Science Applications Incorporated

(NASA-CR-144236) SSL: A SOFTWARE SPECIFICATION LANGUAGE (Science Applications, Inc., Huntsville, Ala.) 74 p CSCL 09B

N76-21931

Unclas

G3/61 21580

TABLE OF CONTENTS

1. THE LANGUAGE.................................... 1-1

1.1 INTRODUCTION................................ 1-1

1.1.1 Need for SSL................. 1-1
1.1.2 Unique Features of SSL....... 1-2
1.1.3 Background................... 1-3

1.2 THE GRAMMAR................................ 1-4

1.2.1 Metalanguage Description..... 1-4
1.2.2 Overview of SSL Grammar...... 1-5
1.2.3 Basic Vocabulary............. 1-7
1.2.4 Basic Language Elements...... 1-9
1.2.5 Requirement Declaration...... 1-13
1.2.6 Data Type and Variable Declaration................... 1-16
1.2.7 Constant Declaration......... 1-30
1.2.8 Data References.............. 1-31
1.2.9 Expressions and Assertions... 1-33
1.2.10 Module Descriptions......... 1-43
1.2.11 Subsystem Descriptions...... 1-54

1.3 EXAMPLE.................................... 1-57

2. INDEX... 2-1

3. REFERENCES...................................... 3-1

LIST OF ILLUSTRATIONS

1-1 Syntactical Overview of an SSL Specification .. 1-6

1-2 Module Structure Chart for Example 1-59

1-3 SSL Description for Example 1-61

LIST OF TABLES

1-1 SSL Special Symbols 1-8

1-2 SSL Reserved Words 1-10

1-3 Arithmetic Operations 1-36

1-4 Operation Hierarchy 1-39

1-5 Logical Operator Truth Table 1-40

1-6 Boolean and Relational Operations 1-41

1-7 Module Descriptions for Example 1-60

SAI-77-537-HU

SSL - A SOFTWARE SPECIFICATION LANGUAGE

Contract No. NAS8-31379

Prepared For:

George C. Marshall Space Flight Center
Data Systems Laboratory
Huntsville, Al. 35812

Prepared By:

Sandra L. Austin
Billy P. Buckles
J. Patrick Ryan

26 January 1976

SCIENCE APPLICATIONS, INC.
2109 W. Clinton Avenue
Suite 800
Huntsville, Alabama 35805
(205) 533-5900

1. THE LANGUAGE

1.1 INTRODUCTION

SSL (Software Specification Language) is a new formalism for the definition of specifications for software systems. The language provides a linear format for the representation of the information normally displayed in a two-dimensional module inter-dependency diagram. In comparing SSL to FORTRAN or ALGOL, one finds the comparison to be largely complementary to the algorithmic (procedural) languages. SSL is capable of representing explicitly module interconnections and global data flow, information which is deeply imbedded in the algorithmic languages. On the other hand, SSL is not designed to depict the control flow within modules. We refer to the SSL level of software design which explicitly depicts inter-module data flow as a functional specification.

We wish to express our appreciation to Mr. Bobby Hodges of Data System Labortory, George C. Marshall Space Flight Center for his guidance and support in the performance of this task.

1.1.1 <u>Need for SSL</u>

The current state of the art in software development permits insufficient formal evaluation prior to implementation. Such questions as:

- Are all requirements fulfilled?
- Have all software elements been defined?
- Are the element interconnections consistent?

cannot be answered in a manner that is independent of the designer's opinion. The intent of SSL is to formalize, through a language, the statement of the functional specification for a software system. Given this formal statement expressed in SSL and a translator for the SSL language, an independent evaluation of the software may begin much earlier in the development cycle.

In addition to evaluation, other aspects of SSL can aid both the designer and implementer. Several things that are characteristically omitted or inadequately performed during early design but required in SSL are:

- A complete and consistent statement of the software requirement
- Unambiguous communication of software organization to the detailed designer
- Enumeration of intraprogram consistency checks (assertions) useful during checkout.

A translator also provides tables and summaries for the final software documentation and a software element cross reference file. The latter could be used to statically verify the fidelity of the final code to original specifications.

1.1.2 Unique Features of SSL

The major contribution of SSL is the formal approach it brings to a phase of software development previously relegated to heuristic techniques as discussed above. Within this framework, there are several unique technical features possessed by SSL. First, the projection of a specialized form of software requirements onto the objects being defined establishes a rationale for the software structure not present in other methodologies. These requirements are an important aspect of consistency checking when evaluating a specific functional design. Second, the incorporation of levels of abstractions directly in a design methodology is a step forward in software engineering. Lastly, an automated SSL translator is being designed that is one of several interlocking software design and evaluation tools collectively called Software Specification and Evaluation System (SSES). SSES includes a static code analyzer, a dynamic code analyzer, and a test case analyzer. The specific capability that SSL brings SSES is the ability to test and evaluate software design early in the development cycle.

SSL also incorporates a flexible data abstraction capability and places emphasis on assertions as a means of describing the dynamic behavior of the software being designed. Although neither of these is unique, they are relatively new concepts in the field of computer science.

1.1.3 Background

In evaluating a new software system, particularly a programming language, it is important to trace the historical developments to which it relates and upon which it is based. The MIL (Module Interconnection Language) system [1] was a principal contributor to the concepts of data creation and data availability restrictions among modules within SSL. Guidelines imposed for the partition of programs into subsystems are derived from the principles embodied in the concept of levels of abstraction [2]. Module descriptions in SSL are a linearlized form of the information available in the two-dimensional diagrams referred to as structure charts[3]. The data description capability is largely the same as that of PASCAL [4]. The syntax for expressions is derived from, but not identical to, that of ALGOL 60 [5]. Assertions in SSL have the form and appearance of those in the language NUCLEUS [6].

1.2 THE GRAMMAR

The material in this section is arranged in the form of a reference guide to the language, and not tutorially in the manner of a user's manual. To aid the reader, a cross reference index is provided in the last section.

1.2.1 Metalanguage Description

For the purposes of automatic translation and unambiguous communication, it is desirable to express SSL via a formal grammar. The vehicle selected for this purpose is the Backus-Naur-Form (BNF) metalanguage [5]. BNF has the advantages of being well known and compact in representation. In addition, most formal methodologies for analyzing grammars are based upon BNF representation.

Any nontrivial language contains an infinite number of legal sentences. Each sentence, in turn, is composed of the concatenation of strings; strings are composed of characters. A grammar uses strings as operands and combines them under the operation of concatenation to finitely depict, all legal sentences. The way in which this is done in BNF can best be interpreted via an example. Consider the following production:

```
<ab> :: = a|b| <ab>  a
```

Sequences of characters enclosed within the brackets < >represent metalinquistic variables called nonterminal symbols. The marks "::=" and "|" are metalinquistic connectives meaning "is composed of" and "or" respectively. Any string not a nonterminal or connective denotes itself and is called a terminal symbol. Juxtaposition of symbols between connectives in a formula, such as the example, signifies that the symbols must be in the exact order denoted. The above production indicates that <ab> may have the values:

- a
- b
- a, aa, aaa, ...
- b, ba, baa, ...

In BNF, the null string is designated by <empty> :: =

SSL is represented as a context-free grammar which means:

- There exist a finite number of productions of the type of the above example.
- The left part of each production (i.e., left of ::=) consists of a single nonterminal symbol.
- There exists a unique nonterminal symbol (called the distinguished symbol) which is in the right part of no production except its own.

1.2.2 Overview of SSL Grammar

Prior to examining the detailed structure of SSL components, it will be useful to identify the overall structure of a software specification expressed in SSL. Figure 1-1 depicts the sequencing of the syntactical items used to describe an SSL specification.

A specification consists of one or more subsystems, each but the first having a name. The first subsystem is referred to as the "main" subsystem and each subsystem is composed of a preamble and one or more module descriptions. The preamble defines the local environment for the subsystem

ORIGINAL PAGE IS OF POOR QUALITY

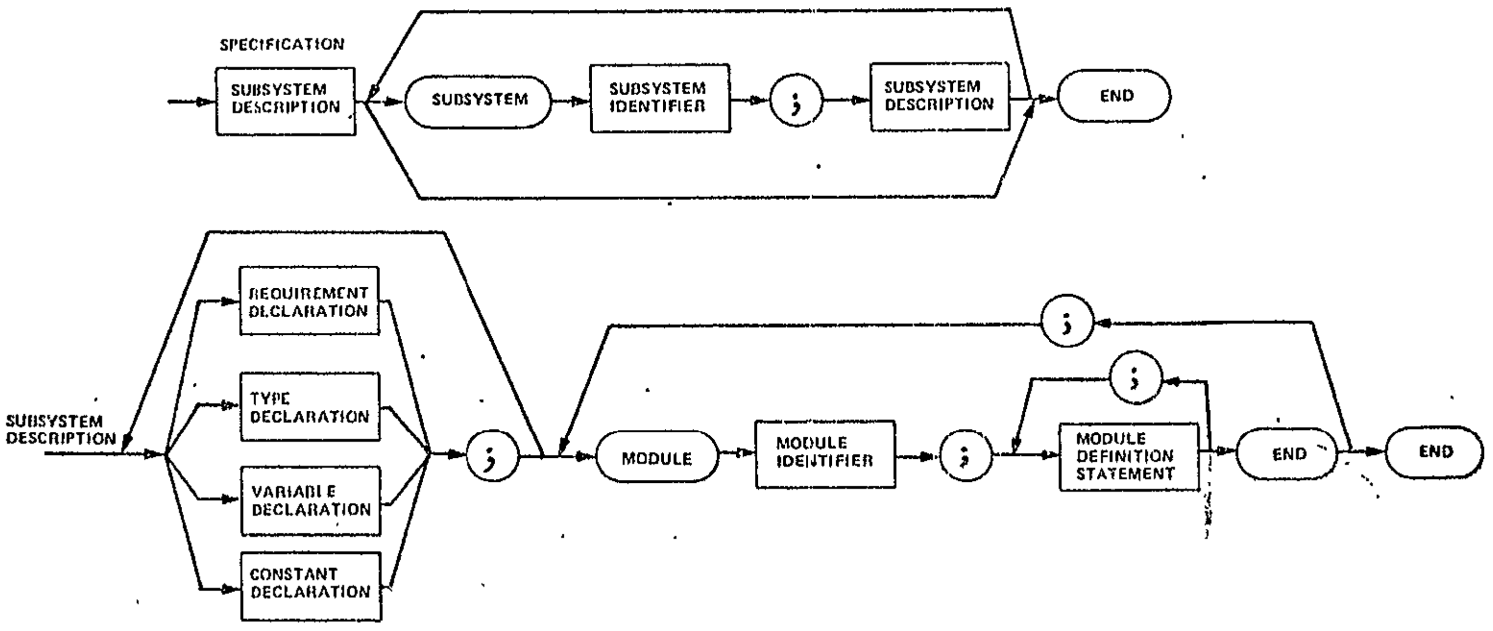


Figure 1-1 Syntactical Overview of an SSL Specification

(constants, requirements, data formats, etc.) and the module descriptions indicate operational aspects of program units (program units are subprograms, procedures, etc.).

In the following subsections, the detailed syntactical descriptions will be presented. To facilitate cross referencing, Section 2 contains an index of nonterminal symbols.

1.2.3 Basic Vocabulary

The basic vocabulary of SSL consists of special symbols, letters, digits, and reserved words. Each special symbol (Table 1-1) is primarily a single character except where limited computer character fonts require the concatenation of two characters. Where a special symbol consists of more than one character, it must be written without an intervening blank. Subsequently, special symbols other than ".", "@", and "_" will be referred to as delimeters. Each character in Table 1-1 is available within the ANSI standard codes [7] for ASCII-8, EBCDIC-8, and HOLLERITH-256. Substitutions may be necessary if an SSL translator is implemented in an environment not conforming to the standard character codes.

Letters and digits do not have individual meanings but are used to construct identifiers, numbers, and reserved words. The following basic productions enumerate these elements of the vocabulary:

```
<letter> ::= a|b| ... |z

<digit>  ::= 0|1| ... |9
```

TABLE 1-1 SSL SPECIAL SYMBOLS

+	[
-	]
*	..
/	**
=	<=
,	>=
;	¬=
:	/*
<	*/
>	.
(	@
)	__
¬	

Reserved words (Table 1-2) are composed entirely of sequences of letters. In this document, they are normally underlined. A reserved word may not contain imbedded blanks and must always be followed by a blank or a delimeter.

The construct

```
/*  any sequence of symbols not containing "*/" */
```

may be inserted between any two identifiers, numbers, delimeters, or reserved words. It is called a comment any may be removed from the program text without altering its meaning.

1.2.4 Basic Language Elements

1.2.4.1 Identifiers

Syntax

```
<identifier> ::= <letter>|<identifier> <letter>
                 |<identifier> <digit>|<identifier> _
```

Examples

Legal	Illegal
a	5ad
b27	sr$p
cr_14dr	

TABLE 1-2 SSL RESERVED WORDS

Access
Accesses
Analog
And
Array
Assume
Assumes
Boolean
Case
Char
Conditionally
Constant
Constants
Constraint
Constraints
Create
Creates
Digital
Doubleprecision
End
Entry
Equ
Execute
Executes
False
File
For
Forall
From
Fulfil
Fulfills
Global
Implies
In
Input
Inputs
Integer
Iteratively
Modify
Modifies
Module
Of
Or
Output
Outputs
Real
Receive
Receives
Record
Requirement
Requirements
Satisfies
Satisfy
Set
Subjectto
Subsystem
Text
To
Transduction
Transductions
Transmit
Transmits
True
Type
Types
Use
Uses
Using
Variable
Variables

Semantics

Identifiers must begin with an alphabetic character and contain only letters, digits, and the "_" symbol. The latter is known as the break character. Identifiers have no inherent meaning, but serve as identification for variables, modules, subsystems, and other elements of a software specification.

Identifiers may be of arbitrary length but must be unique within the first twelve characters. No identifier may be equivalent to the first twelve characters of a reserved word. The same identifier may not be used to denote two different quantities within a subsystem with the exception of field names in different records.

1.2.4.2 Numbers

Syntax

```
< unsigned interger>  ::=<digit> | <unsigned integer>
                           <digit>

<sign  > :: =  + |  -

<exponent  part > :: =  e <unsigned integer >
                       |e <sign> <unsigned integer>
                       |d <unsigned integer>
                       |d <sign> <unsigned integer>
<  decimal number > :: = <unsigned integer>
                       |<unsigned integer> .
                        <unsigned integer>
<  unsigned number >:: = <decimal number >
                       |<decimal number> <exponent
                         part>
```

Examples

Legal	Illegal
57	3,746
14d10	XII
3.7 e-5	e+7
0.2	.14

Semantics

Decimal numbers have their conventional arithmetic meaning. The exponent is a scale factor expressed as an integal power of 10. A number expressed with neither a scale factor nor a decimal fraction is assumed to be of type integer. A number which uses the "d" form for the exponent part is assumed to be double precision. Otherwise, the number is assumed to be type real. Note that if a number contains a decimal point, at least one digit must precede and succeed the point.

1.2.4.3 Logical Values

Syntax

<logical value> ::= true |false

Semantics

Logical values have their conventional meaning and may be defined by describing their combination under the operations "union" and "intersection". The union of the logical value true with any other logical value always yields the result true. The intersection of the logical value false with any other logical value always yields the result false.

1.2.5 Requirement Declaration

The several parts of the requirement declaration are used to identify the data flow between the software package being described and other parts of the total system. In addition, they identify processing steps (called transductions) and restrictions (called constraints) which are attached to both modules and variables.

Syntax

```
<requirement declaration> ::=  <requirement or
                               requirements>
             <requirement statement group> end

<requirement or requirements> ::=   requirement
             |requirements

<requirement statement group> ::= <requirement
             statement part>
             |<requirement statement group> ;
              <requirement statement part>

<requirement statement part> ::= <input part>
             |<output part>|<transduction part>
             |<constraint part>
```

1.2.5.1 Input and Output Parts

An input is a system level input (or stimulus) which a software package receives from an external source. An output is a system level response which has a purpose beyond the immediate concern of the software package being described.

Syntax

```
<input part> :: = <input or inputs> <entire
                   variable list>

<output part> :: = <output or outputs> <entire variable
                    list>

<input or inputs> :: = input|inputs
<output or outputs>:: = output|outputs
```

Examples

- input   state_vector ;
- inputs mass, velocity, distance ;
- output concordance_list ;

Semantics

A variable may be in both an input and an output list. A variable in an output list not used within the subsystem other than in the module in which it is initialized is not required to have a requirement transduction attribute. The structure of all variables in input and output lists must be described within the variable statements of the subsystem preamble. Each subsystem preamble must have a requirement declaration with an output part.

1.2.5.2 Transduction Parts

Transductions are identifiers representing processing steps. They are derived by first writing a high level pseudo-program to "transduce" the input variables into the output variables and then extracting and listing the major verbs of the program. Just as the processing steps of the pseudo-program may be nested, the transductions may likewise be nested. Ideally, for each subsystem there should be from three to seven transductions that are not nested within any others.

Syntax

```
< transduction part> :: = <transduction or
                                 transductions >

                 <transduction clause>
                |<transduction part> ;  <transduction
                     clause >

< transduction or transductions >:: = transduction
                | transductions

< transduction clause> :: = <transduction list>
                |<transduction list>  in <transduction
                 list >

< transduction list> :: =  <transduction identifier>
                |<transduction list >,  <transduction
                 identifier>

< transduction identifier >:: =  <identifier>
```

Examples

- transduction sum_expense, sub_deduct in tax_compute;
  write_paycheck;

- transductions save_options; read_card in parse;

Semantics

Within a transduction clause, each processing step represented by a transduction identifier to the left of in must be a substep of the processing steps listed on the right of in. Each transduction identifier represents a unique processing step, but may be reused to show different substep relationships. Substep relationships must be consistent, i.e., the complete set of substep relationships partially order the transduction identifiers.

1.2.5.3 Constraint Parts

Syntax

```
< constraint part>  ::= <constraint or constraints>
                        < constraint list>

<constraint or constraints >::=  constraint
                        | constraints

< constraint list > ::= <constraint identifier >
                        |<constraint list >,  <constraint
                         identifier>

<constraint identifier> ::= <identifier>
```

Examples

- constraint   carpool_size ;
- constraints   max_targets, minimum_distance ;

Semantics

Each constraint identifier defined must be attached as an attribute to some module in the subsystem.

1.2.6 Data Type and Variable Declarations

Explicit description of data and the ability to define and use new data types is one of the greatest assets of SSL. A new data type may be described directly as part of a variable declaration, or described independently for subsequent use.

Syntax

```
< type declaration> ::= <type or types>
                        <type definition>
            |< type declaration> ;   <type definition>

< type or types> ::=  type | types| global type
            |global types
```

```
<type definition> ::= <identifier> = <type>

<type >::= <simple type> |<structured type>

          |<pointer type>

<variable declaration> ::= <variable or variables>
           <variable definition>
          | <variable declaration> ; <variable definition>

<variable or variables> ::=  variable | variables

<variable definition> ::= <identifier list> :< type>

          |<identifier list> : <type> ; <for clause>
          |<identifier list> : <type> ; <subjectto clause>
          |<identifier list> : <type> ; <for clause>;

               <subjectto clause>

<for clause> ::=  for  <transduction list>
<subjectto clause> ::=  subjectto <assertion list>
<assertion list> ::= <assertion>

          |<assertion list> ; <assertion>

<identifier list> ::= <identifier> |<identifier list> ,
           <identifier>
```

Semantics

A type declaration list is used to define new data types. Each type is named and may be referenced by the identifier to the left of "=" in the <type definition> production. The normal scope of a type identifier is the subsystem in which it is defined. However, the scope of a global type is the entire SSL program. Global types may be defined only in the main subsystem.

A data type need not be named if it is defined intrinsic to the variable declaration. Both type and variable declarations may use data types defined and named elsewhere. Examples of both are given in the following subsections.

The <for clause> of the variable declaration is used to attach requirement attributes. Requirement attributes limit the availability of variables within the modules of the subsystem. All variable declarations must contain a for clause with the exception of output variables identified in the requirement statement.

The <subjectto clause> identifies the global assertions associated with the variables being declared. A global assertion is one that must be true upon exit from the module creating the variable, and true on both entry and exit of modules using the variable.

1.2.6.1 Simple Types

Simple types are data types for which the designer, using SSL, need not define the internal structure or the internal structure has previously been defined and named.

<u>Syntax</u>

```
<simple type> ::= <basic type> |<scalar type>
          |<subrange type> | <type identifier>

<type identifier> ::= <identifier>
```

<u>Semantics</u>

A type identifier must previously have been used to the left of an "=" in a type definition.

1.2.6.1.1 Basic Types

The basic data types are those which are implicity defined by the SSL language.

```
<basic type> ::= integer| real| boolean
              |doubleprecision |char| analog| text
```

Examples

```
• variables    I, J, K:  integer; for  count_people ;

• variables    height:  real;
                 for  record_status;
                 subjectto   height  >0.0 ;
                             height <= 10.0 ;

               employed:  boolean ;
                 for  record_status
```

Semantics

The types integer, real, boolean, and doubleprecision have the conventional meaning. The type char indicates a single unit of hollerith information. Type text indicates hollerith data with unspecified length. Since the length of a text item varies, it may not be combined with other variables in forming structured data types. The type analog designates a data item which contains analog signal information. Like the text type, it may not be combined with other variables to form structured types.

1.2.6.1.2 Scalar Types

Scalar types are used to designate a finite number of disjoint states which a variable may represent. In conventional programming languages, it is customary to declare the variable of type integer and assign it only the cardinal numbers 1, 2, ..., n where each value represents one state of the several possible.

Syntax

```
< scalar type> ::= ( <identifier list> )
```

Examples

```
• type marital_status = (single, married, divorced);
  variable ms: marital_status; for emp_record;

• variable color: (Red, blue, yellow, green) ;
```

Semantics

Conceptually, the elements of scalar types are ordered regardless of whether or not the underlying set of states is ordered. The order is always the same as that of the identifiers in the identifier list. This enables a designer to use relational tests (< ,>, etc.) in assertions involving scalar type variables.

1.2.6.1.3 Subrange Types

Subrange types are used to designate a subset of integers or scalars which a data item may assume.

Syntax

```
< subrange type> ::= <constant> .. <constant>
< constant> ::= <unsigned  integer>
              |<sign> <unsigned integer>
              |<constant identifier>
```

```
            |<sign> <constant identifier>
            |<logical value>
        constant identifier> ::= <identifier>
```

Examples

```
variables   weight: 10..350; for  ins_compute;
            dependents:  0..15; for  tax_compute;

type color =  (purple, blue, red, yellow, green, black);
variable  primary_color:  blue..green;
```

Semantics

A subrange simply indicates the least and largest constant values an item may assume. The lower bound (left-most constant in the production) must be less than the upper bound. A subrange with bounds expressed in types other than integer or scalar is not permitted.

Constant identifiers may arise from two contexts. The first is the appearance of an identifier to the left of "=" in a constant declaration. The second (illustrated by the above example) is thè appearance of an idenfifier as a scalar element. Constant identifiers arising from the second context may not be preceeded by a unary sign.

1.2.6.2 Structured Types

A structured type is a data type composed of more elementary data types.

Syntax

```
< structured type> ::= <array type>|<record type>
        |<digital type>| <set type>
        |<sequence type>
```

Semantics

An SSL structured data type is used to indicate the general form and content of a data structure, not precise implementation word and storage formats. In SSL, the following definitions are used:

Array -	A fixed number of data items, all of the same type and length and accessed by computed index.
Record -	A fixed number of data items, each of fixed length, and each equally accessible.
Digital-	A record having additional restrictions which are discussed in a subsequent subsection.
Set -	An element of the powerset of a finite number of basic elements.
Sequence or File -	A variable number of data items, all of the same type and length; however, each element is not equally accessible at all times.

Stronger connotations (such as elements of an array are sequentially stored) are not implied by the semantics of SSL.

1.2.6.2.1 Arrays

An array is a fixed number of data elements, each of the same type and length and each equally accessible. Elements of an array are ordered and each element is accessed by a cardinal number called its index.

<u>Syntax</u>

```
< array type >::=  array [<index list>] of
                     <component type>

< index list> ::=  <index type> |<index list> ,
                     <index type>

< index type >::=  <simple type>

< component type> ::=  <type>
```

<u>Examples</u>

```
● variable  matrix: array  [1..10, 0..20] of real;
                    for  ta ;

● type people = (adams, buckles, jones, smith);
  variable  employee:  array [people]  of  1..50 ;
                    for  ta;
```

Semantics

Index types must have a finite range and be ordered. This requirement eliminates index type of integer, real, and double-precision. However, subranges of integers are permitted. For the purpose of ordering, false <true for boolean type indices.

1.2.6.2.2 Records

A record is a structure containing a number of components called fields. Fields are not constrained to be of identical type but must be of fixed length. A single record type is permitted to have variants.

Syntax

```
<record type> ::= record <field list> end
<field list> ::= <fixed part> | <fixed part> ;
        <variant part> |<variant part>
<fixed part> ::= <record section> |<fixed part> ;
        <record section >
<record section> ::= <field identifier list> : <type>
<field identifier list>  ::= <field identifier>
       |< field identifier list> , <field identifier>
<variant part> ::= case <tag field> <type identifier>
        of <variant list>
<variant list> ::= <variant> | <variant list> ;
        <variant>
<tag field> ::= <field identifier>  :
<field identifier> ::= <identifier>
<variant> ::= <case label list> : (<field list> )
        |<case label list> :( )
<case label list> ::= <case label>  |<case label list> ,
         < case label>
<case label> ::= < constant>
```

Examples

```
● Type employee = Record

        Number:Integer;
        Salary:Real;
        Name:Array [1..24] of char

     End;

○ Variable machine_part:Record

        Part_No, Order_Quantity:Integer;
        Weight:Real

    End;

    for customer_billing;

● Type complex = Record  real_part, imag_part:real End;

● Type farm =(peaches, cotton, soybeans);
  Type  land_use = Record

        Owner_Name:Array [1..12] of char;
        Plot_No:Integer;
        Case Crop:Farm of

              peaches:(tree_count:Integer);
              cotton, soybeans:(plant_date:Integer
                  herbicide, insecticide:boolean)

    End;

● Variable sizes:Array [1..10] of Record

              Height:Integer;
              Weight:Real

        End;
```

```
for  health_file_update;
subjectto height >0; height <120;
          weight >0.0;  weight  <500.0 ;
```

Semantics

Fields may not be of basic types text or analog. A record may be a component of another record, but a digital type may not. The scope of a field identifier is the smallest record in which it is defined. Field identifiers with disjoint scopes may be reused. Access of a component is always by the field identifier and never by a computed value.

The type associated with the tag field of a variant must contain only a finite number of elements. This limits it to boolean, subrange, and scalar. All elements of the type must appear in some case label list of the variant. If the field list for case label L is empty, the form is:

L : ( )

A record may contain only one variant part and it must succeed the fixed part. However, a variant may contain variants. That is, it is possible to have nested variants. All field names of the same record must be unique even if they are in different variants.

1.2.6.2.3 Digital Types

Digital types are a restricted form of records to represent real time digital signals.

Syntax

<digital type> ::= digital <fixed part> end

Example

```
• Variable    Signal_In: digital
                 Valve_1: boolean;
                 LOX_Switch: 1..3;
                 Command: (Idle, stopped, running)
              End;
              for  check_status;
```

Semantics

Due to their physical interpretation, the type of components within digital types may only be boolean, scalar, or subrange. Digital types may not have variant parts and they may not be used as components of any other type.

1.2.6.2.4 Set Types

Set types represent elements of powersets over a finite set of elements called the base type. Conceptually, a set type variable may be viewed as a bitstring of length equal to the number of elements in the base type. Each bit is associated with a unique element and is "on" or "off" if the element is a member or not a member of the powerset.

Syntax

```
<set type> ::= set of <base type>
<base type>::= <simple type>
```

Examples

```
• Type members = (father, mother, big_sister,
       little_sister, big_sister, little_brother);
  Variable family:  set of members; for arrange;
• Variable  Even_numbers:  set of  -10..10;
       for compute_something ;
```

Semantics

The base type must be either scalar or subrange.

1.2.6.2.5 Sequence (File) Types

A sequence differs from an array in that it may vary dynamically in length and is referenced through a "window" called its buffer (not by computed index). Examples of physical representations of sequences include linked lists and mass storage files.

Syntax

```
<sequence type> ::= <file or sequence> of <type>
<file or sequence> ::= file | sequence
```

Examples

```
• Variable Assembly: sequence of record
            part_name: array [1..6] of char;
            order_no: integer;
            drilled, punched, stamped, purchased:
              boolean
      End; for update_orders ;

• Type roster_entry = record
            name: array [1..20] of char;
            rank: 1..16; base_code: 1000..5000
      End;
  Variable roster: file of roster_entry;
            for assign_new_base ;
```

Semantics

All components of sequences must be of identical type. and length. A sequence may not have sequence type or text type components. Furthermore, digital and analog types may not be combined as sequences.

1.2.6.3 Pointer Types

Variables of type pointer are "bound" to a particular type. That is, the contents of a pointer is used to indicate a second variable, and the second variable is required to be of a predetermined, specific type.

Syntax

```
<pointer type> ::= @ <type identifier>
```

Examples

```
• Type combination = record n, p: integer  End ;
  Variable comb_ptr: @ combination; for select_band;

• Type weather_station = record hi,lo: integer;
        rain: real End;
  Variable  ws_ptr: @ weather_station;
      for  record_temperature;
```

Semantics

The contents of a pointer may be altered, but the data element the pointer indicates is always of the same type.

1.2.7 Constant Declarations

In SSL, constant declarations may appear in the preamble of any subsystem and are used to communicate actual values or parameters to the detailed designer. Normally, a constant declaration would be used only for critical values for which the effects are to be isolated in the final code.

Syntax

```
<constant declaration> ::= <constant or constants>
        <constant definition list>
<constant or constants> ::= constant| constants
<constant definition list> ::= <constant definition>
        |<constant definition list>;<constant definition>
<constant definition> ::= <identifier> = <constant>
        |<identifier> = <simple type>
```

Examples

```
●  Constant   a = 10.0 ; max_count = Integer;
●  Constants  Low = true;
              Tax_cut = 1..5 ;
```

Semantics

An identifier declared equal to a simple type indicates that the exact value is not known at the time of specification, but will be provided before implementation. An identifier used in a constant declaration may subsequently be used any place that a constant (of the same type) may be used.

1.2.8 Data References

Data elements may be referenced by variable name, by selected component, or pointer. A variable has components only if it is a record, digital signal, file, or array.

Syntax

```
<variable> ::= <entire variable>
             |<component variable>
             |<referenced variable>
```

1.2.8.1 Entire Variables

```
<entire variable> ::= <identifier>
```

Semantics

A reference to an entire variable includes all fields of a record or digital signal, all elements of an array, or all records of a file. If the data element is a simple, unstructured variable (integer, boolean, etc.) it may only be referenced as an entire variable.

1.2.8.2 Component Variables

Syntax

```
<component variable> ::= <indexed variable >
                       |<field designator>
                       |<file buffer>

<indexed variable> ::= <array variable>
                     [<expression list>]
```

```
<array variable> ::= <variable>

<expression list> ::= <expression>|<expression list> ,
                      <expression>

<field designator>::= <record variable> . <field
                      identifier>

<record variable> ::= <variable>

<file buffer> ::= <file variable> @

<file variable> ::= <variable>
```

<u>Examples</u>

```
Char_Array [15]
Inverse_Matrix [5, I, 16]
Employee.Name
Owner [15] . Accessed_Value
Name_Record.Character [6]
Transaction_File @
Transaction_File @ . Date
Transaction_File @ . Date. Month
```

<u>Semantics</u>

Indexed variables have the conventional meaning. Field designators denote which field component of a record or digital signal type is to be selected. A file buffer variable designates the current active element of the sequence of elements that comprise the file.

Since arrays, files, and records can be combined in various ways (a record of records, file of arrays, array of records, etc.) a component variable can be arbitrarily complex. It is recommended that data structures be as limited in complexity as the problem permits.

1.2.8.3 Referenced Variables

Syntax

```
<referenced variable> ::= <pointer variable> @

<pointer variable> ::= <variable>
```

Examples

```
Symbol_Pointer @
Student_Name [6] @
Assembly@.Manufacturer@
```

Semantics

The data structure denoted by the contents of the pointer variable is substituted for the referenced variable in expression evaluation.

1.2.9 Expressions and Assertions

Expressions arise in two contexts: subscripts of arrays and as terms within assertions. Assertions may appear in either variable declarations or module descriptions.

1.2.9.1 Arithmetic Expressions

Arithmetic expressions in SSL are similar to those in other high level languages. Results of expressions are single valued with type determined by the operation and the constituent operands.

<u>Syntax</u>

```
<arithmetic expression> ::= <term> | <sign> <term>
     |< arithmetic expression> <sign> <term>

<term> ::= <factor> | <term> <multiplying operator>
     <factor>

<factor> ::= <primary> | <factor> ** <primary>|<set>

<primary> ::= <constant identifier> | <unsigned
     number>| <variable> | < function designator >
     |(<arithmetic expression> )

<set> ::= [<element list>]

<element list> ::= <empty> | <element> | <element
     list>, <element>

<element> ::=<expression> | <expression> ..
    <expression>

< multiplying operator>::= * |/

<function designator> ::= <function identifier>
     (<expression list> )

<function identifier> ::= <identifier>
```

Examples

```
a + b
3.0 * sin ( r + 1.0)
2 * (ifix(c) + blank_common.icount)
name.field1
name_set + [joe, fred]
```

Semantics

Mixed mode expressions are prohibited with the exception of the exponentiation operator as indicated in Table 1-3. In Table 1-3, any operand of type integer may be replaced by an operand of type integer subrange. The symbol "dp" indicates double precision. The unary "+" may be used with any operand permitting a unary "-", but is semantically superfluous (i.e. + is the identity operation). If a type is not included in the operand type columns of Table 1-3 then its use with the designated operator is not permitted. Note, however, that integer and integer subrange are interchangable.

SSL does not contain intrinsically defined functions. All function identifiers are accepted, but it is suggested that those embodied in the proposed implementation language be adopted for each specification. Function types are not explicity declared, but must be consistently used throughout the specification. In addition to the basic types (integer, real, etc.), the permissible function types include scalar and subranges of integers and scalars.

TABLE 1-3 ARITHMETIC OPERATIONS

V1 "OP" V2

Operator	Operation	V1 Type	V2 Type	Result Type
	Arithmetic, Negation		Integer	Integer
			Real	Real
			dp	dp
+,-	Addition, Subtraction	Integer	Integer	Integer
		Real	Real	Real
		dp	dp	dp
+,-	Set Union, Set Difference	Set	Set	Set
*,/	Multiplication, Division	Integer	Integer	Integer
		Real	Real	Real
		dp	dp	dp
*	Set Intersection	Set	Set	Set
**	Exponenciation	Integer	Integer	Integer
		Real	Integer	Real
		Real	Real	Real
		dp	Integer	dp
		dp	Real	dp
		dp	dp	dp

1.2.9.2 Boolean Expressions

Combining arithmetic expressions with the boolean operations produces the expressions used in SSL assertions and array subscript lists.

Syntax

<expression> ::= <implication>|<expression> equ <implication>

<implication> ::= <boolean term>|<implication> implies <boolean term>

<boolean term> ::= <boolean factor>|<boolean term> or <boolean factor>

<boolean factor> ::= <boolean secondary>|<boolean factor> and <boolean secondary>

<boolean secondary> ::= <boolean primary>|¬<boolean primary>

<boolean primary> ::= <logical value>|<arithmetic expression>|<relation>|(<assertion>)

<relation> ::= <arithmetic expression><relational operator><arithmetic expression>

<relational operator> ::= <|<=| =|>= |¬ = | in

Examples

```
Rate = 7.0
Value and  Qual
a>b Implies c>0.0
S ¬= t  Equ  p<t
Color in [red, green, yellow]
abs (buffer @.velocity) <16.0 and  weight >= 14.0
```

Semantics

The arithmetic and boolean operators are grouped into hierarchial levels as exhibited in Table 1-4. Operations are performed in the order of highest hierarchial level first followed by equal hierarchial levels from left to right. This sequence may be overridden by parentheses, in which case the innermost operations are performed first. The meaning of the logical operators ¬ (not), _and_, _or_, _implies_, and _equ_ (equivalent) is given in Table 1-5.

Table 1-6 depicts the required operand types for the boolean and relational operators. For set types, the symbols "[ ]" stand for the empty set. When comparing set types to scalars, the base type of the set must be the same as that of the scalars. The operators <, <=, =, >=, >, ¬= stand for less than, less than or equal, equal, greater than or equal, greater than, and not equal respectively. Relational operators (other than _in_) may be used to compare arrays of equal length composed of characters, in which case they denote alphabetical ordering.

TABLE 1-4. OPERATION HIERARCHY

Level	Operations
1	Equ
2	Implies
3	Or
4	And
5	¬
6	<, < =, =, >=,> ,¬ =, In
7	+, -
8	*, /
9	**

TABLE 1-5 LOGICAL OPERATOR TRUTH TABLE

b1	false	false	true	true
b2	false	true	false	true
¬ b1	true	true	false	false
b1 And b2	false	false	false	true
b1 Or b2	false	true	true	true
b1 Implies b2	true	true	false	true
b1 Equ b2	true	false	false	true

TABLE 1-6 BOOLEAN AND RELATIONAL OPERATIONS

V1 "OP" V2

Operator	Operation	V1 Type	V2 Type	Result Type
	Compare	Integer	Integer	
		Real	Real	
		dp	dp	Boolean
		Boolean	Boolean	
		Char	Char	
		Scalar	Scalar	
In	Set Inclusion	Scalar	Set	
		Set	Scalar	
		Set	Set	Boolean
		Subrange	Set	
		Set	Subrange	
¬	Logical Inversion	Boolean	Boolean	Boolean
And	Logical "And"	Boolean	Boolean	Boolean
Or	Logical "Or"	Boolean	Boolean	Boolean
Implies	Logical Implication	Boolean	Boolean	Boolean
Equ	Logical Equivalence	Boolean	Boolean	Boolean

1.2.9.3 Assertions

Assertions are conditions which may assume only true/ false values. They are attached to variables at their point of declaration and to modules. Module assertions depict entry and exit data conditions.

<u>Syntax</u>

<assertion> ::= <expression><forall clause>
<forall clause> ::= <empty>| <u>forall</u> identifier = <set>

<u>Examples</u>

- a [i] = 0.0 <u>forall</u> i = [1..n-1]
  (b.c [j] = t [k] <u>forall</u> j = [1,3,4..16]) <u>forall</u> k = [16..30]
- big>small
- code = 1 <u>implies</u> (eof <u>equ</u> <u>true</u>)

<u>Semantics</u>

The scope of the identifier in the <forall clause> is the assertion in which it is used and must not overlap that of a local or global variable of the same name. Its type is assumed to be the base type of the set within the <forall clause>. The set must represent a finite number of elements and may not be empty.

The expression within the assertion may assume only the values true and false. If the <forall clause> is present, the expression is evaluated once for each unique value which the <forall identifier> can assume from the set.

1.2.10 Module Descriptions

Modules are basic system objects in an SSL system description. In using SSL, one identifies for each module:

- The module name
- Input and output data
- Conditions placed on data upon entry to and exit from the module
- Dependence of the module on environmental objects and other modules

The rule of correspondence between input and output data is not stated in SSL. Its statement is a function of detailed design.

Syntax

```
<module description> ::= <module statement>;
     <module definition part>  end

<module definition part> ::= <module definition
     statement>|<module definition part> ;
     <module definition statement>

<module definition statement> ::= <assumes statement>
     |<satisfies statement>|<fulfills statement>
     |<accesses statement>|<modifies statement>
     |<creates statement>|<uses statement>
     |<receives statement>|<transmits statement>
     |<executes statement>
```

1.2.10.1 Module Statement

The module statement is always the first statement of a module description. It identifies the module by name and declares the local variables (if any).

Syntax:

```
<module statement> ::= <module or entry> <module
     identifier> <release variable group>


<module or entry> ::= module|entry

<release variable group> ::= <empty>|(<release variable
     list>)
<release variable list> ::= <release variable>|<release
     variable list>; <release variable>

<release variable> ::= <variable > |<local variables>
<local variables> ::= <identifier list>:<simple type>
<module identifier> ::=<identifier>
```

Examples

- module matrix_multiply;
- entry push_stack (stack_item:stack_entry);
- module permutation (m, n:integer; elements:p_array);

Semantics

A module statement introduced by module can only be referenced from within the subsystem in which it is declared.

A module statement introduced by entry can be referenced only from subsystems other than the one in which it is declared.

Release variables occur both in module statements and virtual references within execute statements. Local variables within a release group serve strictly for communication between the module and those calling it. In this respect, they differ from global variables declared in the subsystem preamble which serve to communicate among modules having common requirement attributes. Local variable identifiers must be unique throughout a subsystem. Only the module statements introducing entry modules are permitted release variables which are not local variables. The variables of a release group for a module statement of an entry module must agree in type, number, and sequence to each virtual reference to it from other subsystems.

1.2.10.2 Assumes and Satisfies Statements

The assumes and satisfies statements specify truth conditions for data.

Syntax

```
<assumes statement> ::= <assume or assumes>
     <assertion list>
<satisfies statement> ::= <satisfy or satisfies>
     <assertion list>
<assume or assumes> ::= assume| assumes
<satisfy or satisfies> ::= satisfy| satisfies
```

Examples

- Assume a >0.0 ;

- Satisfies big_sister in family; count ¬ = 0 ;

Semantics

The assumes statement specifies data conditions that must be true upon module entry. The satisfies statement specifies data conditions that must be true upon module exit. Variables used in assertions must be either local variables in the release set or in the availability set pertinent to the module. (The availability set consists of those variables having requirement attributes which subsume all requirement attributes of the module.)

1.2.10.3 Fulfills Statement

The fulfills statement attaches requirement attributes to a module.

Syntax

```
<fulfills statement> ::= <fulfil or fulfills>
     <requirement attribute list>

<requirement attribute list> ::= <attribute identifier>
     |<requirement attribute list> , <attribute
     identifier>
<attribute identifier> ::= <transduction identifier>
     |<constraint identifier>
<fulfil or fulfills> ::=  fulfil | fulfills
```

Examples

- fulfills size_constraint, cluster;

- fulfil name_list

Semantics

All modules must have at least one transduction identifier attached as a requirement attribute. All attribute identifiers must be declared in the preamble to the subsystem in which the module is declared.

1.2.10.4 Accesses Statement

The accesses statement is used to indicate which environmental objects (chiefly peripherals) are utilized by a module.

Syntax

```
<accesses statement> ::= <access or accesses>
     <environmental object list>

<access or accesses> ::=  access | accesses

<environmental object list> ::= <environmental
     object identifier>| <environmental object list> ,
     <environmental object identifier>
<environmental object identifier> ::=<identifier>
```

Examples

- Access line_printer;

- Accesses real_time_clock, system_disk ;

Semantics

For each environmental object there must be a unique identifier for which the scope is the entire specification.

1.2.10.5 Receives and Transmits Statements

The receives and transmits statements are used to indicate real time data activity such as is associated with telecommunications, analog, and digital signals.

Syntax

```
<receives statement> ::= <receive or receives>
     <from clause>|<receives statement> ; <from clause>

<from clause> ::= <entire variable list> from
     <environmental object identifier>

<transmits statement> ::= <transmit or transmits>
     <to clause>|<transmits statement> ;
     <to clause>

<to clause> ::= <entire variable list> to
     <environmental object identifier>
```

<receive or receives> ::= <u>receive</u> | <u>receives</u>

<transmit or transmits> ::= <u>transmit</u> | <u>transmits</u>

<entire variable list> ::= <entire variable>
    |<entire variable list> , <entire variable>

<u>Examples</u>

- <u>Receive</u> weight <u>from</u> strain_gage_1;

- <u>Transmits</u> course_correction <u>to</u> ground_control;

<u>Semantics</u>

The scope of the environmental object name is the entire specification. Note that components of structured variables may not be transmitted or received.

1.2.10.6 Creates, Modifies, and Uses Statements

The creates, modifies, and uses statements distinguish between input and output data variables. They may also indicate how the two are related in a manner short of a rule of correspondence. A complete rule of correspondence (algorithm) is a task of detailed design and not of SSL.

<u>Syntax</u>

<creates statement> ::= <create or creates>
    <create list>

<modifies statement> ::= <modify or modifies>
    <modify list>

<modify list> ::= <variable list><using clause>
 |<modify list>; <variable list><using clause>

<create list>::= <entire variable list><using clause>
 |<create list>;<entire variable list><using clause>

<uses statement> ::= <use or uses> <variable list>

<create or creates> ::= create|creates

<modify or modifies> ::= modify|modifies

<use or uses> ::= use|uses

<using clause> ::= <empty>|using <variable list>

<variable list> ::= <variable>| <variable list >,
<variable>

<u>Examples</u>

- <u>create</u> employee_array <u>using</u> name_file;
- <u>modifies</u> count, fica_rate <u>using</u> tax_table,
  salary_scales;
- <u>modify</u> pressure·weight [4] , names [10] ·initials;
- <u>uses</u> cluster@, transaction_file;

Semantics

The order of the variable references in any variable list has no significance.

The variables within a using clause or a uses statement are input variables. A variable may be both input and output. An input variable in a using clause indicates that its contents are instrumental in determining the final contents of the output variables within the same statement extending to the first semicolon on the left.

The presence of a variable in the output list of a creates statement indicates the first use (in a dynamic sense) of that variable. This does not mean, however, that the variable may not appear previously in the sequential listing of the SSL program. The implication of the creates statement is that all variables in the output list are first computed or initialized in the module being described. All variables declared in the subsystem preamble must appear as an output variable in exactly one creates statement within the subsystem unless it is a release variable of an entry module.

All variables appearing in a creates, modifies or uses statement (other than the output list of the creates statement) must be in the availability set for the module. A variable is in the availability set of a module if the transduction requirement attributes of the variable subsume all the transduction requirement attributes of the module.

1.2.10.7 Execute Statement

The execute statement designates modules which are called by the module being described. It may indicate that specific modules are called iteratively, conditionally, or both.

Syntax

```
<executes statement> ::= <execute or executes>
     <call list>|<executes statement>; <call list> •

<call list> ::= <module reference list>| <module
     reference list>  <call list tail>
   |<call list tail>

<call list tail> ::= <iteratively clause >
     |<conditionally clause>

<iteratively clause> ::=  iteratively <module
     reference list> | iteratively <call list tail>

<conditionally clause> ::= conditionally <module
     reference list>

<execute or executes >::=  execute|executes

<module reference list> ::= <module reference>
     |<module reference list> , <module reference>

<module reference> ::= <concrete reference>
     |<virtual reference>
```

<concrete reference> ::= <module identifier>

<virtual reference> ::= <subsystem identifier> .
    <module identifier><release variable group>

Examples

- Execute matrix_multiply, cluster·group (pointer@);

- Execute iteratively suba, subb;
  conditionally subc, subd, sube;

- Executes sqrt; iteratively cos conditionally sin;

Semantics

The order of module identifiers in the module reference lists is not significant. The domain of either an iteratively or conditionally clause extends to the next semicolon. An iteratively clause may overlap another clause.

Presence of a module identifier in a iteratively clause connotates that it is called from within a loop. Presence in a conditionally clause connotates the module is not always called. If present in neither, the module is called unconditionally but not from within a loop.

A concrete reference is a call to a module within the same subsystem. A concrete reference may never be to an entry module. A virtual reference is a call to a module of a different subsystem and must always be to an entry module.

Within the release variable group, the local variable format must be used for variables never before defined. A variable may have been defined in the preamble to the subsystem or in the last module statement. The entry module to which the virtual reference refers must have the same release list with respect to number, order, and type of variables. All variable types used in a virtual reference release list must be either intrinsically defined (boolean, real, text, etc.) or global types.

1.2.11 Subsystem Descriptions

Subsystems are independent software units, each with its own requirement declaration. Subsystems may not share global variables but communicate via the release group variables of virtual references and entry modules. The only identifiers with scope greater than a single subsystem are global type identifiers, environment object identifiers, subsystem identifiers, and function identifiers.

Syntax

```
<subsystem description> ::= <subsystem preamble> ;
     <module description list> end

<module description list> ::= <module description>
     |< module description list>; <module
     description>

<subsystem preamble> ::= <preamble declaration list>
     | subsystem <subsystem identifier> ; <preamble
     declaration list>
```

```
<subsystem identifier> ::= <identifier>

<preamble declaration list> ::= <preamble declaration>
     |<preamble declaration list> ; <preamble
     declaration>

<preamble declaration> ::= <requirement declaration>
     |<type declaration>|<variable  declaration>
     |<constant declaration>

<subsystem description list> ::= <subsystem
     description>|<subsystem description list> ;
     <subsystem description>

<specification> ::= <subsystem description list>
     end
```

Example

```
Requirement transduction  sort_descend; input n,
   sort_array; output sort_array  end;
Variable  sort_array:array [1..1000] of real;
          for   sort_descend;
          subjectto  sort_array[i] >0.0 forall i =
          [1..n-1] ;
     n:1..1000; for  sort_descend;

Module sort;

     fulfills   sort_descend;
     accesses   card_reader, line_printer;
     creates    n,  sort_array;
     modifies   sort_array  using  n, sort_array;
```

```
        satisfies   sort_array[i] >= sort_array[i+1]
                    forall i = [1..n-1]
            :
            :
        End

    End
    End
```

Semantics

Each subsystem must have a requirement declaration that contains at least one transduction identifier and one output variable. There must also be at least one module description. The first subsystem declared (called the "main" subsystem) does not have a subsystem identifier; all others must have a unique identifier. The scope of the subsystem identifier is the entire specification.

The nonterminal symbol <specification> is the distinguished symbol of the SSL grammar.

1.3 EXAMPLE

The example of this section was selected to demonstrate both the descriptive level of SSL and as many language elements as possible. The requirement of the problem may be stated as follows [8]:

> "A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer where it is to be processed. The words in the telegram are separated by sequences of blanks and each telegram is delimited by the word 'ZZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZZ' and 'STOP' are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word."

To complete the problem statement, several assumptions are necessary. The following alternatives were selected for the purpose of this exposition:

- The character stream from which the telegrams are constructed resides on a drum having fixed length records; the record length itself is left as an implementation option.
- The chargeable word count is the value to be printed and overlength words count as one word.
- If a physical end of file is encountered before the logical end of the data stream, an error message and the partial telegram is printed.

The software is organized into four modules as indicated by Figure 1-2. The purpose of each module is given in Table 1-7. Figure 1-3 contains the SSL description of the telegram processor. The right margin of the statement listing contains reference notes to subsections containing detailed descriptions of the language elements used.

A careful examination of Figure 1-3 will indicate an interesting application of the subsystem capability. The subroutines GET_CHAR and FILL_BUFFER occupy a separate subsystem with the sole purpose of handling file I/O. The characteristics of the device on which the telegrams are stored are encapsulated within these two modules.

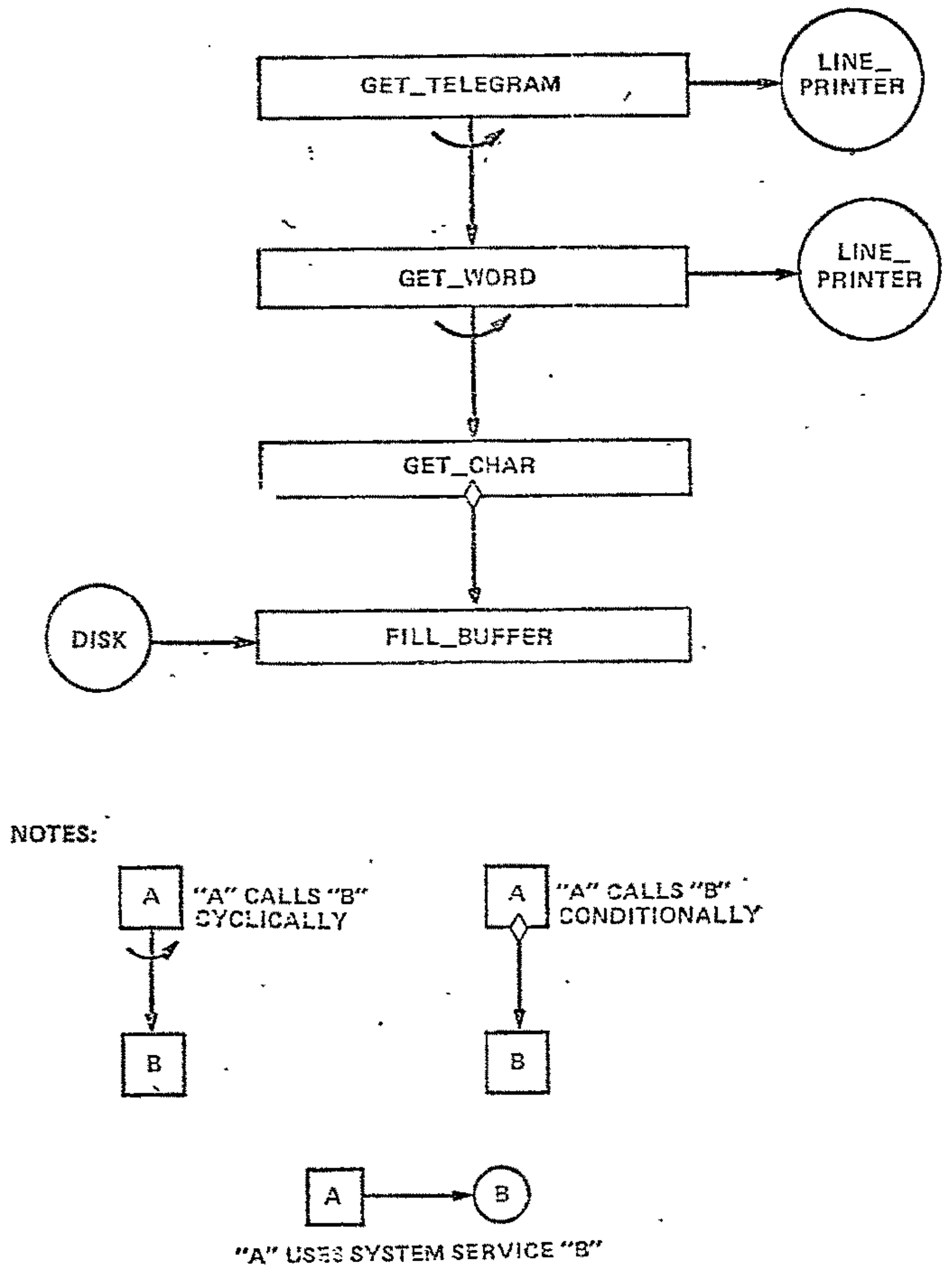
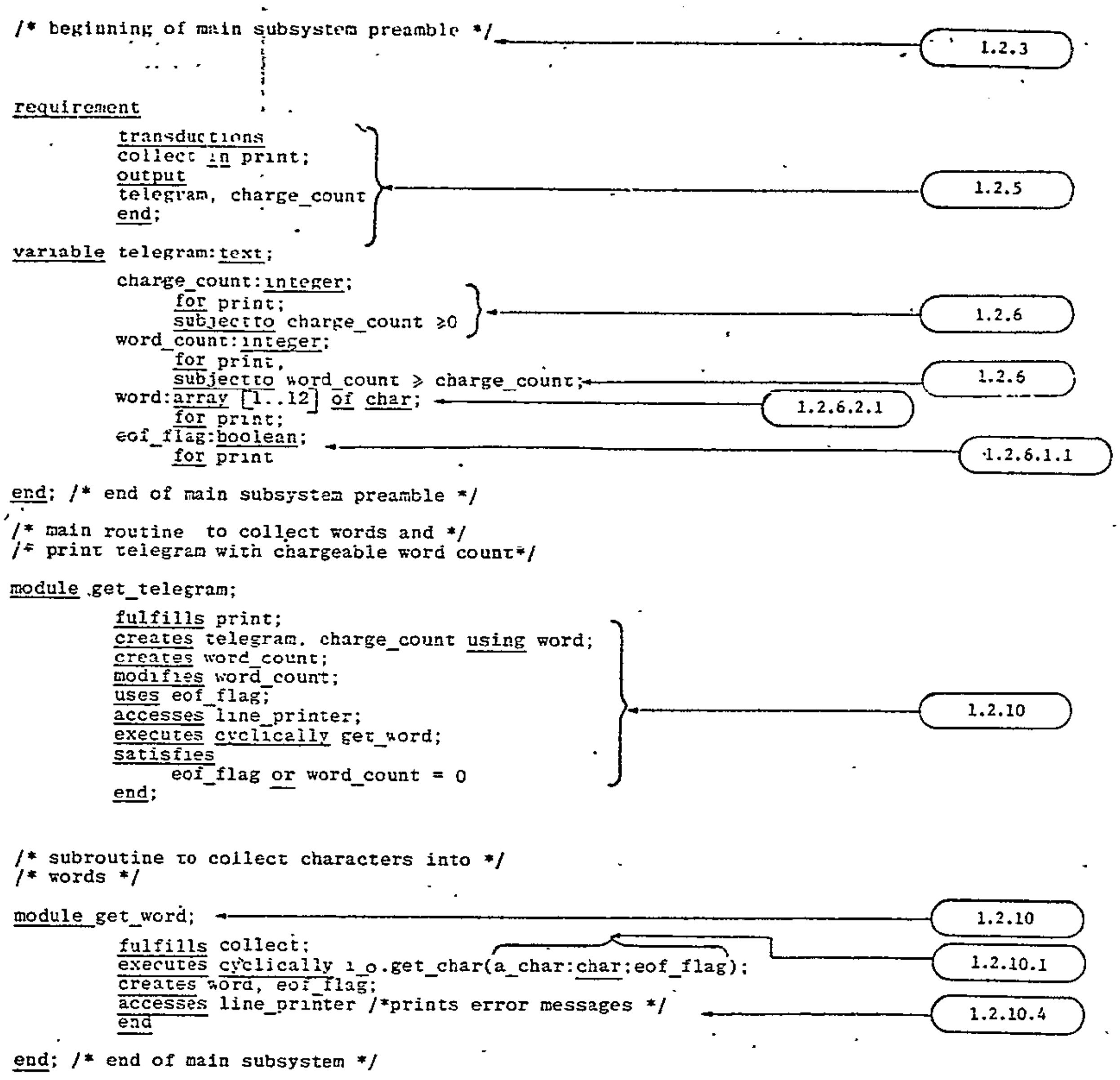


Figure 1-2 Module Structure Chart for Example

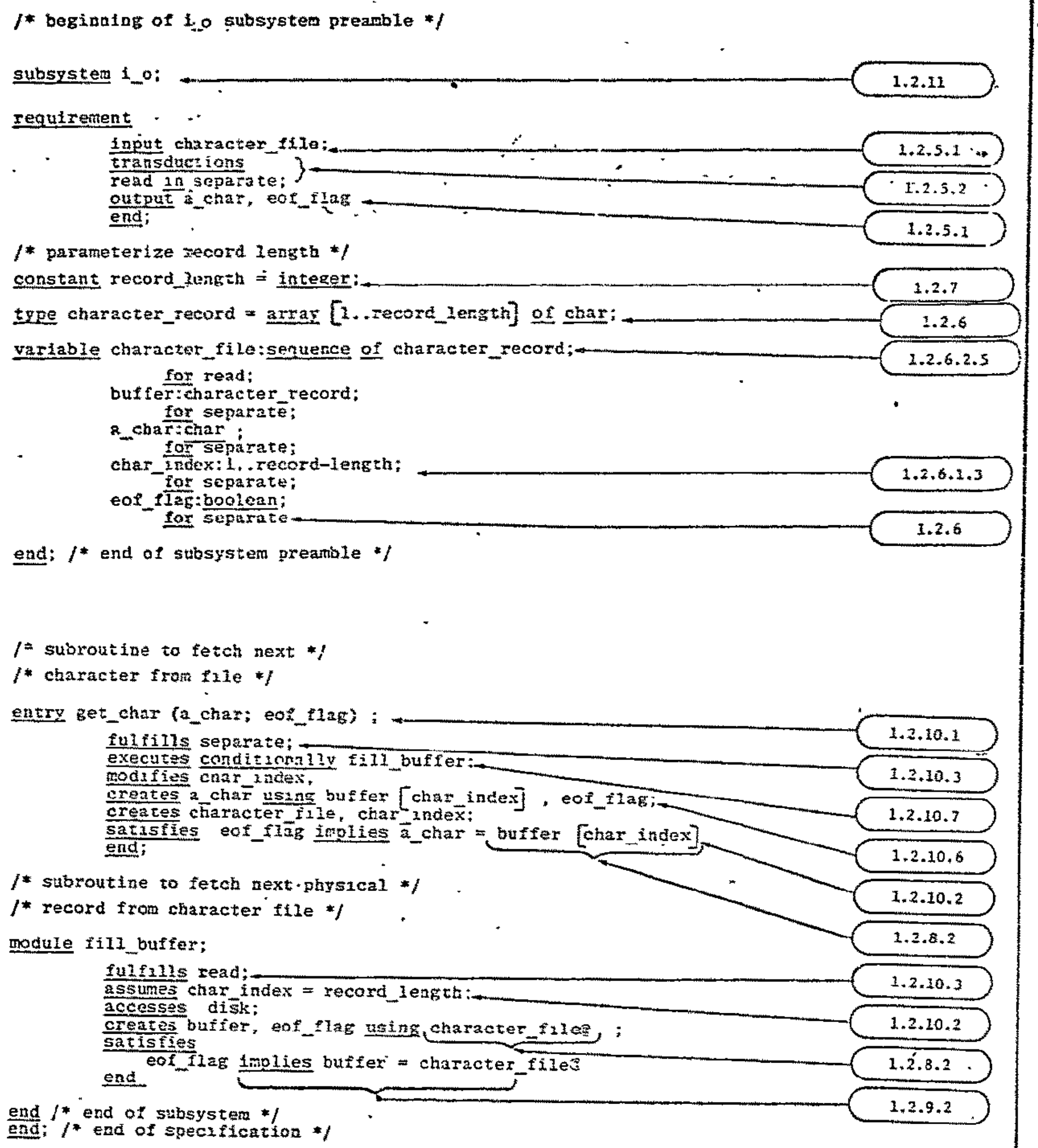
TABLE 1-7 MODULE DESCRIPTIONS FOR EXAMPLE

MODULE	PURPOSE
GET_TELEGRAM	Collects words belonging to each telegram and prints them in a neat manner along with the chargeable word count.
GET_WORD	Collects characters into words and prints error messages denoting over-length word or physical record end of file.
GET_CHAR	Returns the next character in the telegram file.
FILL_BUFFER	Enters the next physical record from the drum into the character buffer.



```
/* beginning of main subsystem preamble */

requirement
        transductions
        collect in print;
        output
        telegram, charge_count
        end;
variable telegram:text;
        charge_count:integer;
             for print;
             subjectto charge_count ≥0
        word_count:integer;
             for print,
             subjectto word_count ≥ charge_count;
        word:array [1..12] of char;
             for print;
        eof_flag:boolean;
             for print
end; /* end of main subsystem preamble */

/* main routine  to collect words and */
/* print telegram with chargeable word count*/

module get_telegram;
        fulfills print;
        creates telegram, charge_count using word;
        creates word_count;
        modifies word_count;
        uses eof_flag;
        accesses line_printer;
        executes cyclically get_word;
        satisfies
             eof_flag or word_count = 0
        end;

/* subroutine to collect characters into */
/* words */

module get_word;
        fulfills collect;
        executes cyclically i_o.get_char(a_char:char;eof_flag);
        creates word, eof_flag;
        accesses line_printer /*prints error messages */
        end

end; /* end of main subsystem */
```

Figure 1-3   SSL Description for Example



```
/* beginning of i_o subsystem preamble */

subsystem i_o;

requirement
        input character_file;
        transductions  }
        read in separate;
        output a_char, eof_flag
        end;

/* parameterize record length */
constant record_length = integer;

type character_record = array [1..record_length] of char;

variable character_file:sequence of character_record;
             for read;
        buffer:character_record;
             for separate;
        a_char:char ;
             for separate;
        char_index:1..record-length;
             for separate;
        eof_flag:boolean;
             for separate

end; /* end of subsystem preamble */


/* subroutine to fetch next */
/* character from file */

entry get_char (a_char; eof_flag) ;
        fulfills separate;
        executes conditionally fill_buffer;
        modifies char_index,
        creates a_char using buffer [char_index] , eof_flag;
        creates character_file, char_index;
        satisfies  eof_flag implies a_char = buffer [char_index]
        end;

/* subroutine to fetch next physical */
/* record from character file */

module fill_buffer;
        fulfills read;
        assumes char_index = record_length;
        accesses  disk;
        creates buffer, eof_flag using character_file@ ;
        satisfies
            eof_flag implies buffer = character_file@
        end

end /* end of subsystem */
end; /* end of specification */
```

Figure 1-3 SSL Description for Example (continued)

2. INDEX

Each reference in the left column is to a nonterminal symbol. The right column contains the number of the subsection in which the nonterminal is defined via one more productions.

Access or accesses	1.2.10.4
Access statement	1.2.10.4
Arithmetic expression	1.2.9.1
Array type	1.2.6.2.1
Array variable	1.2.8.2
Assertion	1.2.9.3
Assertion list	1.2.6
Assume or assumes	1.2.10.2
assumes statement	1.2.10.2
attribute identifier	1.2.10.3
Base type	1.2.6.2.4
Basic type	1.2.6.1.1
Boolean factor	1.2.9.2
Boolean primary	1.2.9.2
Boolean secondary	1.2.9.2
Boolean term	1.2.9.2
Call List	1.2.10.
Call list tail	1.2.10.7
Case Label	1.2.6.2.2
Case label list	1.2.6.2.2

Concrete reference	1.2.10.7
Conditionally clause	1.2.10.7
Component type	1.2.6.2.1
Component variable	1.2.8.2
Constant	1.2.6.1.3
Constant declaration	1.2.7
Constant definition	1.2.7
Constant definition list	1.2.7
Constant identifier	1.2.6.1.3
Constant or constants	1.2.7
Constraint identifier	1.2.5.3
Constraint list	1.2.5.3
Constraint or constraints	1.2.5.3
Constraint part	1.2.5.3
Create list	1.2.10.6
Create or creates	1.2.10.6
Creates statement	1.2.10.6
Decimal number	1.2.4.2
Digit	1.2.3
Digital type	1.2.6.2.3
Element	1.2.9.1
Element list	1.2.9.1
Empty	1.2.1
Entire variable	1.2.8.1
Entire variable list	1.2.10.5
Environmental object identifier	1.2.10.4
Environmental object list	1.2.10.4

Execute or executes	1.2.10.7
Executes statement	1.2.10.7
Exponent part	1.2.4.2
Expression	1.2.9.2
Expression list	1.2.8.2
Factor	1.2.9.1
Field designator	1.2.8.2
Field identifier	1.2.6.2.2
Field identifier list	1.2.6.2.2
Field list	1.2.6.2.2
File buffer	1.2.8.2
File or sequence	1.2.6.2.5
File variable	1.2.8.2
Fixed part	1.2.6.2.2
For clause	1.2.6
Forall clause	1.2.9.3
From clause	1.2.10.5
Fulfil or fulfills	1.2.10.3
Fulfills statement	1.2.10.3
Function designator	1.2.9.1
Function identifier	1.2.9.1
Identifier	1.2.4.1
Identifier list	1.2.6
Implication	1.2.9.2
Index list	1.2.6.2.1

Index type	1.2.6.2.1
Indexed variable	1.2.8.2
Input or inputs	1.2.5.1
Input part	1.2.5.1
Iteratively clause	1.2.10.7
Letter	1.2.3
Local variables	1.2.10.1
Logical value	1.2.4.3
Modifies statement	1.2.10.6
Modify or modifies	1.2.10.6
Module definition part	1.2.10
Module definition statement	1.2.10
Module description	1.2.10
Module description list	1.2.11
Module identifier	1.2.10.1
Module or entry	1.2.10.1
Module reference	1.2.10.7
Module reference list	1.2.10.7
Module statement	1.2.10.1
Multiplying operator	1.2.9.1
Output or outputs	1.2.5.1
Output part	1.2.5.1
Pointer type	1.2.6.3
Pointer variable	1.2.8.3

Preamble declaration 1.2.11
Preamble declaration list 1.2.11
Primary 1.2.9.1
Receive or receives 1.2.10.5
Receive statement 1.2.10.5
Record section 1.2.6.2.
Record type 1.2.6.2.
Record variable 1.2.8.2
Referenced variable 1.2.8.3
Relation 1.2.9.2
Relational operator 1.2.9.2
Release variable 1.2.10.1
Release variable group 1.2.10.1
Release variable-list 1.2.10.1
Requirement attribute list 1.2.10.3
Requirement declaration 1.2.5
Requirement or requirements 1.2.5
Requirement statement group 1.2.5
Requirement statement part 1.2.5
Satisfies statement 1.2.10.2
Satisfy or satisfies 1.2.10.2
Scalar type 1.2.6.1.2
Set 1.2.9.1
Set type 1.2.6.2.4
Sequence type 1.2.6.2.5
Sign 1.2.4.2

Simple type	1.2.6.1
Specification	1.2.11
Structured type	1.2.6.2
Subjectto clause	1.2.6
Subrange type	1.2.6.1.3
Subsystem description	1.2.11
Subsystem description list	1.2.11
Subsystem identifier	1.2.11
Subsystem preamble	1.2.11
Tag field	1.2.6.2.2
Term	1.2.9.1
To clause	1.2.10.5
Transduction clause	1.2.5.2
Transduction identifier	1.2.5.2
Transduction list	1.2.5.2
Transduction or transductions	1.2.5.2
Transduction part	1.2.5.2
Transmit or transmits	1.2.10.5
Transmits statement	1.2.10.5
Type	1.2.6
Type declaration	1.2.6
Type definition	1.2.6
Type identifier	1.2.6.1
Type or types	1.2.6
Unsigned integer	1.2.4.2
Unsigned number	1.2.4.2

Use or uses	1.2.10.6
Uses statement	1.2.10.6
Using clause	1.2.10.6
Variant	1.2.6.2.2
Variant list	1.2.6.2.2
Variant part	1.2.6.2.2
Variable	1.2.8
Variable declaration	1.2.6
Variable definition	1.2.6
Variable list	1.2.10.6
Variable or variables	1.2.6
Virtual Reference	1.2.10.7

3.1 REFERENCES

1. Frank DeRemer and Hans Kron, "Programming-In-The-Large Versus Programming-In-The-Small," Proceedings 1975 International Conference on Reliable Software, 114-121 (1975).

2. B. H. Liskov, "A Design Methodology for Reliable Software Systems," Proceedings Fall Joint Computer Conference, 191-199 (1972).

3. Larry L. Constantine, "Structure Charts, A Guide," unpublished manuscript (1975).

4. Kathleen Jensen and Niklaus Wirth, PASCAL User Manual and Report, Springer-Verlag, New York, N.Y. (1975).

5. Peter Naur (ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM 6, 1-17 (Jan. 1963).

6. D.I. Good and L. C. Ragland, "NUCLEUS - A Language of Provable Programs," In William C. Hetzel (ed.), Program Test Methods, Prentice-Hall, Englewood Cliffs, N.J., 29-40 (1973).

7. E. Lohse (ed.), "Correspondence of 8-Bit Hollerith Codes for Computer Environments," Comm. ACM 11, 783-789 (Nov. 1968).

8. P. Henderson and R. Snowdon, "An Experiment in Structured Programming," BIT 21, 38-53 (1972).