ways these two language processors handle the irksome TSX restriction. FORTRAN actually compiles a CALL to the external subroutine as part of the parameter

list. For example above, FORTRAN would compile

```
CALL    BARF
DC      A
CALL    YECH
```

The FORTRAN subroutine receiving an EXTERNAL name in its parameter list knows that there are two words defining this parameter. It is possible for CESSL programs to receive calls from FORTRAN programs which have EXTERNAL parameters by inserting a dummy parameter into the ENTRY formal parameter list in the position just before the EXTERNAL parameter appears. In this case, the CESSL program will pick up the second word of the FORTRAN compiled CALL XXX statement and use it as the address of the subroutine. The case of CESSL to FORTRAN will not work (since FORTRAN wants to pick up two words).

The reason for the implementation restriction which disallows TSX SKELETON routine names to be passed as FUNCTION names is another TSX baddy. If XXX is the name of a subroutine embedded in the skeleton, a "CALL XXX" statement is fixed by the loader to be "BSI I XXX", i.e., an indirect reference rather than a long reference. Thus the second word of the call is not the address of the subroutine as expected by the techniques described avove.

If the subroutine called is expected to return a non-primitive value, the calling program allocates a block of temporary storage and loads XR1 with the address of that block before performing the call.

4.7 Internal Functions

Internal functions are local subroutines and, as such, the handling for parameters and returned values is exactly the same as for subroutines, as described in Section 4.6, so that this section will describe only the differences.

Calls to internal functions are performed exactly as calls to subroutines, the only exception being that a "BSI L" is compiled instead of a "CALL" statement. The name of the internal function thus contains the return address for the function, and the FUNCTIONRETURN statement compiles a "B I" (branch indirect) through the name, since there may be only one entry point to an internal function.

For functions returning a non-primitive value, index register one is saved in the location immediately preceding the entry point to the function.

Since internal functions may be called from an expression in the main program or from another function, they need their own set of temporary locations. As each internal function is encountered, it is assigned a unique set of temporaries, each of which is an array of storage locations with the name "#TXX", where XX is an octal number between 0 and 63. All temporary references in expression evaluation (Section 4.2) within the function refer, then, to the local array. This local array is allocated after the ENDFUNCTION statement.

To protect against illegally "flowing into" an INTERNALFUNCTION statement or past an ENDFUNCTION statement, the following call is emitted at points where such statements occur.

```
CALL    $FLOW
```

$FLOW is an LOCG system subroutine which prints an error comment of the appropriate type (See Section 3.3 for run-time error messages).

Figure 4.6 has an example definition and invokation of an internal function.

Definition

```
DECLARE INTEGER:  C,D;
INTERNALFUNCTION QQSV!(A,B);
     DECLARE INTEGER:  QQSV,A,B;
     C = (A + B) * (A - B);
     FUNCTIONRETURN C + 4;
ENDFUNCTION;
```

```
     CALL       $FLOW
QQSV EQU        *
     NOP
     LDX   I1   QQSV
     LD     1   /0000
     STO   L    A
     LD     1   /0001
     STO   L    B
     MDX    1   1+/0001
     STX    1   QQSV
     LD    I    A
     A     I    B
     STO   L    #T00-/0001
     LD    I    A
     S     I    B
     M     L    #T00-/0001
     XCH
     STO   L    C
     LD    L    C
     A     L    ='4'
     B     I    QQSV
     CALL       $FLOW
     BSS   E    /0001
#T00 BSS   E    0
```

Invokation

```
D = QQSV!(4,3) + 5
```

```
     BSI   L    QQSV
     DC         ='4'
     DC         ='3'
     A     L    ='5'
     STO   L    D
     .
     .
     .
C    BSS        1
D    BSS        1
A    BSS        1
B    BSS        1
```

Figure 4.6

Example Code for Internal Functions

4.8 Miscellaneous

The CONTINUE statement produces no code. If there was a label on the statement it is defined in the normal fashion.

The EXECUTE statement produces code for the expression, nothing more.

The PAUSE statement produces

```
LIBF    PAUSE
DC      PAR
```

with all the problems of subroutine calls if the parameter is a subscripted variable or formal parameter (See Section 4.6.2).

The ENDPROG statement produces

```
CALL    $FLOW
```

which is a call to an LOCG system subroutine to produce an error message about "illegal flow".

CHAPTER FIVE

SPECIAL TOPICS

This chapter is devoted to topics which do not fit easily elsewhere and which will be of interest mostly to enthusiastic users (although the authors do not admit the existence of any other type).

5.1 Efficient Coding Practices

CESSL does not produce highly optimized code. About the only optimization performed is on the use of registers over a single statement. However, there are a few things the user can do to increase the run-time efficiency of his program. We will not mention the normal programming good sense which all programmers should use because, as previously stated, this is not a tutorial but a reference manual. The points below refer to a particular implementation of a particular compiler and should be used as such.

1. The use of switches (i.e. BOOLEAN variables) is encouraged. A switch in an IF or ORIF statement produces very fast code (relative to a comparison). Thus, if the same condition needs to be checked mere than once, a speed advantage may be obtained by first calculating the comparison into a BOOLEAN variable and then using the BOOLEAN variable in the conditionals.
2. Assignment of a value to more than one variable is the most efficiently done by an embedded assignment statement. That is,

```
A = (B = C);
```

   is more efficient than

```
B = C;  A = B;
```

3. Comparisons are most efficiently performed when one of the comparands

is an INTEGER zero (0).

4. The user should factor out common expressions in statements, even subscript expressions.
5. An expression involving all constants will be completely calculated every time it is encountered in the program so that it is advantageous for the user to do the precalculation.

5.2 Dynamic Data Types

It is possible to change the description of a data type at run-time. Every data type has a descriptor (dope vector) which is used at run-time by the subscription routine. Change the dope vector and the data type has been "changed". Naturally this capability should be used with great care, and users should study Section 4.3 (Subscription) before attempting it. It is easiest to do with ARRAYs--indeed it is most difficult with BLOCKS--and most of the motivation for doing it comes from the use of ARRAYs, so that only they will be mentioned below.

Recapitulating the important points of Section 4.3, the dope vector for an ARRAY consists of a label for the data type name and three words, the first of which gives the number of components in the ARRAY (i.e. the maximum subscript value), and the third of which gives the number of *words* which each component occupies. These words can be readily accessed by use of the EQU statement.

For example, the most common use of this technique concerns the passing of "arrays" and "matrices" to subroutines. Often the subroutine can be written to be perfectly general except for the size of the array or matrix passed. Every parameter to the subroutine must have a data type defined for it and

then declared to be of that type. Requiring the binding time for the definition of the type to be at compile time inplies two things for arrays. First, the maximum size of the array expected must be decided ahead of time. Second, the subscript size checking is defeated when smaller-than-the-maximum-size arrays are passed. For matrices, it is impossible to provide other than the correct size of the rows of the matrix. These restrictions can be defeated by passing the SIZE of the data types associated as parameters to the subroutine using the technique illustrated in the following example:

```
ENTRY SUB!(IARRAY,IN,IMATRIX,N,M);
DEFINE IA ARRAY INTEGER SIZE 1:  IARRAY;
EQU (SIZEIA,IA);
DEFINE ROW ARRAY INTEGER SIZE 1;
DEFINE MATRX ARRAY ROW SIZE 1:  IMATRIX;
EQU (SIZEROW,ROW) (NUMROWS,MATRX) (SIZEMATRXROW,"MATRX+2");
DECLARE INTEGER:  SIZEIA, SIZEROW, NUMROWS, SIZEMATRXROW;


SIZEIA = IN;
SIZEROW = N;
NUMROWS = M;
SIZEMATRIXROW = N;
   o
   o
   o
```

IARRAY is a simple INTEGER array whose size is given by the parameter IN. By equivalencing the INTEGER variable SIZEIA with the first word of the dope vector describing the data type of IARRAY the assignment "SIZEIA = IN;" declares

IARRAY to be an INTEGER array of size IN. The original definition of IA as an ARRAY SIZE 1 merely provides a name and number which can be changed later.

A similar technique works with matrices. IMATRIX is an INTEGER matrix declared in row major order (i.e. it is stored in memory with the second subscript varying fastest). Since it is an ARRAY of ARRAYs and since the size of both arrays is variable, two dope vectors need changing as regards to the number of components in the ARRAY. Note however, that the dope vector for "MATRX" needs altering in the third position, which describes the number of words taken by each component. In this case, this becomes the number of words in a row, which is the same as the size of the row. The reader should understand this example completely before attempting any such action.

An important point to remember is that REAL variables require two words of storage, twice as much as INTEGER variables. Thus, if the matrix above was made up of REAL variables, the last statement would have to be changed to

```
SIZEMATRIXROW = 2*N;
```

Although the most common use of this chicanery is for parameter type definition it is also permissable to use the technique on data types defined in the same program as they are used to allocate storage. For example, it is possible to redefine a four by five INTEGER ARRAY to be a two by ten array. The restrictions which remain are three. First, it is not possible to provide more storage for a variable than was allocated as a result of the original definition of the data type. For example, a four by five cannot be redefined to be a three by eight. Second, the original hierarchical structure must be maintained; i.e. a two-dimensional array cannot be changed into one or three dimensional array. (If it is desired to refer to an array alternately

as one- and two-dimensional, two different data types should be defined, assigned to two variables, and the two variables EQUivalenced (see Section 2.4.1.6).) Thirdly, the variable in question should not appear with a set of subscripts which are all constants except all one's), since the address resulting from such a subscription is calculated at compile-time from the original definition of the data type.

5.3 TSX Calls

The TSX system has specially defined subroutines to perform various arcane functions. These routines are designed to be called from FORTRAN. Most are thus available to CESSL users directly. The one source of difficulty arises with subroutines which require parameters with a FORTRAN EXTERNAL specification. As explained in Sections 2.4.3.2 and 4.6.2, FORTRAN EXTERNAL and CESSL FUNCTION parameters do not mix. The same effect can be achieved, however, by the use of the following methods. The important point to remember is that a FORTRAN EXTERNAL parameter is compiled as a CALL statement.

The calls

```
CALL CHAIN(NAME)
CALL SPECL(NAME)
```

can be performed from CESSL by the following statements:

```
EXECUTE CHAIN!;  EXECUTE NAME!;
EXECUTE SPECL!;  EXECUTE NAME!;
```

Although NAME is the name of a coreload, the EXECUTE statement compiles the code the loader expects. The two EXECUTE statements should not be separated; that is, they should appear as successive statements in the program.

Other TSX calls require additional parameters. For example:

```
CALL QUEUE(NAME, P, E)
```

All these can be written in CESSL by two EXECUTE statements, the first of which calls the desired routine and the second of which refers to the name of the coreload or subroutine, along with the parameters, as for example:

```
EXECUTE QUEUE!; EXECUTE NAME!(P,E);
```

Calls which fall into this class are: QUEUE, QIFON, UNQ, TIMER, and COUNT.

5.4 In-line Code

The DATA statement in CESSL differs from similar statements in other languages in two respects: the data is compiled in-line rather than apart from the generated code; and variables as well as constants may appear in the list of values assigned to the name. The latter results in the compilation of the address of the variable. The two features together allow numerous applications, of which we will name only two. Users whould be familiar with Assembly language and the contents of Chapter Four before attempting to use these special features.

In-line Code

There are some instructions in the IBM 1800 CPU which are not normally available through CESSL: XIO (Input/Output control), STS and LDS (Status control), LDD and STD (Load and Store Double), etc. Programs requiring such complete control of the machine are usually best written in Assembly language. Occasionally, however, it is convenient to use such an instruction in a CESSL program without calling an Assembly language subroutine. Since such use is infrequent, CESSL has no special language constructs for it. The in-line DATA statement allows the user to compile the exact instructions he needs, if he is willing to specify them in hexadecimal.

For example, to execute the two word IOCC (Input/Output Control Command)

at location 46 (absolute), one may write:

```
SUBSTITUTE (XIOL,?0C00);
DEFINE I2 ARRAY INTEGER SIZE2:  DOIT;
...
DATA DOIT XIOL,46;
```

The compiled code would resemble the Assembly language statement

```
XIO L 46
```

Alternately, if the variable FDL contains the IOCC to start the Framl Dakting Lifter, one may write:

```
DATA DOIT XIOL,FDL;
```

In a similar manner, every machine instruction and series of instructions may be compiled into a CESSL program.

In the example above, we had to define a data type, I2, and assign it to a name, DOIT, in order to specify the code produced. Usually the name and the data type definition is unimportant, we just need a handle. For this reason, the special atom $$$ may be used as the name in DATA statements where the name is immaterial and the sole purpose for the statement is to put out some code. Rewriting the example, we have:

```
SUBSTITUTE (XIOL,?0C00);
...
DATA $$$ XIOL,46;
```

Since $$$ is not defined as a label in the generated code it may be reused in many such DATA statements for the same purpose. Retention of the variable name and type, on the other hand, allows the user to create an instruction and store it in-line, if he happens to be addicted to von Newmann machines.

Special Parameter Lists

The second application of DATA is a special case of the first, in-line code, but it important enough to be separately emphasized. Often, subroutines written in Assembly language are not callable from higher level languages

like CESSL and FORTRAN because they use non-standard calling sequences. One of the most frequent examples of this is in the use of a value as a parameter rather than the CESSL/FORTRAN standard of using the address of a variable containing the value as a parameter. For example,

```
EXECUTE SUB!;
DATA $$$ 4,XYZ;
```

produces code equivalent to the Assembly language:

```
CALL    SUB
DC      4
DC      XYZ
```

Again, use of a name in the DATA statement allows dynamic alteration of a parameter list.

Another non-standard kind of call is the passing of parameters in the general registers. Use of in-line instructions makes this possible. For example, if the user were writing a subroutine which expected its only parameter to be in the accumulator on entry, he could put the accumulator into the variable PAR as in the following:

```
SUBSTITUTE (STOL,?D400);
ENTRY SUBB!;
DATA $$$ STOL,PAR;
```

The user should be very familiar with Section 4.6.1, CESSL Compiled Subroutines, especially the method of picking up actual parameters, before he gets too fancy in this regard.

Appendix

EBCDIC Characters

Character	Hex	Dec	Format	Character	Hex	Dec	Format
Tab	5	5	-305	A	C1	193	-493
EOP*	6	6	-306	B	C2	194	-494
Shift-to-Black	14	20	-320	C	C3	195	-495
Carriage Return	15	21	-321	D	C4	196	-496
Back Space	16	22	-322	E	C5	197	-497
Line Feed	25	37	-337	F	C6	198	-498
EOF**	26	38	-338	G	C7	199	-499
Shift-to-Red	35	53	-353	H	C8	200	-500
Space	40	64	-364	I	C9	201	-501
¢	4A	74	-374	J	D1	209	-509
. (period)	4B	75	-375	K	D2	210	-510
<	4C	76	-376	L	D3	211	-511
(	4D	77	-377	M	D4	212	-512
+	4E	78	-378	N	D5	213	-513
| (logical OR)	4F	79	-379	O	D6	214	-514
&	50	80	-380	P	D7	215	-515
!	5A	90	-390	Q	D8	216	-516
$	5B	91	-391	R	D9	217	-517
*	5C	92	-392	S	E2	226	-526
)	5D	93	-393	T	E3	227	-527
;	5E	94	-394	U	E4	228	-528
¬ (logical NOT)	5F	95	-395	V	E5	229	-529
- (minus)	60	96	-396	W	E6	230	-530
/	61	97	-397	X	E7	231	-531
, (comma)	6B	107	-407	Y	E8	232	-532
%	6C	108	-408	Z	E9	233	-533
_ (underscore)	6D	109	-409	0	F0	240	-540
>	6E	110	-410	1	F1	241	-541
?	6F	111	-411	2	F2	242	-542
:	7A	122	-422	3	F3	243	-543
#	7B	123	-423	4	F4	244	-544
@	7C	124	-424	5	F5	245	-545
' (prime)	7D	125	-425	6	F6	246	-546
=	7E	126	-426	7	F7	247	-547
"	7F	127	-427	8	F8	248	-548
				9	F9	249	-549

*Used in LOCG system as End-of-Page character.
**Used in LOCG system as End-of-File character.

BIBLIOGRAPHY

1. Brender, R. F., A Programming System for the Simulation of Cellular Spaces, Technical Report 25, Concomp Project, University of Michigan, Ann Arbor, January, 1970.

2. Brender, R. F., D. R. Frantz, J. L. Foy, Jr., and T. W. Schunior, Specialized System Software for Interacting DEC PDP-7 and IBM 1800, Technical Report 11, Concomp Project, University of Michigan, Ann Arbor, December, 1968.

3. Frantz, D. R., and R. F. Brender, The Cellular Space Simulation System Manual, Forthcoming Technical Report, Logic of Computers Group, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, about December, 1971.

4. International Business Machines Corp. IBM 1800 Assembler Language, IBM Publication C26-5882.

5 ___. IBM 1800 Functional Characteristics, IBM Publication A26-5918.

6. ___. IBM 1800 Time-Sharing Executive System Concepts and Techniques, IBM Publication C26-3703.

7. ___. IBM 1800 Subroutine Library, IBM Publication C26-5880.

8. ___. IBM 1800 Time-Sharing Executive System Operating Procedures, IBM Publication C26-3754.

9. Michigan Algorithm Decoder (MAD) Manual, University of Michigan Press, Ann Arbor, August, 1966.

E

F

G