

SPL COMPILER

by

Daniel Ross

December 1968

Technical Report

Prepared Under

Contract AT(04-3)-515

for the USAEC

San Francisco Operations Office

Also supported in part by The Advanced Research Projects Agency of the  
the Department of Defense and by Carnegie-Mellon University, Pittsburgh,  
Pennsylvania.

## ABSTRACT

A compiler source language and internal organization are described, which utilize program block structure to provide a virtual memory capability for linked-list hierarchically structured data. A nonprocedural source language notation is introduced, resembling conventional mathematical set notation, for describing the search and selection of the members of subsets of ordered sets. An algorithm is presented for the translation of these statements into conventional compiler loop statements. Some convenience features in compiler source language notation are introduced, including the ability for the compiler to "stay in context" with the programmer. One partial implementation of the compiler is outlined.

## CONTENTS

1	Introduction
2	Comment Convention
3	Data Storage
4	Data Structures
5	Names of Data Constructs
5.1	Type Names
5.2	Local Names
5.3	Releasing and Reserving Names
6	Structure Activity, and Virtual Memory
7	Structure-Pointing Atoms
8	Graphs and Terminology of SPL Trees
9	Staying in Context
10	Isolated Cells
11	Access Chains
12	Storage Assignment
13	Initial Values
14	Creating, Erasing, and Destroying Constructs
14.1	Creating Constructs
14.2	Copying Constructs
14.3	Erasing Constructs
14.4	Destroying Constructs
15	Program Block Structure, BEGIN and END
16	Procedures
17	Loops
17.1	Explicit Loops
17.2	Implicit Loops
17.3	Search and Select Loops
17.4	Implicit Program Block Structure of Explicit Loops
17.5	CYCLE and LEAVE
17.6	Boolean Implicit Loops
17.7	Counting Elements
18	Translating Boolean Search and Select Loops
18.1	Definition of the Problem
18.2	Examples Demonstrating Some of the Problems Involved in Translation
18.3	Developing a Chart
18.4	Developing a Graph
18.5	Interpreting a Chart to Determine Loops
18.6	Clustering S's About the Main Diagonal
18.7	Propagating Dependency
18.8	Shifting Data Atoms to Eliminate Unnecessary Nesting of Loops
18.9	Independence of Loops Executed Sequentially
18.10	Mutual Dependency Among Nested Loops
18.11	EXISTS
18.12	Starting the Search at Some Other Element
18.13	Source Code Errors Not Detectable by Chart
18.14	Specifying Order of Execution
18.15	Translated Code
18.15.1	Simple Loops
18.15.2	Mutual Dependency for Selection
18.15.3	EXISTS as a Selection Criterion
18.15.4	Conditional Statements Using EXISTS
18.16	Selecting All Elements
18.16.1	Interpretation of the Word ALL
18.16.2	Restrictions on the Use of ALL
18.16.3	Representing ALL in Chart and Graph
18.17	Extension of Source Code Syntax for Boolean Search and Select Loops

19	Connecting Elements of a Complex
20	Extensions to the Declarations
20.1	Definitions
20.2	Collections
20.3	Declaration Macros
20.4	Molecules
20.5	Compile-Time Procedures
21	Input/Output
22	Implementation
23	Models
24	Stack
25	Fields Within Local Names
26	When Not to Activate Structures
27	Bookkeeping Fields Within Structures
28	Table of Structure Locations
29	Structure Storage Area
30	Recursive Generator
31	Auxiliary Storage
32	Collections
33	Extensions and Modifications
34	Acknowledgements
35	Bibliography

## 1. INTRODUCTION

SPL is a compiler designed for the processing of heirarchically structured data. The overall appearance of SPL source language, and the internal representation of data formats, both are somewhat similar to those of PL/1. But the detailed differences between the two source languages and between the two data representations allow for significant improvements and extensions of the data handling capability of SPL over PL/1.

The applications for which SPL is particularly useful are those which require a large amount of "pointer chasing". SPL originally was designed as a language in which to write school class scheduling programs. In this application, the types of data structures needed (for describing the properties of students, classes, rooms, instructors, etc.) are known beforehand, and may be declared at compile time. The total amount of data that must be processed is nearly overwhelming -- perhaps 100 times the primary memory storage capacity of the computer. Both bit-packing to conserve memory space, and a virtual memory capability are imperative. But the conceptually difficult part of an application program is expected to be the choice and understanding of the complicated decision-making processes involved in the application. SPL allows the user to concentrate his efforts on the decision-making processes, by simplifying as much as possible the source code statement of these processes, and by automating the system overhead considerations such as bit-packing and virtual memory.

Some of the unique features of SPL are:

- (1) Automatic control of all data constructs, even those which are used in list processing applications, via the program block structure.
- (2) A virtual memory scheme using some auxiliary storage device, such as disk or drum. The scheme employs the program block structure of (1) to predict when data should be retained in primary core storage.
- (3) A concise source language notation for the programming of loops. The loops may range over all the elements of a linked list, or over a selected subset of those elements.
- (4) A unified source language notation for data stored in either tabular form or linked list form, or in any one of the many composite forms which include some table structuring and some linked list structuring. The SPL programmer has the freedom to change the organization of his data merely by changing a few declarations at the beginning of his program.
- (5) Ability of the SPL compiler to "stay in context" with the source language code being supplied to it, much as a person might retain context between sentences of English prose.
- (6) Free storage recovery is performed in an orderly, directed manner. It is known in advance the location and length of regions of consecutive memory which are to be freed.

It is typical of applications such as class scheduling that the rate at which data is created or destroyed is low compared with the rate at which the program shifts its "focus of attention" among existing data constructs. The shifts of attention correspond quite closely with the program block structure (1 above), whereas data creation and destruction are relatively independent of program block structure. For these reasons, SPL program block structure is used to control the focus of attention automatically, while the user is given the responsibility of creating and destroying data. See Section 6.

In addition to the above features, the design of the SPL compiler led to an interesting theoretical study of a translation process: from a nonprocedural source language statement of a search and selection operation, into the backtrack code procedure necessary to execute the search operation. The method developed here enables the translation of a new class of compiler source language statements.

One of the major design considerations of SPL was the development of a very concise source language notation, which still would not restrict the flexibility inherent in the use of linked list structures, nor sacrifice efficiency in program execution. New notations were devised to describe some of the most frequently occurring special cases of more general operations. These special cases also could have been described at greater length without the new notations. The concise notation is most valuable where it allows a complicated process to be described in a single source language statement. For example, an entire loop usually can be described in a single statement, if the action to be performed within the loop can be described in a single statement.

The goal has been to reduce confusion by reducing the number of statements in the source code. However, every effort was taken to avoid introducing cryptic abbreviations of common English words, merely to reduce the number of source string characters that must be typed. Each implementation of SPL is free to adopt its own set of abbreviations, as long as the unabbreviated words also remain valid. The declaration NO ABBREVIATION appearing in the source code prevents SPL from recognizing the abbreviations peculiar to a particular implementation. The strings which otherwise would be translated as abbreviated reserved words, then may be used as names.

A sufficiently large part of the grammar of SPL is context-sensitive, so that it is inappropriate to describe SPL in a metalanguage such as Backus-Naur Form. No metalanguage has been developed to date which achieves the required goals of accuracy, clarity, and economy of notation in describing context-sensitive grammars. The only alternative, and the one taken in this paper, is to describe the language by example.

## 2. COMMENT CONVENTION

Comments may be embedded anywhere within SPL source code. The comments are delimited by two "less than" symbols on the left, and two "greater than" symbols on the right. Example:

```
<<This is a comment.>>
```

Source code containing comments is translated by deleting the comments and the "less than" and "greater than" symbols. The character immediately to the left of the first "less than" and the character immediately to the right of the last "greater than" are translated as though they were adjacent.

## 3. DATA STORAGE

SPL data may be stored in either of two organizations of memory. One of these organizations consists of "data structures", the other of "isolated cells". The difference between the two organizations lies in the way they respond to the SPL program block structure (the equivalent of BEGINS and ENDS in ALGOL-60).

Data structures must be created and destroyed explicitly by the SPL programmer. The duration of existence of data structures is independent of the program block structure, but the number of paths by which data within structures can be accessed, is determined implicitly by the program block structure. Any type of data for numerical or nonnumerical processing, including arrays, may be held in data structures. All data used in list processing must be held in data structures. The virtual memory capability of SPL applies only to data structures.

Isolated cells are created and destroyed as program execution enters and leaves the outermost blocks in which the isolated cells are mentioned. In this respect, isolated cells correspond to the variables of ALGOL.

Where there is no possibility of confusion between data structures and program block structure, data structures sometimes may be called just "structures".

#### 4. DATA STRUCTURES

The format of each type of data structure to be used in a program must be declared at the beginning of the program. During execution of the program, there may simultaneously exist several instances of each declared type of data structure. For example, if a data structure of type HOUSE has been declared, there may exist instances of houses at 107 Main St., 221 Elm St., and 999 Skid Row. The amount of variability allowed between instances of the same declared structure type is shown by example. Fig. 4-1 shows the declaration of structure type HOUSE and the conceptual representation of an instance of a house.

Referring to Fig. 4-1, a structure consists of "atoms" and "complexes". Each atom in the structure is a single-valued attribute. Its value may be a number (STREET NUMBER), an alphanumeric string (STREET NAME), a Boolean truth value (GARAGE), or a pointer to an instance of some declared structure type (HOUSE ON LEFT). The declaration of an atom includes the maximum size for the data contained in the atom, except for those atoms which point to other structures. The declaration of a structure-pointing atom includes specification of the type of structure being pointed to. See Section 7.

Each complex in the structure is a multi-valued attribute, all of whose values are of the same declared type. Each one of the values of a complex is called an "element" of the complex. The declaration of a complex consists of the word COMPLEX, followed by the type name of the complex, followed by the declaration of an element, in parentheses. The number of elements in a complex may vary dynamically during program execution -- for example, the number of elements in PEOPLE IN ROOM. It can be seen that an array, as used in ALGOL, is a special case of an SPL complex. Further discussion of complexes appears in Section 19.

It is necessary to distinguish between structures and elements for reasons of storage allocation. This is explained in Section 6.



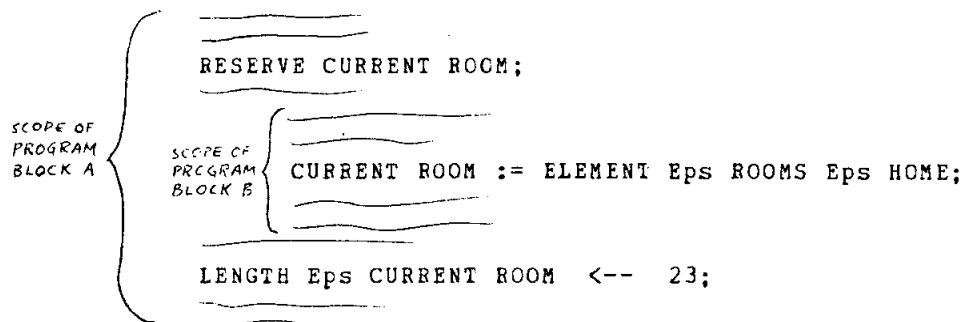
### 5.3. RELEASING AND RESERVING NAMES

Local names may be released explicitly by RELEASE statements. Example:

RELEASE CURRENT ROOM

Any local name which has not been released explicitly, is released implicitly when program execution leaves the block in which it was assigned. See Section 5.2.

If a local name must be used outside the block in which it was assigned, it must be reserved in an outer block. The reserved local name is not implicitly released until program execution leaves the block where it was reserved. Example:



Programmers writing SPL code should rarely, if ever, have occasion to reserve local names. However, the SPL compiler itself often causes local names to be reserved. Programmers must understand the meaning of reserving a local name, in order to understand the semantics of certain other source language statements.

## 6. STRUCTURE ACTIVITY, AND VIRTUAL MEMORY

Typically, computer programs are considered well-organized if they are divided into some sort of functional segments, where any one segment does not need to access all the data "simultaneously". During execution of some functional segment, only the data accessed by that segment need be in core memory. The remaining data can be stored on some auxiliary memory device, such as disk, where direct processing of the data is not possible. This opens the possibility of a program processing more data than can be stored in core memory, provided that (1) there is a way of bringing the data into core memory when it must be processed, and of freeing the core memory space that the data occupied when that processing is completed, and (2) there is an access function which can address every existing item of data uniquely. If storage allocation and addressing can be accomplished automatically, so that a programmer never explicitly writes code for these functions, then the program may be written as though the computer had a "virtual" memory which is larger than its actual core memory.

The virtual memory scheme in SPL is accomplished by introducing the concept of "activity", which is applied to data structures. Data structures are the basic units of storage allocation, in the sense that any given instance of a structure either is entirely in core memory or entirely in auxiliary storage (disk). It is this property which necessitates distinguishing a structure from an element of a complex.

Whenever a structure or any construct within a structure is accessed, the entire structure automatically is brought into core memory, if it is not there already. The core memory space which the structure occupies is taken from some other structure which is not being processed by the currently executing functional segment of the program. The other structure is moved to auxiliary memory and its core memory space is freed automatically by SPL. SPL decides which structures to move by classifying the structures in core memory as either active or inactive; inactive structures may be moved when their space is needed.

Stored in a special bookkeeping area in each structure is an activity count, which is incremented by 1 each time a local name is assigned to any construct within the structure, and decremented by 1 when the local name is released. Any structure with a positive activity count is active.

The activity count also may be incremented and subsequently decremented automatically by SPL, when for certain reasons it becomes necessary to hold a structure in core memory, even though the programmer did not assign a local name.

Since the location where a structure is stored may be changed from time to time, all references to the structure are indirect; they index into a table of structure locations which is an intrinsic part of SPL. Every currently existing structure is uniquely identified by its index number in the table of structure locations.

The assignment of a local name makes the structure active, and consequently immovable; accesses via local names point directly to core memory locations.

## 7. STRUCTURE-POINTING ATOMS

An atom belonging to one structure may contain a pointer to another structure. The pointer consists of the index number of the structure being pointed to. It is independent of structure activity. There does not exist in SPL any type of atom which points to constructs other than structures; this restriction is imposed by the SPL storage allocation scheme. Except as stated at the end of this section, instances of any one type of structure-pointing atom are restricted to pointing either to instances of a single type of structure, or to nothing at all. A structure-pointing atom which points to nothing at all contains the constant 0. In Fig. 4-1, OCCUPANT, HOUSE ON LEFT, and HOUSE ON RIGHT are structure-pointing atoms. An example of a data reference using a structure-pointing atom is:

```
IF COLOR Eps HOUSE ON LEFT Eps HOME = COLOR Eps HOME  
THEN GO TO TRACTHOUSES;
```

In the absence of further notation, an ambiguity would arise in the interpretation of

```
NEIGHBOR := HOUSE ON LEFT Eps HOME
```

Is the local name NEIGHBOR assigned to the structure-pointing atom, or to the structure pointed to by that atom? The question is significant only in determining which structure becomes active. The possible ambiguity is resolved by saying that, in the above situation, the local name is assigned to the structure-pointing atom. A dot meaning "contents of structure-pointing atom" indicates that a local name is assigned to a structure:

```
NEIGHBOR := . HOUSE ON LEFT Eps HOME
```

The restriction that all instances of a single type of structure-pointing atom must point to a single type of structure, enables the compilation of accesses to the structure. Where compilation of accesses is not necessary, the restriction may be relaxed. Certain system functions provided by SPL are fundamentally interpretive in nature. These functions obtain the information about the type of a structure from a private bookkeeping area within the structure itself. Included among these functions are copying, erasing, destroying, and printing the entire contents of a structure. If an SPL programmer can guarantee that the only accesses of the contents of some declared type of structure-pointing atom (let it have type name GARBAGE, for example) are for interpretive functions, then he may let instances of this single type (GARBAGE) of structure-pointing atom point to various types of structures. This is shown in the declaration of the atom by using the word STRUCTURE in place of the type name of a structure:

```
ATOM GARBAGE (STRUCTURE);
```

## 8. GRAPHS AND TERMINOLOGY OF SPL TREES

Each declared SPL structure type forms a tree, if the contents of structure-pointing atoms are ignored. Each instance of a structure also forms a tree, which is closely related to the tree formed by the structure type declaration. Where it is necessary to distinguish between them, we may call them type-trees and instance-trees.

Fig. 8-1 shows a graph of the type-tree for the example structure declared in Fig. 4-1. In Fig. 8-1, STREET NUMBER, STREET NAME, COLOR, MATERIAL, FRONTAGE, ROOMS, SIDE OF STREET, HOUSE ON LEFT, HOUSE ON RIGHT, and GARAGE are called "siblings" of each other. USE, LENGTH, WIDTH, FURNITURE, and PEOPLE IN ROOM are siblings of each other. ITEM NAME and COST are siblings of each other, but not siblings of OCCUPANT. The "first-order ancestor" of COST is ELEMENT Eps FURNITURE, the "second-order ancestor" of COST is FURNITURE, the "third-order ancestor" of COST is ELEMENT Eps ROOMS, etc. The "first-order descendant" of ROOMS is ELEMENT Eps ROOMS, etc. The structure pointed to by a structure-pointing atom is not considered a descendant of the atom.

Referring back to Fig. 4-1 for a graph of an instance-tree, the atoms containing LIVING, 35, and 25, and the two complexes drawn beneath them, all are siblings of each other, but are not siblings of the atom containing KITCHEN. Elements of the same complex, drawn connected together with arrows, are siblings of each other.

It can be seen that a type-tree is isomorphic to an instance-tree in which each complex has exactly one element.

## 9. STAYING IN CONTEXT

If an SPL programmer does not fully qualify a data reference in his source code, the SPL translator still may be able to fill in the remaining qualification needed to make the reference unique. For example, if the source code is:

```
STREET NUMBER Eps HOME <-- 107;
STREET NAME <-- 'MAIN ST.';
COLOR <-- 'RED';
MATERIAL <-- 'BRICK';
FRONTAGE <-- 65;
```

the translator interprets the code as:

```
STREET NUMBER Eps HOME <-- 107;
STREET NAME Eps HOME <-- 'MAIN ST.';
COLOR Eps HOME <-- 'RED';
MATERIAL Eps HOME <-- 'BRICK';
FRONTAGE Eps HOME <-- 65;
```

The ability of the SPL translator to stay in context with its source code allows the programmer to use a more concise notation than fully qualified data references. The concise notation is allowed only in data references whose meanings are "obvious", making any additional qualification "superfluous". The exact interpretations of "obvious" and "superfluous" are described below, but the general approach taken in the design of SPL is to be rather conservative. SPL attempts to be helpful in simple situations, without interpreting the "obvious" so liberally as to introduce spurious source code errors.

SPL maintains a first-in, first-out list, of limited length, containing the names of constructs most recently scanned in the source code. If an incompletely qualified name appears in the source code, the translator tries to match it with the names of siblings and first-order descendants, taken from the type-trees of the construct names already appearing in the list. Although the storage of construct names into the list is first-in, first-out, the searching of type trees is performed first on the construct name most recently stored into the list.

Also, if intermediate qualification is missing but the type can be determined uniquely, SPL automatically supplies the missing qualification. For example, if the source code is:

```
COST Eps HOME <-- 200;
```

the translator interprets the code as:

```
COST Eps ELEMENT Eps FURNITURE Eps ELEMENT Eps ROOMS Eps HOME <--
200;
```

## 10. ISOLATED CELLS

At certain places within a program, it is convenient to store data temporarily in some buffer area that is not associated with any structure. The locations used for this mode of storage are called "isolated cells". They may be used for the storage of numeric, alpha-numeric, or Boolean data, but they may not be used for the storage of pointers to other constructs. Local names and structure-pointing atoms are used for this purpose.

The duration of existence of isolated cells is determined by the program block structure. Each isolated cell is created when program execution enters the outermost block in which the cell is mentioned, and destroyed when program execution leaves that block.

Isolated cell names are unqualified (that is, they do not use "Eps"), since isolated cells do not belong to any other construct. The compiler decides that a name appearing in the source code refers to an isolated cell, if the name is unqualified, not a local name, and the search for additional context (described in Section 9) fails.

A type declaration may appear with the first use of an isolated cell. Example:

```
PI := REAL <-- 3.1416;
```

In the absence of a declaration, the isolated cell assumes the type of the first data stored into it. Example:

```
RUG LENGTH <-- LENGTH Eps ELEMENT Eps ROOMS Eps HOME;
```

Consistent with the declaration in Fig. 4-1, the isolated cell RUG LENGTH assumes the type UNSIGNED INTEGER with a maximum value of 40.

The possible types of isolated cells depend to some extent on the hardware implementation of SPL, but include at least:

```
UNSIGNED INTEGER
INTEGER
BOOLEAN
ALPHANUMERIC
REAL
COMPLEX
```

If the hardware permits, they also may include:

```
LONG REAL
LONG COMPLEX
DECIMAL
```

## 11. ACCESS CHAINS

Source code phrases such as:

```
LENGTH Eps CURRENT ROOM := ELEMENT Eps ROOMS Eps HOME
```

are called "access chains". The example above is the access chain "for" a particular instance of atom LENGTH.

Access chains have slightly differing forms, depending upon where they appear in SPL code. As specifications of formal parameters to a procedure, they must have a local name assignment on the left, no other local name assignments within the access chain, the type name of a structure on the right, and they must not "pass through" any structure-pointing atoms. Example:

```
PROCEDURE PROC1 := USE Eps ELEMENT Eps ROOMS Eps HOUSE  
  (FURN := FURNITURE Eps ELEMENT Eps ROOMS Eps HOUSE;  
   HOME := HGOUSE);
```

When used for the access of some instance of a construct, without creating any new constructs, access chains must have a local name on the right. Example:

```
LENGTH Eps CURRENT ROOM := ELEMENT Eps ROOMS Eps HOME <-- 23;
```

When used for the simultaneous creation of a structure and the access of some construct within the structure, access chains must have the type name of the structure on the right. Example:

```
HUE := COLOR Eps HOME := HOUSE <-- 'GRAY';
```

## 12. STORAGE ASSIGNMENT

The usual syntax for data storage assignment is:

`[DESTINATION] ← [EXPRESSION]`

However, SPL has alternative syntaxes for certain frequently occurring special cases. The syntax:

`[DESTINATION] ←← [EXPRESSION]`

may be used if the programmer can guarantee that the destination field contains 0 (if it is numeric) or blanks (if it is alphanumeric). SPL can compile better code for the double left arrow than for the single left arrow, since it is not necessary to compile the instructions for masking and saving the contents of fields adjacent to the destination field. Since double left arrows restrict the flexibility for future recoding, they are recommended only for improving the efficiency of the innermost nested loops.

Another alternative syntax:

`[DESTINATION] ↔ [DESTINATION]`

indicates a swap of the contents of the two destination fields. The fields must contain the same type data and be of the same size.

The word SAME may be used in place of the expression in a storage assignment statement, if the immediately preceding statement also is a storage assignment statement containing an expression or SAME. Example:

```
LENGTH Eps CURRENT ROOM := ELEMENT Eps ROOMS Eps HOME <-- 20;  
WIDTH Eps CURRENT ROOM <-- SAME;
```

The previously evaluated expression is stored a second time as a result of using SAME.



### 13. INITIAL VALUES

Data atoms may be declared to have constant initial values.  
Example:

```
ALPHANUMERIC ATOM COLOR INITIALLY 'WHITE' (6);  
ALPHANUMERIC ATOM MATERIAL INITIALLY 'WOOD' (5);  
ATOM FRONTAGE INITIALLY 50 (200);
```

In the absence of declared initial values, the default initial values are 0 for numeric atoms, all blanks for alphanumeric atoms, and FALSE for Boolean atoms. Structure-pointing atoms can have only the initial value 0.

### 14. CREATING, COPYING, ERASING, AND DESTROYING CONSTRUCTS

#### 14.1. CREATING CONSTRUCTS

Creating new instances of data structures, or new elements in a complex, is the responsibility of the SPL programmer. Atoms and complexes cannot be created individually. Isolated cells are created automatically, as a consequence of the program block structure.

A new instance of a structure is created implicitly during execution of any access, if the type name of the structure is rightmost in the access chain. Example:

```
HUE := COLOR Eps HOME := HOUSE <-- 'GRAY';
```

Note that declarations and specifications do not cause accesses to be executed; therefore, no new structure is created.

A new instance of an element is created if the access chain contains the word PREFACE, or the word APPEND, or the words INSERT and either BEFORE or AFTER. The particular choice of words designates where among the other existing elements of a complex the new element is to be placed. Examples:

- (1) LENGTH Eps PREFACE ELEMENT Eps ROOMS Eps HOME <-- 23;
- (2) LENGTH Eps CURRENT ROOM := APPEND ELEMENT Eps ROOMS Eps HOME <-- 23;
- (3) LENGTH Eps INSERT ELEMENT AFTER ELEMENT (4) Eps ROOMS Eps HOME <-- 23;
- (4) LENGTH Eps INSERT ELEMENT BEFORE CURRENT ROOM <-- 23;

In example (3), the new element is inserted after the previously existing 4th element of the complex.

Newly created constructs automatically are assigned their declared or default initial values.

#### 14.2. COPYING CONSTRUCTS

COPY is a pre-declared SPL system procedure which interpretively copies a given structure or element and all the descendant constructs of that structure or element. The actual output value of COPY is the identity of the newly created copy. It may be assigned a local name, stored in a structure-pointing atom, inserted into a complex, etc., as appropriate. The actions which may be performed depend on the declared type of structure or element being copied. Example:

```
APPEND COPY (ELEMENT Eps ROOMS Eps HOME) Eps ROOMS Eps HOME;
```

In the above example, a copy of the first element of complex ROOMS is appended to become the last element of complex ROOMS.

#### 14.3. ERASING CONSTRUCTS

Erasing data is the responsibility of the SPL programmer. An entire structure, or any construct within a structure, is erased when an ERASE statement is executed. Data in isolated cells cannot be erased, except by storage assignment statements which put the desired values into the isolated cells.

Erasing an atom is the same as assigning it its declared or default initial value. Erasing a structure or an element is the same as individually erasing all the atoms and complexes within that structure or element. Erasing a complex is the same as erasing all of its elements. In no case does erasure cause the destruction of any construct; it merely changes the data content of the construct being erased. Examples:

- (1) ERASE HOME;
- (2) ERASE CURRENT ROOM := ELEMENT Eps ROOMS Eps HOME;
- (3) ERASE HOUSE ON LEFT Eps HOME;
- (4) ERASE . HOUSE ON LEFT Eps HOME;

As in Section 7, a dot is used to distinguish between erasing a structure-pointing atom, in example (3), and erasing the structure pointed to by the atom, in example (4).

#### 14.4. DESTROYING CONSTRUCTS

Destroying instances of data structures, or elements in a complex, is the responsibility of the SPL programmer. Atoms and complexes cannot be destroyed individually. Isolated cells are destroyed automatically, as a consequence of the program block structure.

Destroying a structure or element completely frees all the storage used by that structure or element. Examples of DESTROY statements:

- (1) DESTROY HOME;
- (2) DESTROY ELEMENT Eps ROOMS Eps HOME;
- (3) DESTROY . HOUSE ON LEFT Eps HOME;

No new local names may be assigned in the access chain of a DESTROY statement. If the named construct (HOME in the examples above) is destroyed, the local name automatically is released. This would occur in example (1). Other than this last possible remaining name (HOME), there must not be any local names still in effect which point to the construct being destroyed, or to any descendant of that construct. If a structure is being destroyed while local names still are in effect, SPL detects this error by a positive activity count. But if an element is being destroyed, SPL cannot detect the error. The consequences of the error may not appear until some later time when the local name either is used or released.

When a structure is destroyed, it is the responsibility of the SPL programmer to erase, destroy, or alter all structure-pointing atoms which point to the structure. If an unaltered reference to the structure subsequently is used, the error possibly may not be detected immediately by SPL. Detection of the error depends upon whether the index in the table of structure locations has been reused.

When an element is destroyed, its sibling elements (if any) automatically are relinked. The element is removed from the complex without damaging the integrity of the rest of the complex.

## 15. PROGRAM BLOCK STRUCTURE, "BEGIN" AND "END"

In ALGOL, program blocks are bracketed by BEGIN and END. All the statements within a block are treated from outside as though they were a single statement. Variables and arrays automatically are created when program execution enters the block, and destroyed when program execution leaves the block.

In SPL, the two functions of program block structure are assigned to separate types of program blocks. Explicit program blocks, which consist of several statements bracketed by BEGIN and END, cause all the enclosed statements to be treated from outside as though they were a single statement. But explicit program blocks have no effect on the duration of validity of local names, or the duration of existence of isolated cells.

Implicit program blocks are recognized by SPL as a consequence of procedure calls or loop statements. The way procedure calls and loops are coded, and the resulting implicit program blocks, are described in Sections 16 and 17. The duration of validity of local names is bounded by the outermost implicit program block in which the local names are reserved. The duration of existence of isolated cells is bounded by the outermost implicit program block in which the isolated cells are mentioned.

Each procedure call or loop statement may result in more than one implicit program block. The executable statements in the body of the procedure or in the scope of the loop are confined to a particular one of the possibly several implicit program blocks. From outside that block, all the statements within the block are treated as though they were a single statement.

Explicit program blocks and implicit program blocks all must be either disjoint or properly nested within each other.

Since isolated cells need not be declared explicitly, a naming conflict might arise if several separate programs are merged into a single program. Local names also might be subject to a naming conflict. To avoid these conflicts, a NEW NAME statement appearing in any program block forces a reinterpretation within that block of the specified names. Example:

```
NEW NAME JOE, PETE, CURRENT ROOM, HOME;
```

All other types of names (besides isolated cells and local names) are required to have sufficient declaration for other reasons, that SPL incidentally is able to resolve naming conflicts.

## 16. PROCEDURES

A procedure declaration consists of a procedure declaration head, a body of executable code, and finally END PROCEDURE. An example of a procedure declaration head is:

```
PROCEDURE PROC1 := USE Eps ELEMENT Eps ROOMS Eps HOUSE
  (FURN := FURNITURE Eps ELEMENT Eps ROOMS Eps HOUSE;
   HOME := HOUSE);
```

In the example, PROC1 is the name of the procedure. The name of a procedure must be unique within the program block in which the procedure is declared.

USE Eps ELEMENT Eps ROOMS Eps HOUSE is the type declaration of the value of the procedure. The type of value a procedure may have may be the type of some construct (USE Eps ELEMENT Eps .... in the example), or any of the types of isolated cells, or LOCATION, or no value at all.

The access chains for FURN and HOME are the formal parameter specifications for the procedure.

Execution starts at the first statement in the body of the procedure. If the procedure has a formal value, then somewhere within the body of the procedure must be the code to assign an actual value to the procedure. The statement RETURN, appearing in the body of the procedure, acts as a special purpose GO TO statement which transfers execution back to the code which called the procedure.

A procedure can be called only within the same program block in which it is declared. A procedure call consists of the procedure name, followed by parentheses enclosing the actual parameters to the procedure. If the procedure has a value, the procedure call may be used in any way that that particular type of value can be used.

Example procedure declaration:

```
(1-1)  PROCEDURE BRICK HOUSE := HOUSE (GIVEN HOUSE := HOUSE);
(1-2)    IF MATERIAL Eps GIVEN HOUSE = 'BRICK'
(1-3)    THEN PRINT ('BRICK HOUSE AT ');
(1-4)    STREET NUMBER Eps BRICK HOUSE := GIVEN HOUSE)
(1-5)    ELSE BEGIN
(1-6)      BRICK HOUSE := BRICK HOUSE (NEIGHBOR := .
(1-7)      HOUSE ON LEFT Eps GIVEN HOUSE);
(1-8)      PRINT ('TO THE RIGHT OF '; STREET NUMBER Eps NEIGHBOR;
(1-9)      ' IS '; STREET NUMBER Eps GIVEN HOUSE;
(1-10)     ', MADE OF '; MATERIAL Eps GIVEN HOUSE)
(1-11)    END
(1-12)  END PROCEDURE;
```

Example procedure call:

```
(2-1)  IF COLOR Eps BRICK HOUSE (- HOUSE ON LEFT Eps HOME) =
(2-2)    COLOR Eps HOME
(2-3)  THEN PRINT ('COLORS MATCH');
```

In the examples above, BRICK HOUSE is a recursive procedure which finds the nearest brick house to the left of a given house, and prints some information about its search. In the example procedure call, the dot indicates that the actual parameter is a structure of type HOUSE, rather than a structure-pointing atom of type HOUSE ON LEFT. Had the dot been omitted from the source code, SPL automatically would have supplied a dot, in order to match the formal parameter specifications.

Fig. 16-1 shows a typical implicit program block structure resulting from a procedure call. In the following discussion, various features in Fig. 16-1 will be related to lines of code in examples (1) and (2) above, although Fig. 16-1 does not exactly correspond with either of the code examples.

The procedure call for Fig. 16-1 appears in the source code in block A. All the other program blocks in Fig. 16-1 are created implicitly for the processing of the procedure call. In general, the statement containing the procedure call also will contain other executable phrases, perhaps even other procedure calls. These other phrases are executed in block A, either before or after the procedure call, depending upon the processing order appropriate to the statement.

If the source code shows any local names are to be assigned during evaluation of the actual parameters (such as NEIGHBOR in code example line (1-6)), these local names are reserved in block A. Reserving the local names is necessary so they will remain valid for later use in block A (line (1-8)), even though the assignment of constructs must occur in an inner block, block B.

If the source code shows a local name assigned to the actual value of a procedure, the local name is reserved in block A. If the source code does not show a local name assigned to the actual value (line (2-1)), then SPL reserves a dummy local name. The dummy local name serves to keep the named construct active during execution of the remainder of the statement after the procedure has returned. The dummy name is released immediately following the statement. Had the formal value of the procedure (the first occurrence of HOUSE in line (1-1)) been declared of type REAL, INTEGER, etc., instead of being declared a type of construct, then an isolated cell would have substituted for the local name or dummy local name.

If the local name for an actual parameter or for the actual value of the procedure already exists in block A or some outer block (the first occurrence of BRICK HOUSE in line (1-6)), there is no need for SPL to reserve the local name.

Program block B acts as an interface between the environment of the procedure call (block A and the outer program blocks), and the body of the procedure (block D). A storage location is reserved in block B for the return branch address of the procedure call. Dummy local names, or isolated cells as appropriate, are created in block B for all the parameters and the value of the procedure. These dummy names appear in the physical order that matches the procedure's specifications. The actual parameters to the procedure are evaluated in block B, from left to right. The evaluated constructs are assigned to the dummy names, and those which were given local names in the source code (NEIGHBOR in line (1-6)) also are assigned to their reserved local names. Assigning at least dummy local names to all the constructs guarantees that the constructs remain active during execution of the body of the procedure.

The access chains for some of the actual parameters may pass through structure-pointing atoms (lines (1-7) and (2-1)). The structures which contain these atoms are activated during evaluation of the actual parameters, but they do not necessarily have to remain active during execution of the procedure. Blocks C1, C2, ..., Cn show the brief activation of these structures.

After the actual parameters have been evaluated, the procedure is called and executes in block D. The procedure assigns the actual output value to the reserved dummy local name in block B, the interface block. When the procedure returns, code in block B copies this assignment into the local name or dummy local name reserved in block A, for use in executing the remainder of the statement containing the procedure call.

If the procedure has no input parameters and no output value, then block B is omitted and the procedure call is executed from block A.

## 17. LOOPS

The critical facility in the coding of complicated decision-making processes is the ease with which associations among data items can be described. Where the underlying organization of data is hash coding, languages like LEAP may be used to describe associations as Boolean relations among the bound variables of associative triples. In SPL the underlying organization of data is a network of pointers, in which associations are described as search loops among ordered sets, to find the members which have the desired properties. Thus much of the language emphasis of SPL is in the concise description of loops, and much of the programmatic emphasis of SPL is in the optimization of those loops.

This section introduces the various notational forms for loops, including Boolean search and select loops. Boolean search and select loops are the most frequently used form for describing associations. The translation from the concise notation of Section 17.3 into the equivalent basic notation of Section 17.1 is not immediately obvious. Section 18 describes that translation, which constitutes one of the major contributions of this paper.

In addition to the loops described below, loops also may be generated by the use of collection names. See Section 20.2.

### 17.1. EXPLICIT LOOPS

There are several ways of coding SPL loop statements. The most basic of these are explicit loop statements. All other ways of coding loop statements are defined in terms of equivalent explicit loop statements.

The syntax of explicit loop statements is to a large extent context free. Fig. 17-1 shows the syntax in the metalanguage "Box Syntax". As can be seen in Fig. 17-1, more than one generator may be coded for a single loop. Each of the generators is advanced after every cycle of the loop. The first generator to terminate causes termination of the entire loop.

In the phrase

FOR [ ITERATION VARIABLE NAME ] ← [ EXPRESSION ] \* \*

the expression may be any of the types allowable for isolated cells, described in Section 10, as long as all replications of the expression are of the same type. In the phrase

[ FORWARD  
BACKWARD ] FOR ALL [ LOCAL NAME ] := ELEMENT [ Eps [ ACCESS CHAIN ] ] \* \*

all the access chains must be for the same type of elements. In the phrase

[ FORWARD  
BACKWARD ] FOR ALL [ LOCAL NAME ] := ELEMENT STARTING AT [ ACCESS CHAIN ]

the access chain must be for the element of a complex. Backward looping is not allowable in complexes declared to have forward links only. See Section 19 for a discussion of the various types of links. In the phrase FOR .... FROM .... STEP .... UNTIL ...., the left arrow and parentheses surrounding arithmetic expressions indicate that the expressions are to be evaluated once only, before executing any cycles of the loop. Without the left arrow and parentheses, the expressions following STEP and UNTIL are re-evaluated before execution of each cycle of the loop.



## 17.2. IMPLICIT LOOPS

If a loop over all the elements of some complex contains only a single executable statement, it may be coded as an implicit loop. An implicit loop uses the word ALL in the access chain to indicate that a loop is desired, and eliminates the words LOOP, FOR, DO, and END LOOP, and the local name for the elements being generated. The scope of an implicit loop is the statement in which it appears. For example, the implicit loop

```
LENGTH Eps ALL ELEMENTS Eps ROOMS Eps HOME <-- 10;
```

is equivalent to the explicit loop

```
LOOP FOR ALL DUMMY1 := ELEMENT Eps ROOMS Eps HOME
DO LENGTH Eps DUMMY1 <-- 10
END LOOP;
```

where DUMMY1 is a local name automatically created by SPL, and assigned successively to each element as it is generated.

A second example, where PRICE is an isolated cell,

```
PRICE <-- 0;
PRICE <-- PRICE + COST Eps ALL ELEMENTS Eps FURNITURE Eps
ALL ELEMENTS Eps ROOMS Eps HOME;
```

is equivalent to

```
PRICE <-- 0;
LOOP FOR ALL DUMMY1 := ELEMENT Eps FURNITURE Eps ALL ELEMENTS Eps
ROOMS Eps HOME
DO PRICE <-- PRICE + COST Eps DUMMY1
END LOOP;
```

which in turn is equivalent to

```
PRICE <-- 0;
LOOP FOR ALL DUMMY2 := ELEMENT Eps ROOMS Eps HOME
DO LOOP FOR ALL DUMMY1 := ELEMENT Eps FURNITURE Eps DUMMY2
DO PRICE <-- PRICE + COST Eps DUMMY1
END LOOP
END LOOP;
```

A third example is to create a list of the costs of all the furniture in HOME. The list will be the elements of a new structure whose declaration is:

```
STRUCTURE PRICE LIST (  
  COMPLEX PRICES (  
    ATOM COST (1000)));
```

The code to create an instance of PRICE LIST, create one new element of PRICES for each item of furniture in HOME, and store the cost of that item of furniture into the new element, is:

```
HOME PRICE LIST := PRICE LIST;  
COST Eps PREFACE ELEMENT Eps PRICES Eps HOME PRICE LIST <--  
  COST Eps ALL ELEMENTS Eps FURNITURE Eps ALL ELEMENTS Eps ROOMS  
  Eps HOME;
```

In the above example, the costs are stored in elements of HOME PRICE LIST in inverse order of their appearance in HOME. They would have been stored in direct order of their appearance in HOME, had APPEND ELEMENT been coded instead of PREFACE ELEMENT. For the simplest types of complexes, where the elements are connected by forward links only, appending elements in the above example would be a computation of order  $n^2$  steps. Prefacing elements would be of order  $n$  steps. With more elaborate linking among the elements, the number of steps in appending elements can be reduced to order  $n$ . See Section 19.

SPL creates an implicit loop for each occurrence of the word ALL in an access chain. If ALL occurs several places in a single access chain, the leftmost occurrence corresponds to the innermost loop, as in the second example above. If ALL occurs in several separate access chains within a single statement, the implicit loops are created in the processing order appropriate to the statement. Each implicit loop created includes all the previously created loops within its scope. These rules do not necessarily apply if the statement contains any Boolean search and select loops, described in Section 17.3.

### 17.3. SEARCH AND SELECT LOOPS

If an access chain contains the word ELEMENT followed by parentheses enclosing an arithmetic expression, such as:

```
THIS ROOM := ELEMENT (7*L+2) Eps ROOMS Eps HOME;
```

then the arithmetic expression is the index of the particular element selected. In the above example, local name THIS ROOM is assigned to the (7\*L+2)th element in the complex. SPL creates a numeric search and select loop, which sequences along the elements of the complex ROOMS until the proper element is selected. In order to avoid possible side effects, the arithmetic expression is not evaluated until immediately before the execution of the loop. The code for the explicit loop equivalent of the above example is:

```
THIS ROOM := ELEMENT Eps ROOMS Eps HOME;  
LOOP ENTIER (7*L+2) -1 TIMES  
DO THIS ROOM := ELEMENT AFTER THIS ROOM  
END LOOP;
```

If the arithmetic expression does not evaluate to an integer, it is truncated to an integer. The truncated value must be strictly positive.

The equivalent explicit loop statement takes an error exit if the complex ROOMS does not have at least the specified number of elements. The code shown below is not equivalent to the source statement, because the code below does not take an error exit if there are an insufficient number of elements.

```
RESERVE THIS ROOM;  
LOOP FOR ALL THIS ROOM := ELEMENT Eps ROOMS Eps HOME;  
  ENTIER (7*L+2) TIMES  
DO  
END LOOP;
```

If an access chain contains the word ELEMENT followed by parentheses enclosing a Boolean expression, such as:

```
THIS ROOM := ELEMENT (LENGTH Eps THIS ROOM > 20) Eps ROOMS Eps  
HOME;
```

then SPL creates a Boolean search and select loop, which sequences along the elements of the complex ROOMS until the first element is found for which the Boolean expression has the value TRUE. An error exit is taken if no element of the complex satisfies the Boolean expression.

Boolean search and select loops provide a means of selecting one element among the possibly many elements of a complex, based on some property of that element. In the example above, the selected element must have the property that the LENGTH field contains a number > 20. The Boolean expression describing this property may be arbitrarily complicated, but of course it ultimately must depend on some property of the element being selected. It would be meaningless to attempt to select an element, if the selection were not based on any property of that element.

Other examples of Boolean search and select loops are:

```
HOME PRICE LIST := PRICE LIST;
COST Eps PREFACE ELEMENT Eps PRICES Eps HOME PRICE LIST <--
  COST Eps ALL EXPENSIVE := ELEMENT (COST Eps EXPENSIVE > 200)
  Eps FURNITURE Eps ALL BIG ROOM := ELEMENT
    (LENGTH Eps BIG ROOM * WIDTH Eps BIG ROOM > 400)
  Eps ROOMS Eps HOME;
```

```
HOME PRICE LIST := PRICE LIST;
COST Eps PREFACE ELEMENT Eps PRICES Eps HOME PRICE LIST <--
  COST Eps ALL ELEMENTS Eps FURNITURE Eps
    GUEST ROOM := ELEMENT (USE Eps GUEST ROOM = 'BEDROOM')
  BACKWARD STARTING AT SMALL ROOM := ELEMENT
    (LENGTH Eps SMALL ROOM * WIDTH Eps SMALL ROOM < 150)
  Eps ROOMS Eps HOME;
```

The translation from the source code of Boolean search and select loops into the equivalent explicit loop statements is a fairly involved process. Section 18 is devoted entirely to describing this process. As shown in Section 18, SPL translates statements which select the first element which has some desired property, or all the elements which have that property, or the first element which has that property provided that there exist any elements which have that property.

#### 17.4. IMPLICIT PROGRAM BLOCK STRUCTURE OF EXPLICIT LOOPS

Fig. 17-2 shows a typical implicit program block structure resulting from an explicit loop statement. The loop statement appears in the source code in block A. All the other program blocks in Fig. 17-2 are created implicitly for the processing of the loop statement.

The access chains for some of the element generators may pass through structure-pointing atoms. The structures which contain these atoms are activated during the initial evaluation of the first-order ancestor complexes of the elements to be generated, but the structures containing these atoms do not necessarily have to remain active during execution of the loop. Blocks C1, C2, . . . , Cn show the brief activation of these structures.

#### 17.5. "CYCLE" AND "LEAVE"

A CYCLE statement consists of the word CYCLE, optionally followed by an arithmetic expression. A LEAVE statement consists of the word LEAVE, optionally followed by an arithmetic expression. If the arithmetic expression is omitted, the value 0 is assumed. CYCLE and LEAVE may appear only within loops.

CYCLE statements and LEAVE statements act as special purpose GO TO statements for terminating execution of a cycle of a loop, or for terminating execution of a loop entirely. CYCLE is the same as GO TO which branches to a fictitious location just before the end of the loop. Example:

```
LOOP _____  
DO _____  
    _____  
    CYCLE;  
    _____  
END LOOP;
```

is equivalent to:

```
LOOP _____  
DO _____  
    _____  
    GO TO DUMMY1;  
    _____  
DUMMY1:  
END LOOP;
```

where DUMMY1 is a dummy statement label automatically supplied by SPL. LEAVE is the same as GO TO which branches to a fictitious location just after the end of the loop. Example:

```
LOOP _____  
DO _____  
    _____  
    LEAVE;  
    _____  
END LOOP;
```

is equivalent to:

```
LOOP _____  
DO _____  
    _____  
    GO TO DUMMY1;  
    _____  
END LOOP;  
DUMMY1:
```

where DUMMY1 is a dummy statement label automatically supplied by SPL.

If an expression follows CYCLE or LEAVE, its value is truncated to an integer which must be nonnegative. SPL leaves that many inner nested loops, and then cycles or leaves an outer loop. Example:

```
LCOP _____  
DO _____  
    LOOP _____  
    DO _____  
        CYCLE 1;  
    _____  
    END LOOP;  
    _____  
END LOOP;
```

is equivalent to:

```
LCOP _____  
DO _____  
    _____  
    LCOP _____  
    DO _____  
        GO TO DUMMY1;  
    _____  
    END LOOP;  
    _____  
    DUMMY1:  
END LOOP;
```

In the above example, execution leaves the 1 inner loop, and then cycles the outer loop. It is an error for the truncated value of the expression to be greater than the number of inner nested loops.

The SPL translator converts all implicit loops into their equivalent explicit loop statements. All these loops are counted in the determination of how many inner nested loops to leave, before cycling or leaving an outer loop.

## 17.6. BOOLEAN IMPLICIT LOOPS

The pronoun phrases ANY OF, ALL OF, or NONE OF may appear in a Boolean expression that includes an implicit loop, thereby forming a Boolean implicit loop. Example:

```
IF ANY OF LENGTH Eps ALL ELEMENTS Eps ROOMS Eps HOME = 20
THEN C <-- C + 1;
```

is equivalent to:

```
LOOP FOR ALL DUMMY1 := ELEMENT Eps ROOMS Eps HOME
DO IF LENGTH Eps DUMMY1 = 20
    THEN GO TO DUMMY2
END LOOP;
GO TO DUMMY3;
DUMMY2: C <-- C + 1;
DUMMY3:
```

A local name may be assigned to the elements. SPL automatically reserves the local name, for subsequent use. After execution of the loop, the element (if any) assigned to the local name depends on the pronoun phrase, the Boolean expression, and whether there exist any elements in the complex. Listed below are the translated equivalents of the various Boolean implicit loops.

Example (1) source code:

```
IF ANY OF A Eps ALL B := ELEMENT Eps C = K
THEN <<code 1>>
ELSE <<code 2>>;
```

Example (1) translated equivalent:

```
RESERVE B;
LOOP FOR ALL B := ELEMENT Eps C
DO IF A Eps B = K
    THEN GO TO DUMMY1
END LOOP;
<<code 2>>;
GO TO DUMMY2;
DUMMY1:
<<code 1>>;
DUMMY2:
```

Example (2) source code:

```
IF ALL OF A Eps ALL B := ELEMENT Eps C = K
THEN <<code 1>>
ELSE <<code 2>>;
```

Example (2) translated equivalent:

```
RESERVE B;
LOOP FOR ALL B := ELEMENT Eps C
DO IF ~ (A Eps B = K)
  THEN GO TO DUMMY1
END LOOP;
<<code 2>>;
GO TO DUMMY2;
DUMMY1:
<<code 1>>;
DUMMY2:
```

Example (3) source code:

```
IF NONE OF A Eps ALL B := ELEMENT Eps C = K
THEN <<code 1>>
ELSE <<code 2>>;
```

Example (3) translated equivalent:

```
RESERVE B;
LOOP FOR ALL B := ELEMENT Eps C
DO IF A Eps B = K
  THEN GO TO DUMMY1
END LOOP;
<<code 1>>;
GO TO DUMMY2;
DUMMY1:
<<code 2>>;
DUMMY2:
```

Example (4) source code:

```
IF ~ ALL OF A Eps ALL B := ELEMENT Eps C = K
THEN <<code 1>>
ELSE <<code 2>>;
```

Example (4) translated equivalent:

```
RESERVE B;
LOOP FOR ALL B := ELEMENT Eps C
DO IF ~ (A Eps B = K)
  THEN GO TO DUMMY1
END LOOP;
<<code 1>>;
GO TO DUMMY2;
DUMMY1:
<<code 2>>;
DUMMY2:
```



The word ALL must occur at least once in the access chain for each Boolean implicit loop. Each occurrence of the word ALL indicates another nested loop.

Example (5) source code:

```
IF ANY OF A Eps ALL B := ELEMENT Eps C Eps ALL D := ELEMENT Eps E
  = K
THEN <<code 1>>
ELSE <<code 2>>;
```

Example (5) translated equivalent:

```
RESERVE B;
RESERVE D;
LOOP FOR ALL D := ELEMENT Eps E
DC LOOP FOR ALL B := ELEMENT Eps C Eps D
  DO IF A Eps B = K
    THEN GO TO DUMMY1
  END LOOP
END LOOP;
<<code 2>>;
GO TO DUMMY2;
DUMMY1:
<<code 1>>;
DUMMY2:
```

Several Boolean implicit loops may be combined in a single Boolean expression.

Example (6) source code:

```
IF ANY OF A Eps ALL B := ELEMENT Eps C
  = ALL OF D Eps ALL E := ELEMENT Eps F
THEN <<code 1>>
ELSE <<code 2>>;
```

Example (6) first translated equivalent:

```
RESERVE B;
LOOP FOR ALL B := ELEMENT Eps C
DC IF A Eps E = ALL OF D Eps ALL E := ELEMENT Eps F
  THEN GO TO DUMMY1
END LOOP;
<<code 2>>;
GO TO DUMMY2;
DUMMY1:
<<code 1>>;
DUMMY2:
```

Example (6) second translated equivalent:

```
RESERVE B;
RESERVE E;
LOOP FOR ALL B := ELEMENT Eps C
DO LOOP FOR ALL E := ELEMENT Eps F
  DO IF  $\neg$  (A Eps B = D Eps E)
    THEN GO TO DUMMY3
  END LOOP;
GO TO DUMMY4;
DUMMY3:
GO TO DUMMY1;
DUMMY4:
END LOOP;
<<code 2>>;
GO TO DUMMY2;
DUMMY1:
<<code 1>>;
DUMMY2:
```

As can be seen in the above examples, the final assignment of elements to the reserved local names is somewhat erratic. Boolean implicit loops provide a convenient way of performing tests, but an inconvenient way of selecting elements. On the other hand, Boolean search and select loops provide a convenient way of selecting elements, but an inconvenient way of performing tests.

#### 17.7. COUNTING ELEMENTS

SPL has the built-in function COUNT, which counts all the elements of a complex, or a selected subset of those elements. The resulting value is of type UNSIGNED INTEGER. Examples:

- (1) NUMBER <-- COUNT ELEMENTS Eps ROOMS Eps HOME;
- (2) NUMBER <-- COUNT LONG ROOM := ELEMENT (LENGTH Eps LONG ROOM > 20)  
Eps ROOMS Eps HOME;

The translated equivalent of example (1) is:

```
COUNT <-- 0;
LOOP FOR ALL DUMMY1 := ELEMENT Eps ROOMS Eps HOME
DO COUNT <-- COUNT + 1
END LOOP;
NUMBER <-- COUNT;
```

## 18. TRANSLATING BOOLEAN SEARCH AND SELECT LOOPS

### 18.1. DEFINITION OF THE PROBLEM

Section 18 is an extension of Section 17.3, in which Boolean search and select loops were introduced. A Boolean search and select loop appears in SPL source code as an access chain, containing somewhere within it the word ELEMENT followed by parentheses enclosing a Boolean expression. The Boolean expression may be arbitrarily complicated, perhaps itself containing Boolean search and select loops. From this source code, SPL compiles an effective procedure for searching among the elements of a complex, and selecting the first element or all elements for which the Boolean expression has the value TRUE. The only restriction is that the Boolean expression somehow depend on some property of the element or elements it is supposed to select.

This section is written for two audiences. First, it is directed to the programmer writing SPL code. It shows him the expansion of his source code into the effective search and select procedure, written as explicit loop statements. This allows him to resolve any questions about the interpretation of his source code, and to pinpoint any ambiguities or inconsistencies. Second, this section is directed to the person implementing SPL, as a possible means of performing the implementation. The translation process described here has as input SPL source code including Boolean search and select loops, and as output SPL source code from which all Boolean search and select loops have been eliminated. The translation process also detects all ambiguities and inconsistencies, and detects when the Boolean expression does not depend on any property of the elements being searched. One approach to implementing an SPL compiler is to implement compilation of explicit loop statements only, and to include an extra pass which translated implicit loop statements and search and select loop statements into their equivalent explicit loop statements.

The translation process described here uses the type-tree formed from the structure declarations (see Section 8) in conjunction with the source code statement, to determine the appropriate sequence and nesting of the loops so that the required chain of data accesses can be performed. Where several sequences or nesting arrangements of the loops are possible, it shows all possible arrangements and indicates an optimal arrangement, in the absence of statistical information about the data.

The description of the translation process is itself composed of two steps. The first step is the development of a "chart" suitable for computer processing, which characterizes the Boolean search and select loops. The second step is the interpretation of that chart as explicit loop statements in SPL source code, for the next pass of the SPL compiler. The chart is isomorphic to the type-trees of the constructs which participate in the loops, with some auxiliary edges and with directions assigned to all the edges. This collection of type-trees and auxiliary edges is called the "graph" of the loops. It is not suitable for computer processing, but is included as an aid to human comprehension.

The notational conventions used throughout Section 18 are that the upper case letters A, B, C, .... represent local names or type names of constructs which appear in the source code, and that DUMMY1, DUMMY2, .... represent local names, isolated cells, or statement labels automatically supplied by SPL. No declarations are shown in this section; the appropriate declarations can be inferred from the source code. The distinction between local names and type names also can be inferred from their position in the source code. For example, if the source code is

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps H;
```

then A, C, E, and G must be type names and B, D, F, and H must be local names.

## 18.2. EXAMPLES DEMONSTRATING SOME OF THE PROBLEMS INVOLVED IN TRANSLATION

Note the similarity in source code between examples (2) and (3), and between examples (3) and (4).

Example (1) source code:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps H;
```

Example (1) translated equivalent:

```
RESERVE D;
LOOP FOR ALL D := ELEMENT Eps G Eps H
DO IF E Eps D = F
  THEN GO TO DUMMY1
END LOOP;
ERROR; <<required element does not exist>>
DUMMY1:
A Eps B <-- C Eps D;
```

Example (2) source code:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps
H := ELEMENT (I Eps H = J) Eps K Eps L;
```

Example (2) translated equivalent:

```
RESERVE D;
RESERVE H;
LOOP FOR ALL H := ELEMENT Eps K Eps L
DO IF I Eps H = J
  THEN GO TO DUMMY1
END LOOP;
ERROR;
DUMMY1:
LOOP FOR ALL D := ELEMENT Eps G Eps H
DO IF E Eps D = F
  THEN GO TO DUMMY2
END LOOP;
ERROR;
DUMMY2:
A Eps B <-- C Eps D;
```

Example (3) source code:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps
H := ELEMENT (I Eps H = J Eps D) Eps K Eps L;
```

Example (3) translated equivalent:

```
RESERVE D;
RESERVE H;
LOOP FOR ALL H := ELEMENT Eps K Eps L
DO LOOP FOR ALL D := ELEMENT Eps G Eps H
DO IF E Eps D = F
THEN GO TO DUMMY1
END LOOP;
ERROR;
DUMMY1:
IF I Eps H = J Eps D
THEN GO TO DUMMY2
END LOOP;
ERROR;
DUMMY2:
A Eps B <-- C Eps D;
```

Example (4) source code:

```
A Eps B <-- C Eps D := ELEMENT ((E Eps D = F) &
(I Eps H = J Eps D)) Eps G Eps
H := ELEMENT (EXISTS D) Eps K Eps L;
```

Example (4) translated equivalent:

```
RESERVE D;
RESERVE H;
LOOP FOR ALL H := ELEMENT Eps K Eps L
DO LOOP FOR ALL D := ELEMENT Eps G Eps H
DO IF (E Eps D = F) & (I Eps H = J Eps D)
THEN GO TO DUMMY1
END LOOP
END LOOP;
ERROR;
DUMMY1:
A Eps B <-- C Eps D;
```

### 18.3. DEVELOPING A CHART

The translation of Boolean search and select statements into their equivalent explicit loop statements is based on interpretation of a chart. The chart characterizes the loops by describing the various dependencies involved in the search and selection process. There are six types of dependencies, two of which are discussed here, two are discussed in Section 18.11, and two are discussed in Section 18.16.3.

The sequence of accesses described by an access chain starts with some known construct which is identified by its local name. The next access is of the first-order descendant of the known construct, and the next access is of its descendant, etc. In this context, the descendant of a structure-pointing atom is the structure to which it points. Each construct after the known construct is said to "depend for access" on its first-order ancestor. Dependence for access is one of the dependencies shown in the chart.

The selection of one element among the many elements of a complex is based on some property of that element. The properties of an element are the values stored in the atoms within the element. The atoms may be either first-order or higher-order descendants of the element. The element is said to "depend for selection" on some of its descendant atoms. Dependence for selection is another of the dependencies shown in the chart.

Source code from example (1) of Section 18.2 is used in describing the development of the chart. The source code is repeated here, as follows:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps H;
```

A depends on B for access, C depends on D for access, D depends on G for access, and G depends on H for access. D also depends on E and F for selection.

In the chart, each of the names A,B,C,D,E,F,G,H is used as a heading for a row R(i) and for the column C(i) with the same subscript. The chart subsequently may be rearranged so that the names head different rows and columns, but all rearrangements are performed such that a name always heads a row and column with equal subscripts. If name N1 depends on name N2 for access, then the letter A is entered in the chart in the intersection of row R(N1) and column C(N2). If name N1 depends on name N2 for selection, then the letter S is entered in the chart in the intersection of row R(N1) and column C(N2). Fig. 18-1 illustrates the chart for the example source code.

Each row in a properly formed chart contains either no letter A or one letter A. If the name heading the row appears only in the rightmost position of one or more access chains, then the row will contain no letter A. If the name heading the row appears in some access chain as a descendant construct, the row will contain exactly one letter A, because in the trees formed by structures each construct can have only one first order ancestor, and therefore depend on only one other construct for access. A single statement in the source code may contain several access chains which mention different instances of the same type of construct. Although the type names are identical, the different instances are distinguished (by examination of the local names at the rightmost ends of the access chains) and each instance heads a separate row and column in the chart. If any row contains more than one letter A, and if the name heading the row is the type name of a construct, then that type name refers to different instances of the construct. The instances should be distinguished. If any row contains more than one letter A, and if the name heading the row is the local name of a construct, then there is an inconsistency in the source code. If two separate rows  $R(i)$  and  $R(j)$  are headed by identical type names and the letter A is in the same column for both rows, then the two identical type names possibly may refer to a single instance of a construct. A compile-time warning message should be issued. The rows  $R(i)$  and  $R(j)$  and columns  $C(i)$  and  $C(j)$  should be merged if they correspond to a complex or to an atom. But they should not be merged if they correspond to an element. The SPL programmer may want to select different elements of the same complex in several different loops within a single statement.

If any row-column intersection of the chart contains more than one letter (either A or S), or if the main diagonal is not empty, then the source code is inconsistent.

Once formed, there must exist at least one arrangement of the chart (simultaneously rearranging row  $R(i)$  to  $R(j)$  and column  $C(i)$  to  $C(j)$ ) in which all the A's lie in the upper-right triangle. Fig. 18-2 shows such a rearrangement of the chart of Fig. 18-1. This arrangement must exist because SPL structures are trees: the sequence of accesses from ancestor to descendant constructs is mirrored in the chart as a sequence of accesses from the name heading the bottom row (or rightmost column) to the name heading the top row (or leftmost column). The arrangement of all A's in the upper-right triangle is a consequence of the ancestor-descendant relation being nonreflexive. If no such arrangement exists for some particular chart, the source code from which the chart was formed is inconsistent. In the subsequent discussion, the only chart arrangements considered are those in which the A's lie in the upper-right triangle.

The chart is derived in several steps. An original chart is drawn showing all the dependencies for access and dependencies for selection which appear in the source code. In succeeding steps the dependencies which are not relevant to the loops gradually are eliminated from the chart, until finally an irreducible chart is obtained. The equivalent explicit loop statements are determined from an interpretation of this irreducible chart.

The charts following the original chart are derived successively from their predecessors by deleting both a row and its corresponding column if either the row is empty or the column is empty. The process is repeated until no more deletions are possible. Figs. 18-3(a) and 18-3(b) show two steps in reducing the chart of Fig. 18-2. The chart of Fig. 18-3(b) is irreducible.

A row being empty means that the construct does not depend on the other constructs, either for access or selection. The construct is constant relative to the search and select loops; therefore its inclusion in the chart is not relevant to the goal of characterizing the loops. A column being empty means that no other construct depends on this one. While the construct itself is dependent on the result of the search and select loops, its inclusion in the chart is not relevant to the goal of characterizing the loops.

Even for an irreducible chart, several arrangements may be possible without violating the restriction that the A's remain in the upper-right triangle. Fig. 18-4 shows an example.

The interpretation of a letter lying in the upper-right triangle of a chart is that the named construct heading the column can be determined before the named construct heading the row. Each row containing at least one letter S corresponds to an element of a complex for which a search and select loop is needed. Each row containing at least one letter S must have at least one letter S in the lower-left triangle, if the source code is error-free. Otherwise, the selection of the elements could be determined before the elements were accessed, so the search and select loop would be unnecessary. Similarly, if a row containing at least one letter S is deleted during the derivation of an irreducible chart, the search and select loop corresponding to the row is unnecessary, indicating an error in the source code.



#### 18.4. DEVELOPING A GRAPH

Fig. 18-5 shows the development of both the chart and graph for the source code from example (3) of Section 18.2. Fig. 18-5(a) shows the original chart formed from the source code, rearranged so that all the A's lie in the upper-right triangle. Fig. 18-5(b) shows the type-trees associated with the source code. The type-trees are drawn with heavy lines. Also shown in Fig. 18-5(b) are some auxiliary edges drawn with light lines. The auxiliary edges represent the connection between the elements of a complex and the atoms which participate in determining the selection of the elements. A direction is assigned to each of the edges, going from a given construct to another construct on which it depends. Thus the direction always is upward on the edges of the type-trees, indicating that the lower construct depends on the upper construct for access. The direction always is from an element to an atom on the auxiliary edges, indicating that the element depends on the atom for selection.

Fig. 18-5(c) shows the irreducible chart derived from Fig. 18-5(a). After all irrelevant rows and columns have been eliminated, only the central part of (a) remains in (c). Fig. 18-5(d) shows those portions of the type-trees and auxiliary edges which still remain in the irreducible chart (c). The irrelevant portions of (b) were eliminated to form (d). Fig. 18-5(d) is called the graph of the search and select loops generated by the source code.

Each row or column in the chart corresponds to a node in the graph. Each letter A or S in the chart corresponds to an edge in the graph. The letter A corresponds to an edge in the type-tree. The letter S corresponds to an auxiliary edge. If the letter A or S is in the intersection of row R(i) and column C(j), the direction of the corresponding edge is from node i to node j.

One of the requirements for well-formedness of each Boolean search and select loop is that the selection depends on some property of the elements being searched. Except where the source code uses EXISTS (discussed in Section 18.11), this requirement is shown in the graph by requiring that there exist at least one auxiliary edge pointing from the element-node to a descendant node.

### 18.5. INTERPRETING A CHART TO DETERMINE LOOPS

Each row containing at least one letter S corresponds to a Boolean search and select loop. The scope and nesting requirements of the loop are shown by drawing an isosceles right triangle on the chart. The base of the triangle lies on the main diagonal, and the apex includes the leftmost letter S in the row. Fig. 18-6 shows the same chart as Fig. 18-5(c), redrawn with the triangles included. In Fig. 18-6, the D-E loop of Fig. 18-5(d) is seen to be nested within the H-I-J loop. This corresponds with the translated equivalent code in example (3) of Section 18.2.

Since the row headings and column headings appear in the same order, the triangles merely are a geometric way of projecting forward the scope of a loop. A loop determining the selection of an element appears as some S's in the row headed by the name of the element. The maximum scope of the loop is the column containing the leftmost S in the row. The column is projected to its corresponding row by travelling up the column to the main diagonal.

Sometimes when the irreducible chart first is developed, the arrangement indicates nesting of the loops. A rearrangement of the chart may show that nesting actually is unnecessary, but that disjoint loops executed sequentially are sufficient. See Fig. 18-7 for an example. Disjoint sequential loops are more economical than nested loops, and should be used wherever possible. Rearranging the chart is discussed in the sections following Section 18.5.

If the selection of elements is determined entirely by the contents of data atoms (not structure-pointing atoms, or other elements or constructs), then it always is possible to arrange the chart so that the triangles are either disjoint or properly nested. Rearrangement to achieve proper nesting is possible because data atoms terminate their access chains, so there is no constraint preventing a data atom from being shifted upward-leftward in the chart. Fig. 18-8 shows two arrangements of a chart, one with improper nesting and one with proper nesting. Atom F is shifted to achieve proper nesting.

However, if the selection of elements is determined partly by the contents of structure-pointing atoms, proper nesting of the triangles sometimes may not be possible. Proper nesting always is possible if the contents of the structure-pointing atoms are used as data only -- names to be tested and compared with other names. But if the contents of the structure-pointing atoms are used both as data in selecting elements of one complex, and as part of the access chain to another complex which must be searched simultaneously, then proper nesting may not be possible.

Fig. 18-9 shows an example where proper nesting is possible, and Fig. 18-10 shows an example where proper nesting is not possible. In both examples, the content of a structure-pointing atom is used both as data and as part of an access chain.

The impossibility of proper nesting of the triangles can be used to detect an obscure source code error which otherwise would be undetectable. Although the graph in Fig. 18-10 seems to indicate that each selection of an element depends on some property of that element, this actually is not so. The source code has an unnecessary search and select loop. The error may be seen in the source code of Fig. 18-10 by observing that, when D is selected, the content of the structure-pointing atom  $E = \text{DUMMY2}$  = the content of structure-pointing atom I. Therefore,  $F \text{ Eps } E$  could just as well have been written  $F \text{ Eps } I$ . But I is a constant relative to the loops, so  $F \text{ Eps } I$  also is a constant relative to the loops, and there is no basis on which to select an element  $\text{DUMMY1 Eps } H$ . The error is more obvious in Fig. 18-11, where the same source code is used, except that  $F \text{ Eps } E$  is rewritten as  $F \text{ Eps } I$ .

Improper nesting also may arise if the Boolean predicate EXISTS, applied to an element, is used to determine the selection of an element in another complex. This use of EXISTS is discussed in Section 18.11.

Proper nesting not only involves the triangles shown in Fig. 18-8, but also subsidiary triangles with apexes including the other S's in the lower-left triangle of the chart. The chart arrangements of Fig. 18-8 are redrawn in Fig. 18-12, showing the subsidiary triangles drawn with light dotted lines.

#### 18.6. CLUSTERING S's ABOUT THE MAIN DIAGONAL

After each rearrangement of the chart for any reason other than the one discussed here, the chart should be rearranged again to improve the clustering of the S's in the lower-left triangle. Shifting the S's in the lower-left triangle of the chart closer to the main diagonal, has the effect of reducing the number of accesses performed during each cycle of the corresponding loop.

The shifting described here has limited goals, to keep this part of the operation simple. Only minor local performance improvements can be expected from this shifting; other rearrangement techniques described in the following sections produce the major performance improvements.

Fig. 18-13 shows an example of poorly clustered and well clustered chart arrangements. Only rows which do not contain S's are rearranged. The chart is partitioned by the rows which contain S's. Each partition of consecutive rows, none of which contain S's, is rearranged internally. The partition as a whole maintains its same position in the chart. In Fig. 18-13(a) there are two partitions, (H,I,E,J,F,G) and (M,K,N).

In addition to confining rearrangement within a partition, no change is made in the relative order of the columns containing S's. The relative order of columns H, E, M, and N is the same in Figs. 18-13(a) and 18-13(b).

### 18.7. PROPAGATING DEPENDENCY

The original chart formed from the source code does not, in general, have all the A's in the upper-right triangle. If there are errors in the source code, they should be detected as soon as possible, in order to make the error messages most meaningful to the SPL programmer. Therefore the chart should be rearranged immediately to put all the A's in the upper-right triangle, so that a source code error which prevents this rearrangement can be detected before the irreducible chart is derived.

Once the irreducible chart has been derived, the arrangement still may not permit proper nesting of the triangles. Proper nesting always can be achieved by shifting data atoms upward-leftward, as described in Section 18.5.

The question then arises: What other chart arrangements are possible? The first derived arrangement of the irreducible chart may not be the most desirable arrangement. Rearrangement may produce greater efficiency of execution, or a different order in which elements are selected.

An exhaustive search for all valid rearrangements of the chart would be a very expensive computation at compile time, of the order of  $N!$  if there are  $N$  rows or columns. This section describes how to obtain the relevant information without any actual rearrangement, using an invariant property of the chart.

A letter A or S in the chart, say at coordinates  $(i, j)$ , indicates that construct  $i$  depends directly on construct  $j$ . This dependency can be propagated to all the constructs on which construct  $j$  depends directly, etc. Eventually one or more paths are created leading from construct  $i$  to all the other constructs on which it depends, either directly or indirectly.

In this section we are interested in propagating dependencies only to other constructs whose identities already have been determined by access and selection. Accordingly, paths in the upper-right triangle of the chart are restricted to remaining in the upper-right triangle. Fig. 18-14 shows an example of the propagation of dependencies. Arrows in the chart trace the paths of propagation.

A path is initiated from each letter A or S in the chart. The path starts propagation along the column containing the letter.

When propagating along a column  $C(i)$ , follow the column to the main diagonal, and then start propagation rightward along row  $R(i)$ . The presence of other letters in that column is a coincidence which has no effect on the path of propagation.

When propagating along a row  $R(i)$ , start a path propagating along each column  $C(j)$  such that  $i < j$  and such that there is a letter A or S at coordinates  $(R(i), C(j))$ .

#### 18.8. SHIFTING DATA ATOMS TO ELIMINATE UNNECESSARY NESTING OF LOOPS

Fig. 18-15 shows two chart arrangements which differ only in the position of data atom H. In Fig. 18-15(a), the loops are nested unnecessarily, since shifting H downward-rightward permits the sequential loop execution shown in Fig. 18-15(b). Shifting H does not change the order in which elements are selected, but does produce greater efficiency of execution.

This situation can be detected by observing that the path of dependency propagation, starting from the letter S at coordinates (G,H), travels above the upper loop corresponding to row D, yet does not depend on loop D. Therefore data atom H can be shifted downward-rightward.

H is shifted to a new position such that column H is immediately to the left of column C(j), where C(j) is the leftmost column such that there is a letter A or S at coordinates (H,C(j)). In the example, C(j) = column G; column H is shifted immediately to the left of column G, and row H immediately above row G. Finally, H is shifted upward-leftward the minimal number of positions necessary to reestablish proper nesting. Proper nesting must be established for the subsidiary triangles, as well as for the triangles indicating loops. The final upward-leftward shift is not necessary in the example of Fig. 18-15.

#### 18.9. INDEPENDENCE OF LOOPS EXECUTED SEQUENTIALLY

As described in Section 18.5, two disjoint triangles in a chart correspond to two separate search and select loops which are executed sequentially. If the loops are independent, either one can be executed before the other. If one of the loops depends on the other for the selection of an element, then either the dependent loop is executed second or else a wasteful nesting of the loops must be used. These conditions may be determined from the chart as follows. Two independent loops A and B produce two arrangements of the chart with disjoint triangles. In one arrangement triangle A is above triangle B, in the other arrangement triangle B is above triangle A. But one loop dependent on the other produces one chart arrangement with disjoint triangles (the starting assumption of this discussion) and one chart arrangement with nested triangles.

Given a chart arrangement with two disjoint triangles, independence of the loops can be determined from the paths of dependency propagation. The loop corresponding to the lower triangle cannot possibly depend on the loop corresponding to the upper triangle. Therefore the loops are independent if and only if the upper loop does not depend on the lower loop.

Let R(upper) be the row corresponding to the upper loop, and let R(lower) be the row corresponding to the lower loop. Follow the paths of dependency propagation from each of the letters A or S in row R(upper). If any of these paths intersect the main diagonal at coordinates (R(lower),C(lower)), then the upper loop depends on the lower loop.

Fig. 18-14 shows an example of one loop depending on another loop. Fig. 18-16 shows an example of independent loops.

#### 18.10. MUTUAL DEPENDENCY AMONG NESTED LOOPS

After data atoms have been shifted downward-rightward as described in Section 18.8, any nested triangles remaining in the chart correspond to nested loops, where the inner loop depends on the outer loop. The inner loop may depend on the outer loop for access, for the selection of elements, or for both.

If the inner loop depends on the outer loop for access, it is impossible to rearrange the chart such that the relative positions of the two loops are interchanged. Fig. 18-6 shows an example where the inner loop depends on the outer loop for access. The path of dependency starting from the letter A in row D eventually intersects the main diagonal at coordinates (H,H).

If the inner loop does not depend on the outer loop for access, the relative positions of the two loops can be interchanged. The resulting chart arrangement shows disjoint loops which are executed sequentially, if what formerly was the outer loop does not depend on what formerly was the inner loop. Rearranging Fig. 18-14(b) to Fig. 18-14(a) is an example.

The resulting chart arrangement again shows nested loops, if the two loops are mutually dependent. Interchanging the inner and outer nested loops alters the order in which elements are selected. Fig. 18-17 shows a simple example of mutual dependency, and Figs. 18-18 and 18-19 show some more complicated examples.

Mutual dependency of nested loops is detected by a slight modification of the method of following dependency propagation. The method described in Section 18.7 avoids loops in the paths of propagation by restricting all path extensions to the upper-right triangle of the chart. All paths starting from the upper-right triangle must trend downward, so no loops can be formed. Similarly, all paths starting from the lower-left triangle must trend upward, all paths ending in the upper-right triangle must trend rightward, and all paths ending in the lower-left triangle must trend leftward. This is a simple consequence of the fact that vertical paths are directed toward the main diagonal, while horizontal paths are directed away from the main diagonal.

There are two modifications to the method described in Section 18.7. The first is to allow loops in the paths of dependency propagation, by allowing the paths to extend leftward from the main diagonal along rows which contain S's in the lower-left triangle. The second modification is to separate those paths which happen to coincide. Coincident paths are distinguished by redrawing them as smooth arcs, an arc from each letter A or S in a column C(i) to each letter A or S in the corresponding row R(i), for all i. Fig. 18-20 shows some of the previous charts redrawn with smooth arcs.

The chart shows mutually dependent nested loops which can be interchanged, if there exists a closed uniformly-minimal-S path which passes through two or more S's. A minimal-S path from a starting letter A or S to an ending letter A or S is defined as a path from the starting letter to the ending letter, such that no other path passes through fewer S's. A closed minimal-S path is defined as a minimal-S path which starts and ends at the same letter. A closed uniformly-minimal-S path is defined as a closed minimal-S path starting (and ending) at any letter A or S through which the path passes.

Fig. 18-20(c) shows a closed uniformly-minimal-S path. Fig. 18-20(b) shows a closed minimal-S path which is not uniformly-minimal-S. The path starting at the letter S at coordinates (G,E) is minimal-S. But if the other letter S at (D,F) or if either of the A's is considered the starting letter, the path is not minimal-S. In this example, the loops corresponding to rows G and D are mutually dependent, but they cannot be interchanged because loop G depends on loop D for access.

### 18.11. "EXISTS"

The Boolean predicate EXISTS may be used to test for a nonzero value in a structure-pointing atom, or for the existence of an element in a complex. Referring back to the example declaration of Fig. 4-1,

```
IF EXISTS NEIGHBOR := . HCUSE ON LEFT Eps HOME  
THEN GO TO ALPHA;
```

conditionally assigns the locan name NEIGHBOR and branches, if the structure-pointing atom HCUSE ON LEFT contains a nonzero value.

EXISTS may be used in two ways to test for the existence of an element. The first of these,

```
IF EXISTS A := ELEMENT (B Eps A = C) Eps D Eps E  
THEN _____  
ELSE _____
```

prevents the system error exit ERROR from being executed, in the event that the Boolean expression has the value FALSE for all elements of complex D. Local name A is assigned only if the specified element exists.

Fig. 18-21 shows an example of the second way in which EXISTS may be used. This is another form of mutual dependency, where the selection of an element of one complex (element K of complex Q in the example) depends on the existence of a specified element of another complex (element L of complex P in the example). The second element (L) must in turn depend on the first element (K) either for access or selection, in order that there ultimately be an effective selection criterion for the first element. In Fig. 18-21, K depends on the existence of L, and L is accessed through K. In Fig. 18-22, K depends on the existence of L, and the selection of L depends on the contents of atom N belonging to K. In Fig. 18-23, the selection of elements never can be resolved, because the selection of each element depends on the previous selection of the other element.

The letter E has been introduced into the chart in these examples, to indicate that the selection of an element of one complex depends on the existence of a specified element of another complex. The E may be in the lower-left triangle, as in Fig. 18-22(a), or in the upper-right triangle, as in Fig. 18-22(b). The scope of the loop corresponding to the E must be expanded until it includes some other loop with an effective selection criterion. The scope is expanded upward in the chart if the E is in the lower-left triangle, or downward in the chart if the E is in the upper-right triangle. In either case, the column C(j) containing the E is projected to its corresponding row R(j).

Fig. 18-24 shows an example which will be used to describe the method of expanding scopes. A square is drawn on the chart for each E, such that the E is in one corner of the square and the square is bisected by the main diagonal. Say the E is located at coordinates (R(i), C(j)) corresponding to a loop on row R(i). Row R(j) also corresponds to a loop, unless there is an error in the source code.

If there are no E's in row  $R(j)$ , then  $R(j)$  must contain at least one S which is strictly to the left of the square. This guarantees that there is an effective selection criterion, which can be propagated back to row  $R(i)$ . The square should be expanded the minimal amount necessary to achieve proper nesting, and include the S in row  $R(j)$ .

If there are E's in row  $R(j)$ , their squares should be expanded first, and then the given square on row  $R(i)$  should be expanded the minimal amount necessary to include (or coincide with) all the expanded squares on row  $R(j)$ . When expanded, the square on row  $R(i)$  must include at least one column to the left of its original boundaries, unless there is an error in the source code.

Finally, the scope of the loop corresponding to row  $R(i)$  is determined by a square of the minimal size necessary to include any S's in row  $R(i)$  in the lower-left triangle of the chart, and to include all the expanded squares corresponding to E's in row  $R(i)$ . This square, like all the squares described above, must be drawn so that it is bisected by the main diagonal.

An exception to this method is the case where a numeric search and select loop provides the effective selection criterion. Numeric search and select loops depend only on themselves for selection; other loops which depend on them for existence do not necessarily require expansion leftward of their scopes. Fig. 18-25 shows an example, with an N on the main diagonal indicating a numeric search and select loop.

When tracing paths of propagated dependency or searching the chart for closed uniformly-minimal-S paths, E's are treated the same as S's. N's are a special case. Since they lie in the main diagonal, they terminate all paths leading to them. Charts really are not helpful in the translation of numeric search and select loops. Numeric loops can be omitted from the charts if the exception mentioned in the paragraph above is recognized.

#### 18.12. STARTING THE SEARCH AT SOME OTHER ELEMENT

Fig. 18-26 shows an example where the search does not necessarily start at the first element of the complex. The search for element D starts after selecting element G of the same complex. The effect on the relative order in which the loops must be executed is the same as though element D depended for access on element G. Therefore the chart contains the letter A at coordinates (D,G). The graph shows the simulated access as a broken line.



#### 18.13. SOURCE CODE ERRORS NOT DETECTABLE BY CHART

Atheists will be gratified to learn that loop analysis by chart is not omniscient. There are some source code errors for which no detection method has (yet) been developed. Fig. 18-27 shows an example where logically independent Boolean search and select loops have been coded in such a manner that they are mutually dependent. Fig. 18-28 shows an example where one of the Boolean factors in a Boolean term does not depend on any property of the element being selected. A third example, as follows, is self-contradictory.

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G
Eps ELEMENT (~ EXISTS D) Eps H Eps I;
```

Special tests could be devised to detect each of these simple examples of source code errors, but not general tests to detect the same type errors embedded in very complicated source code.

#### 18.14. SPECIFYING ORDER OF EXECUTION

The relative order in which numeric or Boolean search and select loops are to be executed can be specified by the SPL programmer. The order of some or all of the loops in a single statement is specified by unsigned integers, enclosed in parentheses and preceding the word ELEMENT. Fig. 18-29 shows an example.

The loops whose order is specified need not be well-ordered. Several of the unsigned integers may be equal. The order of executing these loops is unspecified with respect to each other, but all these loops must follow any loops specified with a lower integer, and precede any loops specified with a higher integer. The loops whose order is not specified in the source code may be executed before, between, or after the specified loops, subject of course to accessing restrictions.

In the absence of any of the previously discussed constraints, the loops are executed in the order imposed by other considerations in the source code: heirarchy of phrases in parsing an expression, left-to-right order, etc.

#### 18.15. TRANSLATED CODE

Examination of the chart is made for the purpose of translating SPL source code containing Boolean search and select loop statements, into equivalent SPL source code containing only explicit loop statements. The relative positions of triangles in the chart determine the relative order of execution and the nesting of the explicit loops. Other statements appear in the translated equivalent code, as well as the loop statements; Section 18.2 contains some examples. This section describes what other statements are needed, and where they are positioned with respect to the explicit loop statements.

### 18.15.1. SIMPLE LOOPS

Section 18.15.1 describes the code to select a single element of a complex. The selected element is the first for which the Boolean expression in the source code has the value TRUE. There must not be mutually dependent interchangeable loops, and the source code must not contain the word EXISTS. Translation of source code containing mutually dependent interchangeable loops or containing the word EXISTS is described in following sections.

Fig. 18-30 shows example source code of the kind described in this section. The outer triangle in the chart corresponds to the outer nested loop statement. The two inner disjoint triangles correspond to the two inner nested loops, which are executed sequentially. The loop corresponding to the lower-right triangle is executed before the loop corresponding to the upper-left triangle.

The local names of the selected elements are used as bound variables for describing the properties of the elements. But they also may be used in subsequent code in the same manner as any other local names: to name instances of constructs (elements, in this case) whose identities already have been determined. Therefore these local names are reserved outside the outermost loop statement.

The code within the scope of each of the loop statements consists of all the inner nested loop statements (if any), followed by the Boolean test. If the Boolean test is successful, a branch is executed to code outside the scope of the loop. Immediately following the end of the loop is an ERROR statement, indicating a programming error if none of the elements have the property specified in the Boolean test. Following the ERROR statement is the branch destination for the Boolean test, and then whatever subsequent code is appropriate. The executable code of the source statement (A Eps B <-- C Eps D in the example) follows the last outermost loop statement.

### 18.15.2. MUTUAL DEPENDENCY FOR SELECTION

If several nested loops are mutually dependent in a manner such that the selection of elements from each loop depends on the selection of elements from all the other loops, yet none depends for existence on any of the others, then the Boolean tests and ERROR statements of the loops are merged. Fig. 18-31 shows an example. The two Boolean tests are "anded" together inside the innermost nested loop, and only a single ERROR statement occurs outside the outermost nested loop.

The Boolean test corresponding to the innermost loop is executed before the Boolean test corresponding to the outermost loop. This is the same order of execution as the order shown in Fig. 18-30. It reduces possible side effects resulting from executing the tests.

### 18.15.3. "EXISTS" AS A SELECTION CRITERION

If the selection of an element of one complex depends on the existence of a specified element of another complex, there is no Boolean test for existence. The only Boolean test within the nested loops is of the effective selection criterion for the element of the second complex. Except for the absence of one Boolean test, the translated equivalent code is the same as for mutual dependency with interchangeable loops.

Fig. 18-32 shows an example using EXISTS. The only Boolean test inside the innermost nested loop is for the selection of element L. If the test succeeds, execution branches outside both loops, thereby effectively selecting element K. Fig. 18-33 shows a more complicated example.

### 18.15.4. CONDITIONAL STATEMENTS USING "EXISTS"

If a conditional statement tests for the existence of a specified element, the code for the ELSE condition substitutes for the ERROR statement following the last outermost loop. In other respects the translated equivalent code is the same as previously described. Fig. 18-34 shows an example.

### 18.16. SELECTING ALL ELEMENTS

#### 18.16.1. INTERPRETATION OF THE WORD "ALL"

The word ALL appearing in an access chain implies the existence of a loop for sequencing over the elements of a complex. If no loop would be compiled in the absence of the word ALL, then SPL compiles a loop specifically in response to recognizing the word ALL. This is called an implicit loop. It is described in Section 17.2.

But if the word ALL is applied to elements chosen by a Boolean search and select loop, SPL does not compile another loop in response to recognizing the word ALL. The loop which performs the selection of elements also is used to sequence over all the selected elements. The scope of the loop is expanded to include all the operations (accesses, tests, stores, other loops, etc.) which depend on the elements selected by the loop.

Fig. 18-35 shows a simple example of Boolean search and select loops, where all the elements are selected rather than just the first element. Although the triangles in the chart are disjoint, the loops are nested. The local names of the elements are not reserved (as they are in Fig. 18-30), there are no ERROR statements, no branches, and the executable code is inside the innermost nested loop.

#### 18-16-2- RESTRICTIONS ON THE USE OF "ALL"

Some uses of the word ALL in access chains are intrinsically meaningless. These source code errors occur in situations like the following.

First, observe that:

```
(1)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D = F Eps G) Eps H
      Eps G := ELEMENT (I Eps G = J) Eps K Eps L;
```

is completely synonymous with:

```
(2)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT
      (E Eps D = F Eps G := ELEMENT (I Eps G = J) Eps K Eps L)
      Eps H Eps G;
```

Next, modify the source code to select all elements G. Then:

```
(3)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D = F Eps G) Eps H
      Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L;
```

is completely synonymous with:

```
(4)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT
      (E Eps D = F Eps G := ELEMENT (I Eps G = J) Eps K Eps L)
      Eps H Eps ALL G;
```

In examples (3) and (4), the word ALL appears in the access chain for C. The access chains for C and F coincide, starting at G. But to the left of G, the access chains are distinct. Examples (3) and (4) are not synonymous with:

```
(5)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT
      (E Eps D = F Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L)
      Eps H Eps G;
```

Example (5) is meaningless; the source code is in error.

By modifying the source code of example (5) into a Boolean implicit loop, the source code once again is meaningful:

```
(6)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D =
      ANY OF F Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L)
      Eps H Eps G;
```

which is completely synonymous with:

```
(7)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D = ANY OF F Eps ALL G) Eps H
      Eps G := ELEMENT (I Eps G = J) Eps K Eps L;
```

Further modification of examples (6) and (7) may lead to two more errors. The word ALL in example (6) or (7) cannot be moved to precede the other occurrence of the letter G, as in:

```
(8)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D = ANY OF F Eps G) Eps H
      Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L;
```

The error in example (8) is similar to the error in example (5). Example (8) has no word ALL in the access chain for the Boolean implicit loop on F, since the access chains for C and F do not coincide until G.

The other error arises if the source code of example (6) or (7) is modified so that the word ALL precedes both occurrences of the letter G, as in:

```
(9)  A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D = ANY OF F Eps ALL G) Eps H
      Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L;
```

At most one word ALL can be applied to a single local name for elements, within a single statement. In example (9), the two occurrences of the single local name G each are preceded by the word ALL. It is meaningless to attempt to use more than one criterion for the selection of elements G.

However, the elements of a single complex may be selected by several criteria, if the resulting selections are assigned different local names. Example:

```
(10) A Eps PREFACE ELEMENT Eps B
      <-- C Eps D := ELEMENT (E Eps D =
      ANY OF F Eps ALL X := ELEMENT (Y Eps X = Z) Eps K Eps L)
      Eps H Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L;
```

### 18.16.3. REPRESENTING "ALL" IN CHART AND GRAPH

As a computational aid in translating the word ALL into its equivalent code, using the chart appears to be of marginal benefit. Once a chart has been developed as described in the preceding sections, with the word ALL ignored, the modifications necessary to account for the word ALL can be computed in a straightforward manner by direct examination of the source code.

However, both the graph and the chart are used in this section to help describe the required modifications. Fig. 18-36 shows the graph and several charts of the example source code used in this section. The example has three occurrences of the word ALL, for selecting all elements named P, T, and AI. Each occurrence of the word ALL must be distinguished. We will do so by assigning them subscripts: ALL(a), ALL(b), and ALL(c).

The graph shown in Fig. 18-36(a) depicts the entire source code statement, rather than just the loops involved. The various relations or operations upon the constructs have been superimposed on the graph, to help clarify the complicated processes described in the source code. Each occurrence of the word ALL is included as a separate node on the graph, as though it were part of the access chain for the descendant constructs. The elements selected by the loops, P, T, and AI, are above the nodes labeled ALL. Each element selected and assigned local name P, T, or AI causes processing to be performed on its descendant constructs. The word ALL causes selection of many instances of Q, V, and AJ as well as many instances of the constructs directly descending from ALL.

Each occurrence of the word ALL also is included in the original chart, shown in Fig. 18-36(b). The dots in the chart are just a visual aid. The original chart is rearranged in Fig. 18-36(c), so that all the A's lie in the upper-right triangle. The error described in example (9) of Section 18.16.2 would show in the chart as two rows headed by the word ALL, both containing A's in the same column.

Fig. 18-36(c) also shows some paths of dependency propagation, as described in Section 18.7. The paths are drawn with solid lines. Only those paths are shown which intersect the main diagonal at a row and column headed by an occurrence of the word ALL. Each intersection of these paths with the main diagonal corresponds to some construct which depends on all the specified elements of a complex being selected, rather than just one of the elements of the complex. If the path bends downward (going from row to column) only at letters A, then the construct is accessed through the word ALL. The upper-leftmost such constructs in the chart are the leftmost constructs of their respective access chains.

For example, consider the path starting at the main diagonal at coordinates (M,M). The path bends downward at the letter A at coordinates (M,L). At row L it diverges into two paths. One of these paths leads to the main diagonal at coordinates (ALL(b),ALL(b)) only through A's. Therefore there is an access chain from construct M to word ALL(b). The other path bends downward at the letter S at coordinates (L,N). Therefore the other path does not correspond to an access chain.

Since there is no path which bends downward only at A's and which leads to the main diagonal at coordinates (M,M), construct M must be leftmost in its access chain. This can be verified by examination of the source code of Fig. 18-36 and the graph in Fig. 18-36(a).

So far, we have used the chart only to find the leftmost constructs or all access chains which pass through the word ALL. It would be equally easy to do this by direct examination of the source code. For each of these constructs, a digit 1 is marked in the chart at the intersection of the row headed by the word ALL and the column headed by the construct name. Light vertical broken lines have been drawn in the columns as a visual aid.

Next, for each digit 1 in a row, mark a digit 2 in the same row, in the column of each construct which participates in the same expression. For example, row ALL(b) contains a digit 1 in column C. Referring to the source code, C is a member of the expression:  $A \leftarrow 2 + C * AD$ . This is easier to see in the graph, Fig. 18-36(a). Constructs A, 2, and AD participate in the same expression as C. Therefore a digit 2 is marked in row ALL(b) in columns A, 2, and AD. If the row-column intersection already contains a digit, then the digit 2 is not marked. We now have marked each row headed by the word ALL with a digit 1 for the leftmost member (call it "LMM") of each access chain passing through the word ALL, and with a digit 2 for each construct participating in the same expression as LMM.

Next, the irreducible chart is derived, as in Fig. 18-36(d). The optimization methods described in the preceding sections are used to find the best chart arrangement, and the triangles are drawn on the chart. The digits 1 and 2 are considered to be significant when deriving the irreducible chart, but are ignored when tracing paths of dependency propagation.

Each occurrence of the word ALL depends for access on the elements or some complex, as shown by the arrows in Fig. 18-36(d). For example, ALL(b) depends on the elements named T. The triangle corresponding to row T must be expanded upward-leftward enough so that it includes the leftmost digit 1 or 2 in row ALL(b). The triangles corresponding to rows P and AI also must be expanded, and proper nesting must be maintained.

Expansion starts with the upper-leftmost of these triangles, corresponding to row P. It is expanded enough to include the leftmost digit in row ALL(a). The next triangle to be expanded corresponds to row T. It must be expanded all the way to the upper-left corner of the chart. This forces the triangle corresponding to row Y to be expanded also, in order to maintain proper nesting. Finally, the triangle corresponding to row AI is expanded all the way to the upper-left corner of the chart, in order to include the digit 2 at coordinates (ALL(c),C). The resulting expanded triangles are shown in Fig. 18-36(e).

### 18.17. EXTENSION OF SOURCE CODE SYNTAX FOR BOOLEAN SEARCH AND SELECT LOOPS

The strongest criticism of the source code syntax is that the desired operations are not immediately obvious to a person reading the source code. Long strings of code describing the selection criteria have the visual effect of separating the access chain. Operands which are logically related in an expression appear physically distant on the printed page.

To some extent this problem is unavoidable where many complicated operations are described in a single statement. For example, the problem arises in ordinary mathematical notation, such as the polynomial

$$3 \cdot X_1 + 25 \cdot X_5 \cdot (5 \cdot X_4 \cdot X_3 + X_1 \cdot (7 \cdot X_2 + X_4) \cdot (X_2 - 3 \cdot X_1) + 2) - 5 \cdot X_2$$

The terms  $3 \cdot X_1$  and  $-5 \cdot X_2$  are closely related in the logical sense, but physically distant.

The problem in SPL can be relieved somewhat by an extension of the syntax, to allow an optional alternative form of writing Boolean search and select statements. The selection criterion may be assigned a name, and then the name defined following the remainder of the statement.

Example:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps H;
```

also may be written as:

```
A Eps B <-- C Eps D := ELEMENT (TESTD) Eps G Eps H
WHERE TESTD = (E Eps D = F);
```

Using the alternative syntax offers no advantage unless the statement of the selection criterion is lengthy, causing visual separation of the access chain. The alternative syntax introduces an additional name for a bound variable into the source code, which merely increases confusion in simple situations like the example above. However, the alternative syntax can reduce confusion in more complicated situations. The examples below are taken from Fig. 18-19 and Fig. 18-24.

Example (1) source code:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F Eps G := ELEMENT
(H Eps I := ELEMENT (J Eps I = K) Eps L Eps G = M Eps D)
Eps N Eps F) Eps Q Eps R;
```

Example (1) alternative syntax:

```
A Eps B <-- C Eps D := ELEMENT (TESTD) Eps Q Eps R
WHERE TESTD = (E Eps D = F Eps G := ELEMENT (TESTG) Eps N Eps
P)
WHERE TESTG = (H Eps I := ELEMENT (J Eps I = K) Eps L Eps G =
M Eps I);
```



Example (2) source code:

```

A Eps B := ELEMENT (EXISTS C := ELEMENT (D Eps C = E) Eps F Eps B)
  Eps G Eps H
<-- I Eps J := ELEMENT (K Eps J = I Eps M := ELEMENT
  ((EXISTS N := ELEMENT (EXISTS P := ELEMENT (Q Eps P = R Eps N)
    Eps S Eps M) Eps T Eps U) &
  (EXISTS V := ELEMENT (W Eps V = X) Eps Y Eps M)) Eps Z Eps J)
  Eps AA Eps AB;

```

Example (2) alternative syntax:

```

A Eps B := ELEMENT (TESTB) Eps G Eps H
<-- I Eps J := ELEMENT (TESTJ) Eps AA Eps AB
WHERE TESTB = (EXISTS C := ELEMENT (D Eps C = E) Eps F Eps B)
WHERE TESTJ = (K Eps J = L Eps M := ELEMENT (TESTM) Eps Z Eps J)
WHERE TESTM = ((EXISTS N := ELEMENT (TESTN) Eps T Eps U) &
  (EXISTS V := ELEMENT (W Eps V = X) Eps Y Eps M))
WHERE TESTN = (EXISTS P := ELEMENT (Q Eps P = R Eps N) Eps S Eps
M);

```

The essential feature of the alternative syntax is that names are assigned to the tests themselves. Even though the tests are described at the end of the statement, the appearance of the names within the Boolean search and select statement unambiguously identifies each test with the elements selected by the test.

## 19. CONNECTING THE ELEMENTS OF A COMPLEX

Elements of a complex normally are linked together with forward-pointing links only, as shown in Fig. 4-1. This provides the greatest space economy while allowing the number of elements to vary dynamically at run time. It also constrains accesses to being sequential -- elements 1 through 14 must be accessed before element 15 can be accessed. In some cases the insertion and deletion of elements is computationally awkward because only forward links are available.

SPL programmers may declare other methods of connecting the elements, which may be more appropriate to the intended processing applications.

The complex may be dimensioned, in which case the elements occupy consecutive storage. Access to the elements is accomplished by computation of relative locations, rather than by following paths of pointers. A dimensioned complex, all of whose descendant complexes also are dimensioned, is the equivalent of an ALGOL array. All the elements of a dimensioned complex are created simultaneously with the creation of the complex itself, so it is an error to attempt to PREFACE, APPEND, INSERT, or DESTROY any elements. The dimension declaration appears as part of the declaration of the complex. Example:

```
COMPLEX FURNITURE, 5 ELEMENTS, (  
  ALPHANUMERIC ATCM ITEM NAME (10);  
  ATCM COST (1000));
```

Bidirectional linking, CORAL-type linking (alternate backward and upward links), or the normal forward linking also may be declared. Examples:

```
COMPLEX FURNITURE, BIDIRECTIONAL LINKS, (  
  ALPHANUMERIC ATCM ITEM NAME (10);  
  ATCM COST (1000));
```

```
COMPLEX FURNITURE, CORAL LINKS, (  
  ALPHANUMERIC ATCM ITEM NAME (10);  
  ATCM COST (1000));
```

```
COMPLEX FURNITURE, FORWARD LINKS, (  
  ALPHANUMERIC ATCM ITEM NAME (10);  
  ATCM COST (1000));
```

It also is possible to declare a multilevel tree of links, each level having its own linking convention. The approximate number of descendant constructs (either lower-level links or elements) must be declared for all but the highest and lowest levels. Example:

```
COMPLEX FURNITURE, ((CORAL LINKS) (BIDIRECTIONAL LINKS, 7)
  (FORWARD LINKS)), (
  ALPHANUMERIC ATCM ITEM NAME (10);
  ATCM COST (1000));
```

In the above example, the top-level links are of the CORAL type, with as many descendant links as are necessary to ultimately point to all the elements of the complex. The second-level links are bidirectional, each pointing to approximately 7 third-level links. The third-level links point forward only, and each is identified with some particular element (stored consecutively with the element).

Another possibility is to declare an arbitrary number of levels of links. A new level of linking is formed whenever the number of descendant links exceeds the declared average. A level of linking is collapsed whenever the number of descendant links is reduced below the declared average. The formation and collapsing of levels of linking is only approximate; some links may point to slightly more or fewer descendant links than the declared average. All levels of linking must be of the same type. Example:

```
COMPLEX FURNITURE, (CORAL LINKS, 20), (
  ALPHANUMERIC ATCM ITEM NAME (10);
  ATCM COST (1000));
```

If the declaration appears in an inner program block, the declared dimension or the declared average number of descendant constructs may be the contents of some variable whose value was set in an outer block.

## 20. EXTENSIONS TO THE DECLARATIONS

Several extensions to the declarative capabilities in SPL are discussed in this section. The extensions do not allow the declaration of additional types of constructs, but instead enable the previously described declarations to be more concise and better documented. These extensions provide rudimentary concordance (IBM calls it "cross-reference"), abbreviation, macro, and subroutine capabilities for the declarations.

### 20.1. DEFINITIONS

Numbers, strings, and Boolean truth values are called "self-defining constants". SPL allows the definition of "compile-time constants" in terms of self-defining constants and previously defined compile-time constants. Compile-time constants are valid only within the program blocks in which they are defined. Their names must be unique in those blocks. Once defined, they may be used in the same manner as self-defining constants. They assume the types and sizes of the terms used in their definitions, unless declared otherwise. Examples:

```
DEFINE CARD LENGTH <-- 80;
DEFINE PI := REAL <-- 3.1416;
DEFINE PIE <-- 'BREADED GOO';
```

### 20.2. COLLECTIONS

Collections are ordered sets whose members all are defined at compile time. Collections are valid only within the program blocks in which they are declared. Their names must be unique in those blocks.

All the members of a collection must be of the same type. The collection assumes the type of its members, and the size of the largest of its members. The use of the collection name results in the implicit generation of a loop. Each member of the collection, in order, is substituted for the collection name in successive cycles of the loop. The scope of the loop includes every construct which participates in the same expression with the collection name. The scope of the loop is determined in a manner similar to that described in Section 18.16.3.

Example collection declaration:

```
COLLECTION USES ('LIVING'; 'DINING'; 'KITCHEN'; 'BREAKFAST';
  'HALL'; 'BEDROOM'; 'BEDROOM'; 'BEDROOM'; 'BATHROOM'; 'BATHROOM';
  'LAUNDRY');
```

Example use of a collection:

```
USE Eps APPEND ELEMENT Eps ROOMS Eps HOME <-- USES;
```

### 20.3. DECLARATION MACROS

Often several different types of structures will contain constructs which have identical declarations. For example, the declaration of structure type HOUSE TRAILER might contain the same complex ROOMS as the declaration of structure type HOUSE:

```
STRUCTURE HOUSE TRAILER (
  ALPHANUMERIC ATCM LICENSE NUMBER (6);
  ALPHANUMERIC ATCM STATE (5);
  ALPHANUMERIC ATCM COLOR (6);
  ALPHANUMERIC ATCM MODEL (8);
  ALPHANUMERIC ATCM MAKE (8);
  COMPLEX ROOMS (SAME AS ROOMS Eps HOUSE));
```

The words SAME AS indicate that the declaration of ROOMS Eps HOUSE also is to be used as the declaration of ROOMS Eps HOUSE TRAILER.

SPL programmers may choose to declare some types of structures, not so they can create instances of the structure types, but for use as parameterless macros in the declarations of other structure types.

### 20.4. MOLECULES

Often several atoms and complexes will be logically related within a user's program, and yet they may comprise only part of a structure or element. These atoms and complexes may be grouped into a "molecule", either for the purpose of macro declaration or macro call using SAME AS. Example:

```
STRUCTURE HOUSE TRAILER (
  MOLECULE VEHICLE IDENTIFICATION (
    ALPHANUMERIC ATOM LICENSE NUMBER (6);
    ALPHANUMERIC ATOM STATE (5);
    ALPHANUMERIC ATCM COLOR (6);
    ALPHANUMERIC ATCM MODEL (8);
    ALPHANUMERIC ATCM MAKE (8));
  COMPLEX ROOMS (SAME AS ROOMS Eps HOUSE));

STRUCTURE AUTOMOBILE (
  ATCM DRIVER (PERSON);
  COMPLEX PEOPLE IN CAR (
    ATOM OCCUPANT (PERSON));
  MOLECULE VEHICLE IDENTIFICATION
    (SAME AS VEHICLE IDENTIFICATION Eps HOUSE TRAILER);
  ATOM TRAILER (HOUSE TRAILER));
```

If two instances of molecules have identical declarations, and if the molecules do not contain any complexes, then (1) storage assignment statements using  $\leftarrow$ ,  $\leftarrow\leftarrow$ , and  $\longleftrightarrow$  may cause the transfer or swap of the entire contents of one molecule into the other molecule, and (2) conditional statements may test whether the molecules are equal. Example:

```
IF VEHICLE IDENTIFICATION Eps THIS CAR =
  VEHICLE IDENTIFICATION Eps STOLEN CAR
THEN BLOWWHISTLE (LOUD);
```

## 20.5. COMPILE-TIME PROCEDURES

Procedures declared outside a given program block may be called within the declarations of the given block. These procedures are executed at compile time. Therefore the actual parameters can be only self-defining constants or previously defined compile-time constants. The output values of the procedures are compile-time constants. The procedures may create and use isolated cells for internal temporary storage, but they cannot use structures or externally created isolated cells because program execution has not yet begun. Example:

```
DEFINE LINE LENGTH <-- MAX (CARD LENGTH; PRINTER COLUMNS);
```

In the above example, procedure MAX must have been declared in an outer nested program block.

## 21. INPUT/OUTPUT

SPL does not include the specification of formatted input/output procedures, although various SPL implementations may have formatted input/output capabilities. The only I/O capabilities basic to SPL, rather than to a particular implementation of SPL, are the procedures PAGE and PRINT.

PAGE causes form ejection to the top of the next page.

The intent of PRINT is to provide a minimum output capability for debugging use. It permits the printing of constant strings or the contents of any construct. The formats for printing the contents of constructs depend on the declarations of the constructs. The standard formats are:

DECLARATION	PRINT FORMAT
UNSIGNED INTEGER	Enough decimal digits to print all significant figures, with leading zeroes omitted. At least one digit is printed.
INTEGER	Sign followed by UNSIGNED INTEGER.
BOOLEAN	TRUE or FALSE.
ALPHANUMERIC	Leading characters up to the last nonblank. Trailing blanks are omitted. Nothing printed if entire string is blank.
REAL	Signed mantissa followed by signed exponent, the number of digits depending on the implementation.
COMPLEX	Real and imaginary parts, each printed in REAL format, separated by comma and enclosed in parentheses.
LONG REAL	Same as REAL, with more digits of significance.
LONG COMPLEX	Same as COMPLEX, with more digits of significance.
DECIMAL	Same as INTEGER.
pointers (structure-pointing the implementation. Leading zeroes are printed. atoms, links between elements, etc.) and addresses	Octal or hexadecimal unsigned integer, depending on the implementation. Leading zeroes are printed.

If a parameter to PRINT is an atom or isolated cell, only the contents are printed. If the parameter is a structure, complex, element, or molecule, the type names of the atoms, elements, etc. and their contents are printed in tabular form. The order of printout is the order in which they are declared, not necessarily the order in which they are packed into the computer memory.

## 22. IMPLEMENTATION

The following sections of this paper discuss various aspects of the implementation of SPL. Only those aspects are discussed which are peculiar to SPL; the reader is presumed to have a general background in implementing algebraic compilers. One particular implementation is described, which may serve as a guide for subsequent implementations. The implementation was done on the CDC (originally Bendix) G-21 computer.

Translation of SPL source code requires at least two passes, and preferably three or more passes. Users should be given the option of having listed the results of each translation pass, as a debugging aid. Debugging facilities should be available in the source language and in any intermediate languages used during translation.

One translation pass should be dedicated solely to translating implicit loop statements of various types into their equivalent explicit loop statements. Some pass before the final translation into object code is needed to determine the declared type of each local name, since the local name may be used for access earlier in the source code (but later in execution) than its assignment.

At run time the primary core storage of the computer contains the object code of the program and models built from the structure declarations, a stack for local names and isolated cells, a table of structure locations, an auxiliary storage table for the virtual memory, and a large structure storage area. In addition, there is an auxiliary storage area on some direct-access device such as disk or drum.



## 23. MODELS

Declarations of SPL structure types result in the creation of models of the declared structures. The models are created at compile time. They are used at compile time for compiling code to access fields within the structures, and for staying in context with incompletely qualified construct names. The models remain in the computer memory at run time, stored with the compiled program. They are used by the interpretive procedures within SPL: creating, copying, erasing, destroying, and printing the contents of structures and their descendant constructs; linearizing inactive structures for writing to auxiliary storage and reconstituting them when they are read back again; and miscellaneous debugging operations.

Since several constructs belonging to different structure types may have the same declared type name, each declared type name is stored only once and is pointed to by the construct models. The type names are retained at run time for use by PRINT and by the debugging operations.

The model of each declared structure or element type (not including the model of any descendant element types) occupies consecutive storage, with pointers to the models of any descendant element types. The model contains one entry for the structure or element as a unit, and entries for each of the descendant atoms, complexes, and molecules. The model of each molecule is just part of the model of its ancestor structure, element, or molecule, with an additional entry for the molecule as a unit. The model of each molecule occupies consecutive storage.

The model entries for descendant atoms, complexes, and molecules are stored in the order in which they were declared. This does not necessarily correspond to the relative positions of the fields within instances of the declared constructs.

Each instance of a structure, element, or molecule within a structure or element, also occupies consecutive storage. The model shows the relative positions of the fields for the various descendant atoms and complexes. The actual positioning of the fields is determined by an implementation-dependent program, which optimizes the placement for the particular computer hardware. For simplicity in writing the SPL interpretive procedures, the private bookkeeping information is given uniform placement in all structure types, the links are given uniform placement in all element types, and the anchor links and dimension (if any) for each complex are partly standardized (for example, they may occupy the rightmost bits within a word).

The fields in each entry of the models are described below. Some of the displacement and size figures must be expressed as words or bytes and remaining bits.

Structure entries:

- (1) Indication that this is the model of a structure.
- (2) Pointer to the type name of the structure.
- (3) Number of entries in the model for molecules and first-order descendant atoms and complexes.
- (4) Amount of consecutive storage required for the structure.

Element entries:

- (1) Indication that this is the model of an element.
- (2) Number of entries in the model for molecules and first-order descendant atoms and complexes.
- (3) Amount of consecutive storage required for the element.

Molecule entries:

- (1) Indication that this is the model of a molecule.
- (2) Pointer to the type name of the molecule.
- (3) Number of entries in the model for molecules and first-order descendant atoms and complexes.
- (4) Amount of consecutive storage required for the molecule.
- (5) Displacement of the start of the molecule from the start of its ancestor structure or element.

Atom entries:

- (1) Indication that this is the model of an atom.
- (2) Pointer to the type name of the atom.
- (3) Amount of consecutive storage required for the atom.
- (4) Displacement of the start of the atom from the start of its ancestor structure or element.
- (5) Indication of the type of atom: UNSIGNED INTEGER, BOOLEAN, etc.
- (6) Pointer to the constant initial value, if it is a data atom. Initial values are stored with the type names of the declared constructs. Pointer to the model of the structure type, if it is a structure-pointing atom.

Complex entries:

- (1) Indication that this is the model of a complex.
- (2) Pointer to the type name of the complex.
- (3) Displacement of the start of the anchor link from the start of the complex's ancestor structure or element. The position of a field containing the dimension is fixed relative to this displacement.
- (4) Pointer to the code segment for determining the dimension of the complex, if any.
- (5) Count of the number of fields described in (6) below. There is one such field for each level of linking, if the number of levels is fixed. There is exactly one field (6) if the number of levels is variable, and no field (6) if the complex is dimensioned.
- (6a) A bit indicating whether this field is for a single level of linking or for all levels of linking.
- (6b) Indication of the type of links: FORWARD, BIDIRECTIONAL, or CORAL.
- (6c) Average number of descendant links. This number always is 1 for the bottom-level links, which are part of the consecutive storage of the elements. This number always is 0 for the top-level links, meaning as many links as necessary.
- (7) Pointer to the model of the elements of the complex.

#### 24. STACK

Local names, isolated cells, temporary storage for evaluating expressions, and temporary storage for the interpretive procedures are kept in a stack at run time. For each program block, the number of local names, isolated cells, and temporary storage locations for evaluating expressions, can be determined at compile time. The stack expands by this amount when the program block is entered, and contracts by this amount when the program block is left. The stack also expands and contracts a variable amount, depending on the needs of the interpretive procedures, when these procedures are executed.

The stack expands downward in memory, with a register pointing to the current end of the stack. Therefore all entries in the stack for local names, etc., are addressable by some fixed positive displacement from the contents of the register. All the interpretive procedures expand the stack by decrementing the contents of the register, but when they have finished execution they restore the former contents of the register. It never is necessary to access any of the stack entries for local names, etc. while the interpretive procedures are executing, so their alteration of the register does not violate the stack addressing capability. Furthermore, no two of the interpretive procedures ever execute simultaneously, with the exception that the free storage recovery procedure may be called while any of the others are executing. Once called, the free storage recovery procedure runs to completion before relinquishing control, so it does not violate the stack addressing capability of the other interpretive procedures.

The local names declared in any one program block are kept in consecutive storage within the stack, to simplify the execution of a routine which releases all the local names just before program execution leaves the block.

## 25. FIELDS WITHIN LOCAL NAMES

Whenever a local name of an instance of a structure or any of its descendant constructs is valid, the entire structure is active. During this time no part of the structure can be relocated. Therefore the local name can point to absolute addresses of any constructs within the structure.

The local name of any descendant construct has a field pointing to the named construct, and a second field pointing to the private bookkeeping area within the structure. The second field is used for incrementing the activity count of the structure when the local name is assigned, and for decrementing the activity count when the local name subsequently is released. A local name of the structure itself contains the entry number for the structure, in the table of structure locations, in place of the pointer in the first field. The pointer in the second field to the private bookkeeping area is sufficient to address the structure. The entry number must be stored in the local name, so it is available for copying into structure-pointing atoms.

In some special cases it is not necessary to increment the activity count when a local name is assigned. These cases are discussed in Section 26. Whenever a local name is assigned and the activity count is incremented, a one-bit field is set in the local name. The field is reset when the local name is released. The local name may be released any time before program execution leaves the block, either because a RELEASE statement was executed or because a DESTROY statement was executed. This bit is examined by the RELEASE and DESTROY statement processors, to ensure that a local name actually was assigned and to prevent the local name from being released more than once. A second attempt to release a local name causes a run-time error. The bit also is examined by the routine which releases all local names just before program execution leaves the block. Only those local names for which the bit is set are released; the others are ignored by this routine.

## 26. WHEN NOT TO ACTIVATE STRUCTURES

Normally a structure is activated whenever a local name is assigned to it or any of its descendant constructs. Under certain circumstances, activating the structure is not necessary because other code guarantees that the structure will remain active while the local name is valid.

This occurs when:

- (1) within the body of a procedure, a local name is assigned to an actual parameter or a descendant of an actual parameter;
- (2) within a loop, a local name is assigned to an element or to the descendant of an element selected for the current cycle by one of the loop generators;
- (3) within an inner nested program block, a local name is assigned to a construct or the descendant of a construct which was assigned a local name in an outer nested program block, provided that none of the intervening statements are labeled.

In any of the above circumstances, the structure must be activated if a RELEASE or DESTROY statement is applied to any of the constructs between the named construct and its ancestor which already was assigned a local name.

It also is unnecessary to inactivate and then reactivate a structure when reassigning a local name to the next element of a complex, between cycles of a loop -- once again, unless the element may have been destroyed within the code body of the loop.

## 27. BOOKKEEPING FIELDS WITHIN STRUCTURES

Each instance of a structure contains a private bookkeeping area whose location is fixed relative to the beginning of the structure. There are two fields in the private bookkeeping area: the location of the model of the structure, and the current activation count of the structure.

In addition, each complex has several fields. A one-bit field indicates whether or not the complex is dimensioned. If the complex is dimensioned, another field contains the dimension. If the complex is not dimensioned, there are sufficient pointer fields to match the declared linking arrangement. See Section 19 for a discussion of the declarations. These fields are stored in the consecutive memory region of the ancestor structure or element which contains the complex. If the complex is dimensioned, all its elements also are stored in this consecutive memory region. If the complex is not dimensioned, its elements are stored elsewhere in the structure storage area, and the pointers in the consecutive memory region of the ancestor structure or element are called the "anchor" of the complex.

## 28. TABLE OF STRUCTURE LOCATIONS

A single table is used to locate all structures in existence at any given time. When each structure is created, it is assigned an entry in the table. It keeps the entry until it is destroyed, at which time the entry is free to be reassigned. The entry points to the current location of the structure, in primary core storage or in auxiliary storage.

Structure-pointing atoms refer to the structure by containing its entry number. An upper bound must be placed on the number of structures which can exist simultaneously, in order to determine the number of bits required for the field of a structure-pointing atom. This upper bound also may be used to determine the maximum size of the table of structure locations.

There is a tradeoff between space and speed in the design of the table. If the entire table is allocated as a single consecutive region, the access through the table will be very fast, but the entire table space is unavailable for other uses. If several smaller regions are linked together to form the table, the accesses will be slower, but initially at least some of the table space is available to hold structures. Additional regions for the table can be taken from the structure storage area, since they are easily relocated. But once allocated, it is very unlikely that their space can be relinquished later. Only the last region of the table can be freed at any given time, and then only if all its entries happen to be free.

The free entries in the table of structure locations are linked together, to speed the assignment of a free entry to a newly created structure. The free entries also are distinguishable by their contents from the entries in use. During the free storage recovery process, all the entries in the table are scanned. Free entries and entries for structures located in auxiliary storage are ignored. Entries for structures located in primary core storage are used to access the structures, in order to determine whether the structures are active. With careful planning of the table, it is not necessary to dedicate a bit in each entry merely to indicate whether the entry is free.

## 29. STRUCTURE STORAGE AREA

All active structures are located in the structure storage area. Inactive structures may be located either in the structure storage area or in the auxiliary storage area. Whenever a structure is used in the program, it is moved into the structure storage area if it is not there already. The space it requires in the structure storage area is taken from free storage, and the space it previously occupied in the auxiliary storage area is made available.

When free storage in the structure storage area is exhausted, normal program execution is delayed for a free storage recovery pass. During this pass, a sweep is made through all entries in the table of structure locations. The entries are examined for structures which are located in the structure storage area, but whose activity count equals zero. As they are found, these inactive structures are moved out of the structure storage area, and the space they occupied is returned to free storage. When this has been completed, free storage is coalesced in the manner described below, and then normal program execution resumes.

During free storage recovery, the only data being transferred are structures of known types, since each structure contains in its private bookkeeping area the address of its model. This makes free storage recovery a much more orderly process than garbage collection, where all of the structure storage area would have to be searched for random odds and ends of unused storage.

All of the structure storage area is subdivided into "storage area cells" of equal size. The storage area cells are the smallest units of space allocation. They are of the smallest convenient size determined by the computer hardware, such that they can contain the bookkeeping information required for the free storage lists.

There are  $N$  separate free storage lists for contiguous regions of free storage of length 1 cell, 2 cells, 4 cells, ...,  $2^{N-1}$  cells. The regions in any one of these lists point to each other with bidirectional links. Each region on one of these lists also has a field of length  $\lceil \log_2(N) \rceil$  bits, identifying the list it is on. The size of the storage area cell must be adequate to contain the bidirectional links and the field for identifying a free storage list, plus possibly one more bit. This bit indicates whether the storage area cell is free or in use. In computers such as the CDC G-21, which have flag bits in every word, the bit can be located in the cell itself. In computers such as S/360, the bit must be located in a separate table of such bits.

Each region of length  $2^K$  cells begins at an address which is an integral multiple of  $2^K$  cells. For example, a storage area cell in S/360 is 8 bytes long and starts on a doubleword boundary. Each region of length  $2^K$  has a unique "mate" of length  $2^K$ , such that they can be coalesced into a region of length  $2^{K+1}$ . Regions of free storage are coalesced only during free storage recovery passes, after all inactive structures have been moved to auxiliary storage. Coalescing is performed by examining all the regions on a free storage list, starting with the list for the smallest regions. If a region and its mate both are on the same list, they are removed from the list, coalesced into a single larger region, and the new region is placed on its free storage list. In this manner, coalescing is attempted when the probability of both  $2^K$  regions being free is greatest.

Figs. 29-1 and 29-2 show a method of assigning and recovering storage. The method places construct boundaries at integral multiples of  $2^K$ , for the largest possible  $K$  which does not force the fragmentation of large regions of free storage. This method maximizes the probability of being able to coalesce free storage during a free storage recovery pass.

### 30. RECURSIVE GENERATOR

The interpretive procedures within SPI need the ability to process all the descendant constructs of any given input construct. A single generator routine for locating and identifying descendant constructs is called by all the interpretive procedures. The generator has an exit for additional processing peculiar to the procedure which called it. An "exit subroutine" is executed each time a descendant element is located and identified. The generator uses the stack for all its storage, so the exit subroutine may call the generator recursively.

Inputs to the generator are the location of the given construct, the location of the model of the given construct, and the location of the exit subroutine. Outputs from the generator which act as inputs to the exit subroutine are the location of the output construct (either the given construct or any of its descendant elements), and the location of the model of the output construct.

### 31. AUXILIARY STORAGE

This aspect of SPI operates under the assumption that a record of fixed length may be written at any one of a large number of fixed locations on the auxiliary storage device, without requiring the rewriting of all auxiliary storage. Certain IBM tapes, for example, fail in this respect because a new record may be written only at the end of the written portion of the tape.

The auxiliary storage table consists of a single bit for each record position in the auxiliary storage area. The bit indicates whether or not the record position is free.

All record positions in the auxiliary storage area are of the same fixed length. No more than one structure is written on any one record. The structure is linearized before it is written to auxiliary storage, and reconstituted after it is read back from auxiliary storage. If the structure is too large to fit into one record, it is written on several records which possibly are nonconsecutive. Space is reserved in each record for a pointer to a possible successor record.

The chosen length of auxiliary storage records depends on many factors, including the relative speeds of the computer vs. the auxiliary storage device, the fixed cost of each I/O operation, the amount of buffer space available, and the expected statistical distribution of the lengths of the structures in the problem being solved. Storing pointers to successor records in the records themselves, rather than in core memory, is costly only when a structure in auxiliary storage is destroyed. Then the entire structure must be read into core memory buffers, merely to determine which auxiliary storage record positions become free. Presumably this is an infrequent operation, compared with activating and inactivating structures.



## 32. COLLECTIONS

Collections are represented internally as tables which exist both at compile time and at run time. They must be implemented so that new collections can be generated from already existing collections.

## 33. EXTENSIONS AND MODIFICATIONS

SPL lacks two facilities which possibly could greatly extend its usefulness in its intended application areas. First, SPL does not have the ability to process strings of arbitrary length. The string processing described in this paper is restricted to strings of declared dimensions, and the storage space used always is the maximum. Second, SPL structure-pointing atoms are restricted to pointing either to a single declared type of structure, or to any possible type of structure. Very few operations are allowable on structure-pointing atoms which may point to any possible type of structure. In some cases it would be convenient to allow a structure-pointing atom to point to any one of a small number of declared structure types, which have some properties in common. A greater variety of operations could be allowed on these common properties.

At the present time, I do not see how either of these facilities can be incorporated into SPL, without seriously degrading the quality of the object code. Much of the code which now can be compiled would have to be interpreted instead, because of storage allocation requirements in the case of strings, and because of the necessity for detecting structure types in the case of structure-pointing atoms. (In general, detection of structure types is necessary since not all properties of the different structure types are identical.) Also, string processing would require the introduction of garbage collecting into the free storage recovery process. At best, garbage collecting is highly inefficient.

On the other hand, if the amount of data storage required for a particular application is small enough to fit entirely within primary core memory, the indirect addressing and virtual memory features of SPL could be eliminated. This includes elimination of activity counts, the table of structure locations, the auxiliary storage table, and the auxiliary storage area. There no longer would be any distinction between structures and elements of a complex. Nevertheless, the appearance of the SPL source code would remain virtually unchanged.

One such application for SPL is the writing of system monitors. New facilities would have to be introduced, for the processing of blocked data, to allow assembly language subroutines for direct interaction with interrupt registers and the like, and to describe such parallel processing concepts as multitasking. It also would be necessary to segment primary core storage into classes for memory protection, and to create atoms which contain program points for execution.

#### 34. ACKNOWLEDGEMENTS

I would like to express my gratitude to Robert T. Braden for his help in designing the SPL syntax, David C. Cooper for his help with Boolean search and select loops, and Allen Newell for his help with free storage recovery. The suggestions of many other persons, at Carnegie Tech and Stanford Univ., have found their way into this paper.

### 35. BIBLIOGRAPHY

#### Program block structure:

- (1) Naur, P., et al, "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM 6,1 (Jan. 1963), pp. 1-17.

#### Data structures and pointers:

- (2) PL/1 Language Specifications, IBM System/360 SRL Form C28-6571.
- (3) Roberts, L. G., "Graphical Communication and Control Languages", Proc. Second Congress on Information System Sciences, Hot Springs, Va. (1964).

#### Metalanguage notation:

- (4) Gorn, S., "Specification Languages for Mechanical Languages and Their Processors -- A Baker's Dozen", Comm. ACM 4,12 (Dec. 1961).
- (5) Ross, Dan, Box Syntax -- A 2-Dimensional Metalanguage, SLAC CGTM No. 16, Stanford Univ. (June 1967).

#### Associative data processing:

- (6) Rovner, P. D., and Feldman, J. A., The LEAP Language and Data Structure, Report DS-5997, MIT Lincoln Lab., Lexington, Mass. (Jan. 1968).

#### More general list processing languages:

- (7) Standish, T. A., A Data Definition Facility for Programming Languages, Comp. Ctr., Carnegie Inst. of Tech., Pittsburgh, Pa. (May 1967).
- (8) McCarthy, J., et al, LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass. (1962).
- (9) Newell, A., et al, Information Processing Language-V Manual, 2nd ed., Prentice Hall, Inc., Englewood Cliffs, N. J. (1964).

#### Programs which maintain a history of their actions:

- (10) Floyd, R. W., "Nondeterministic Algorithms", J. ACM 14,4 (Oct. 1967), pp. 636-644.
- (11) Ross, Dan, et al, DSM - A Text Editor with Time Reversal Capability, SLAC-PUB-504, Stanford Univ. (Sept. 1968).

#### Compilers with code optimization:

- (12) FORTRAN-IV (H) Programmer's Guide, IBM System/360 SRL Form C28-6602, pp. 62-66.

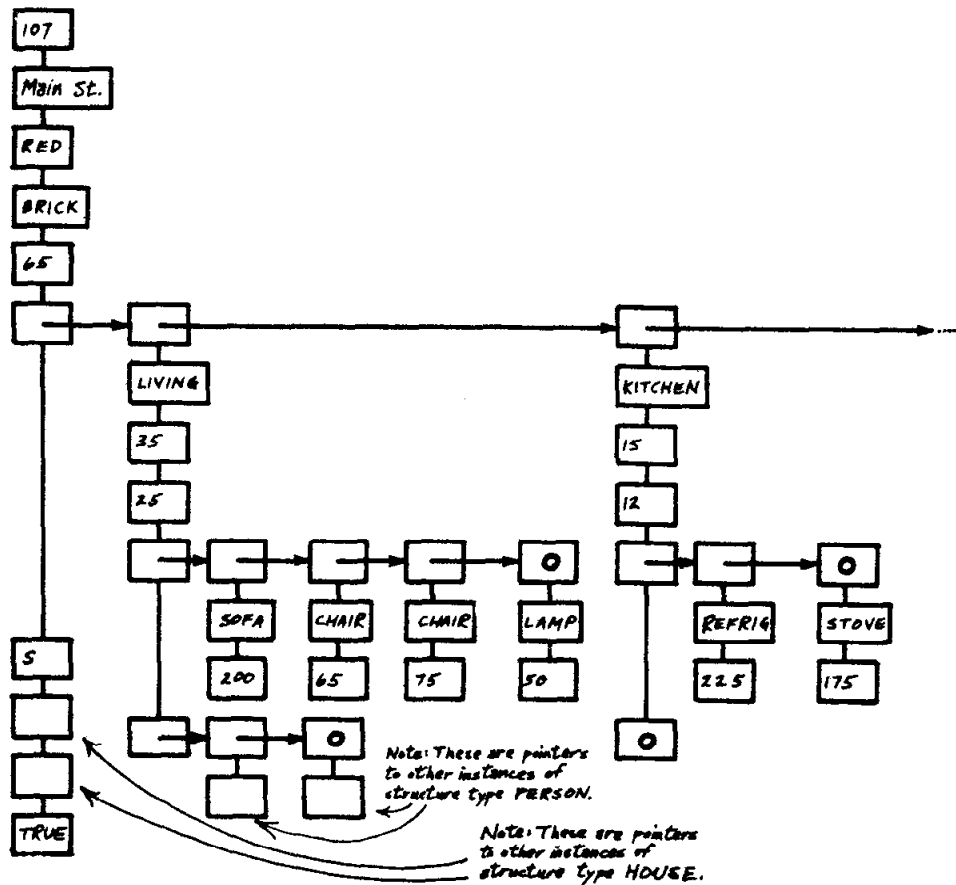
```

STRUCTURE HOUSE (
  ATOM STREET NUMBER (9999);
  ALPHANUMERIC ATOM STREET NAME (20);
  ALPHANUMERIC ATOM COLOR (6);
  ALPHANUMERIC ATOM MATERIAL (5);
  ATOM FRONTAGE (200);
  COMPLEX ROOMS (
    ALPHANUMERIC ATOM USE (10);
    ATOM LENGTH (40);
    ATOM WIDTH (40);
    COMPLEX FURNITURE (
      ALPHANUMERIC ATOM ITEM NAME (10);
      ATOM COST (1000));
    COMPLEX PEOPLE IN ROOM (
      ATOM OCCUPANT (PERSON));
  ALPHANUMERIC ATOM SIDE OF STREET (1);
  ATOM HOUSE ON LEFT (HOUSE);
  ATOM HOUSE ON RIGHT (HOUSE);
  ATOM GARAGE (1));

```

FIG.  
4-1

Example declaration of structure type HOUSE,  
and an instance of a house.



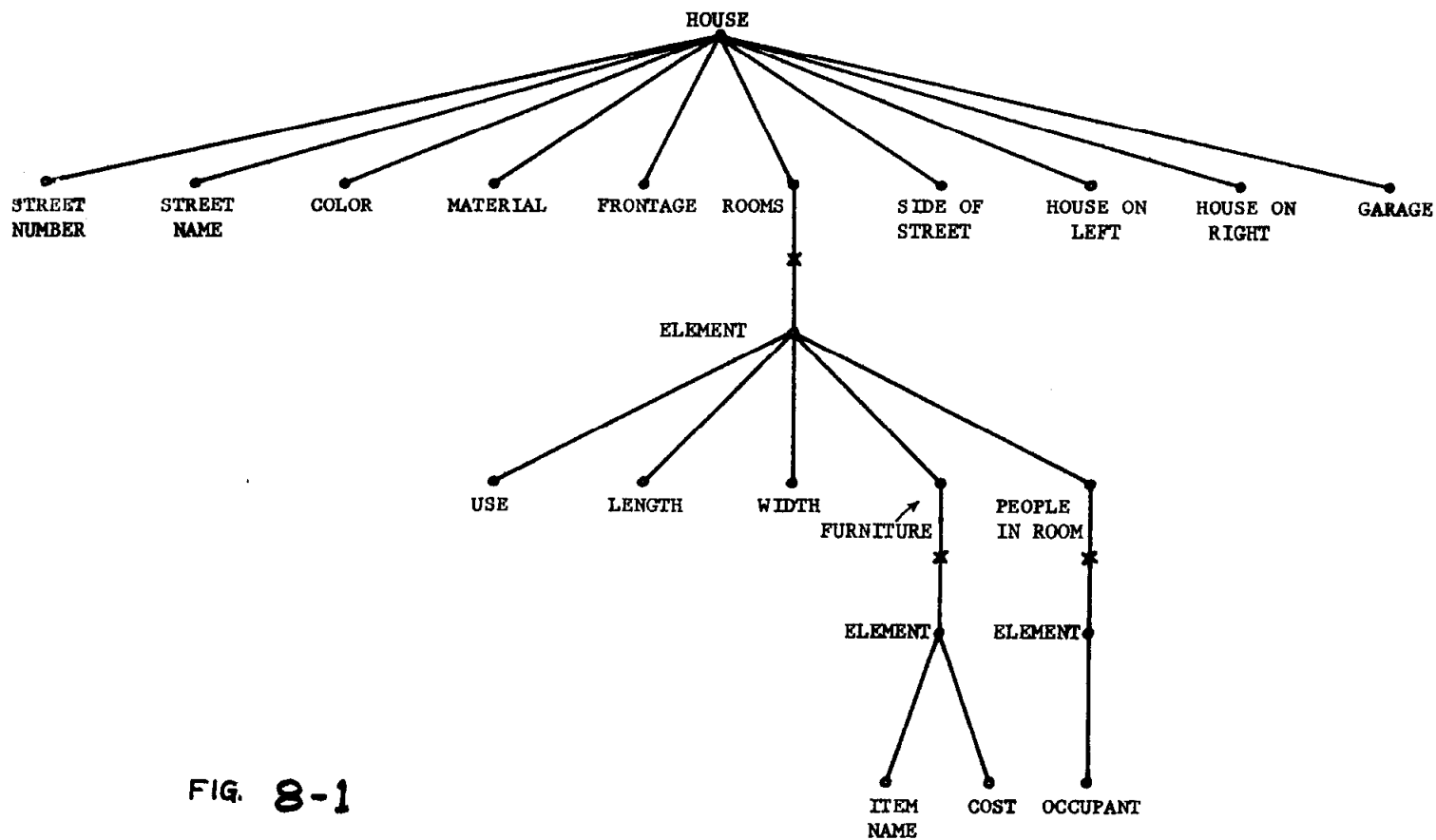
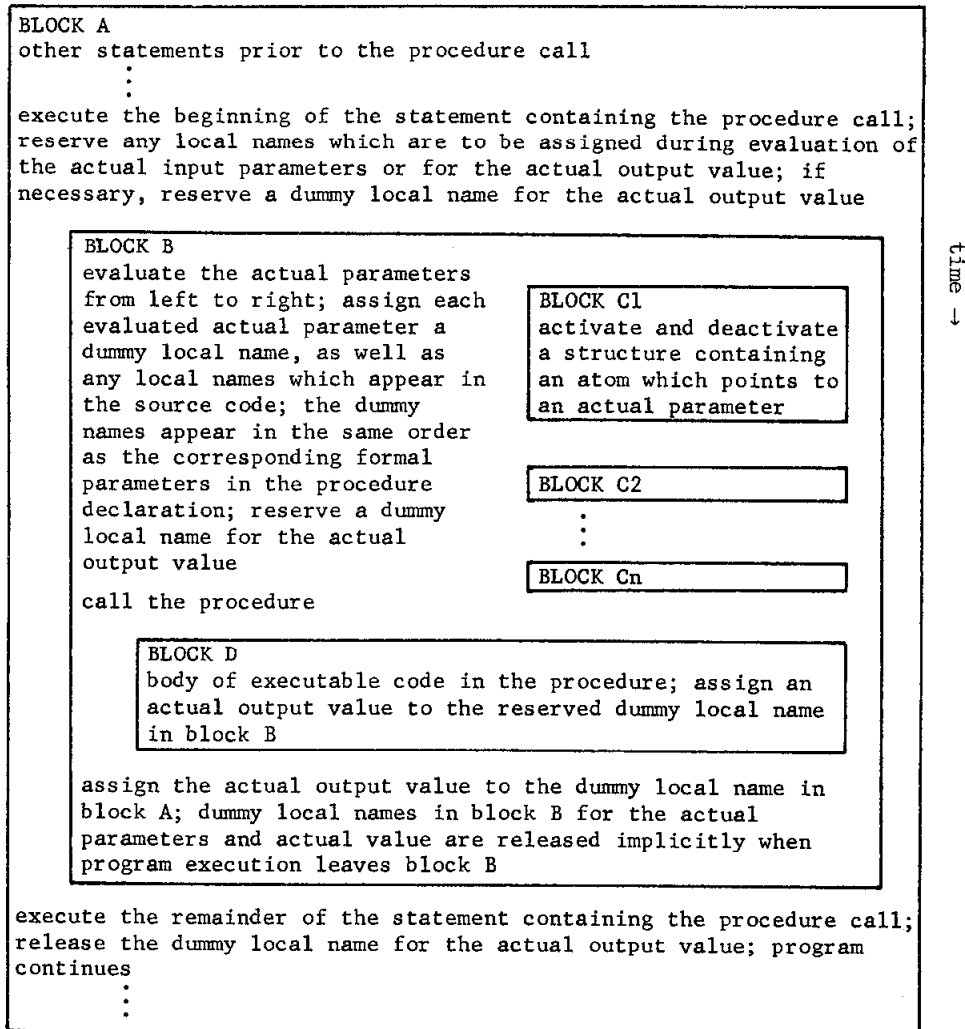


FIG. 8-1

A graph of the type-tree for the example structure HOUSE declared in Fig. 4-1. ROOMS, FURNITURE, and PEOPLE IN ROOM are complexes. A typical element is shown beneath each complex. The X's indicate the separation between complexes and their elements. All the remaining nodes under HOUSE are atoms.



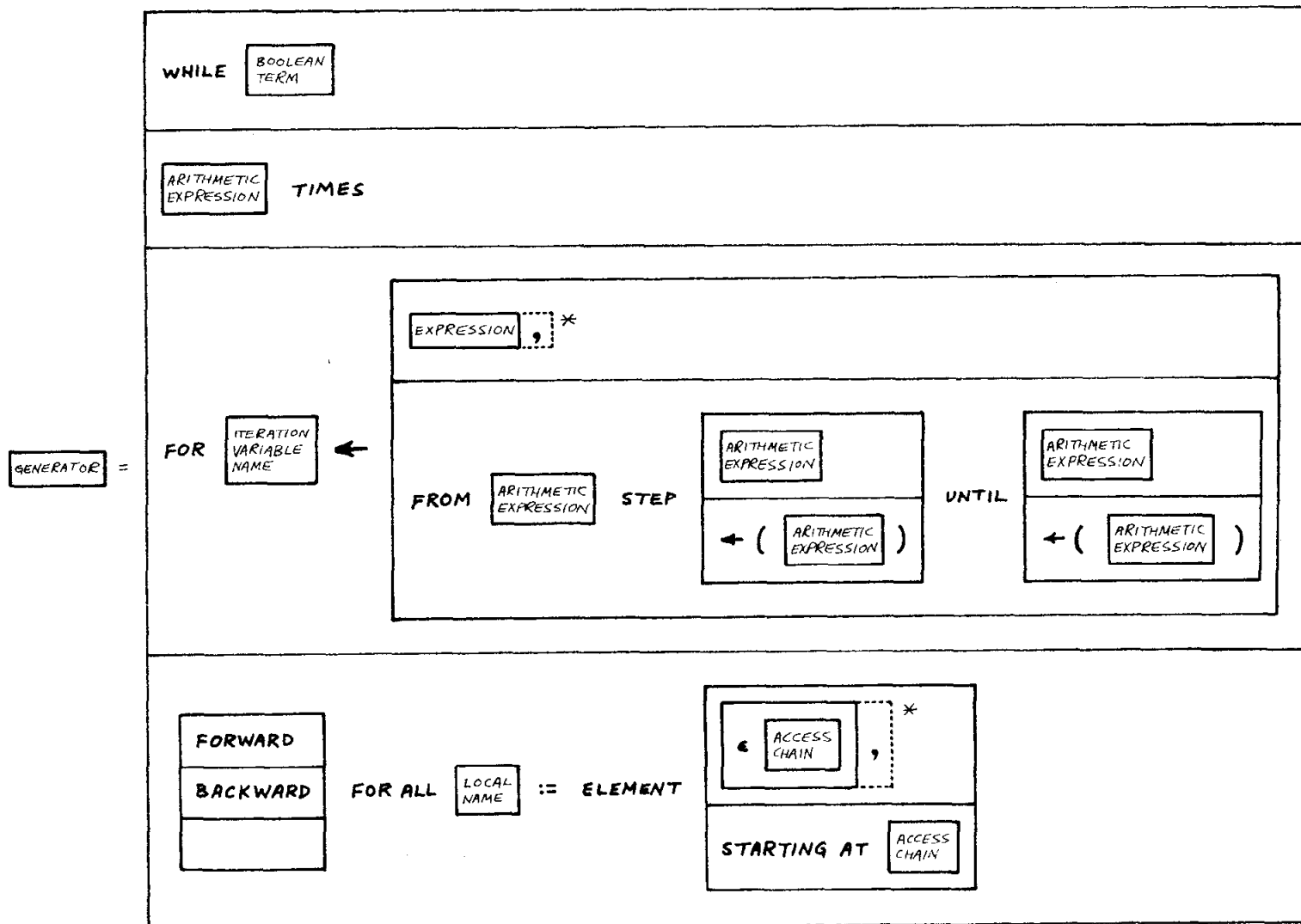
**FIG. 16-1**

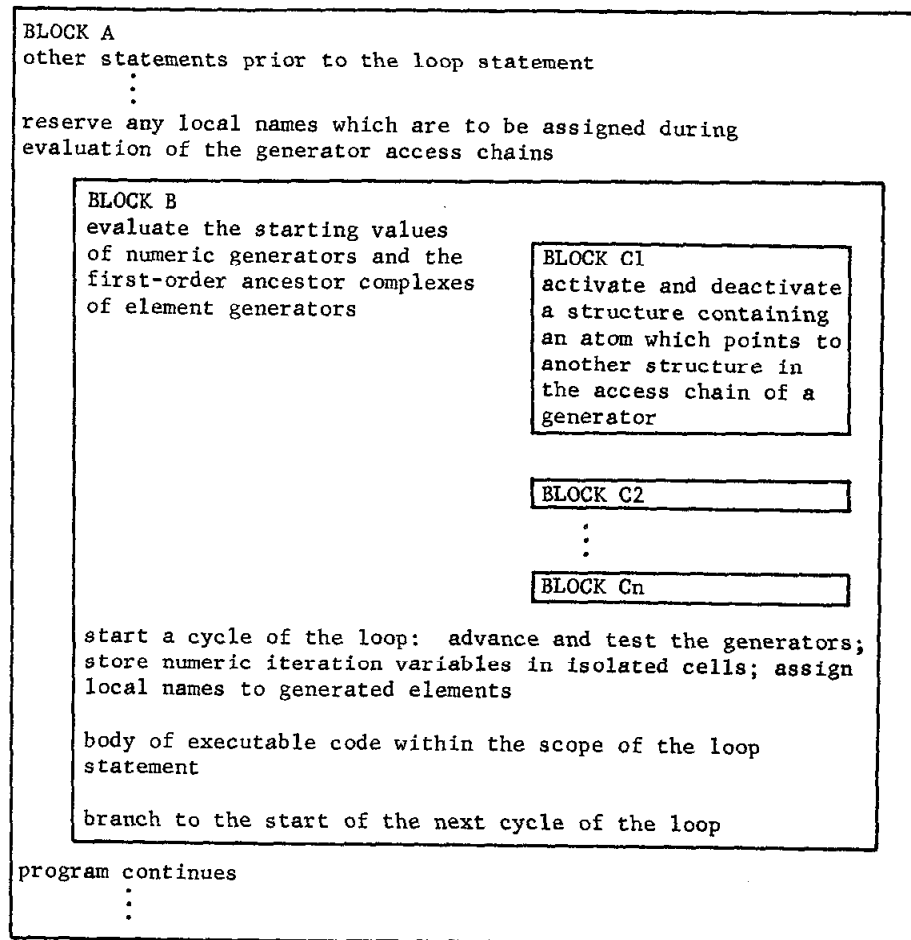
Typical implicit program block structure resulting from a procedure call.

EXPLICIT LOOP STATEMENT = LOOP GENERATOR ; \* DO EXECUTABLE STATEMENT ; \* END LOOP

FIG. 17-1

Syntax of explicit loop statements.





time →

**FIG. 17-2**  
Typical implicit program block structure resulting from an explicit loop statement.



	A	B	C	D	E	F	G	H
A	A							
B								
C				A				
D					S	S	A	
E				A				
F								
G								A
H								

FIG. 18-1

Original chart formed from the example source code:

A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps H;

	A	B	C	E	D	F	G	H
A	A							
B								
C					A			
E					A			
D				S		S	A	
F								
G								A
H								

FIG. 18-2

Rearrangement of the chart so that all the A's lie in the upper-right triangle.

	E	D	G
E	A		
D	S		A
G			

(a)

	E	D
E	A	
D	S	

(b)

FIG. 18-3

Charts derived by successively deleting rows and columns where either is empty. Chart (b) is irreducible.

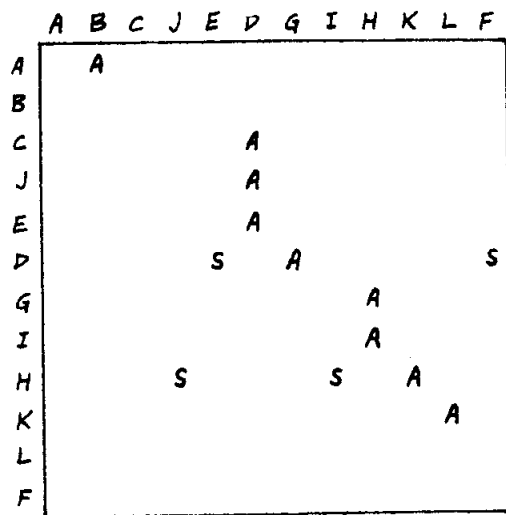
	E	D	F	H	G
E		A			
D	S		S		
F					A
H					A
G				S	

	F	H	G	E	D
F			A		
H			A		
G		S			
E					A
D	S			S	

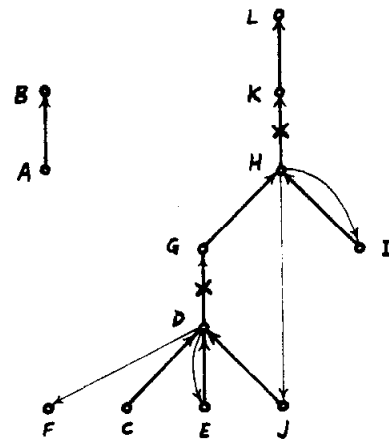
FIG. 18-4

Two arrangements of the irreducible chart derived from:

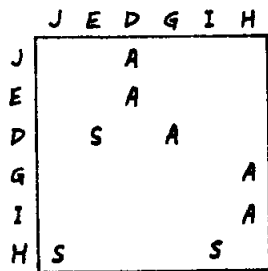
A Eps B <-- C Eps D := ELEMENT  
 (E Eps D = F Eps G := ELEMENT (H Eps G = I) Eps J Eps K)  
 Eps L Eps M;



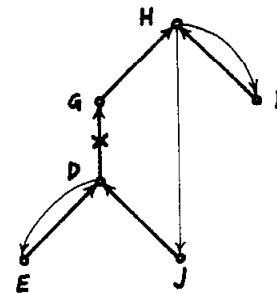
(a)



(b)



(c)



(d)

FIG. 18-5

Development of both chart and graph of:

A Eps B  $\leftarrow$  C Eps D := ELEMENT (E Eps D = F) Eps G  
 Eps H := ELEMENT (I Eps H = J Eps D) Eps K Eps L;

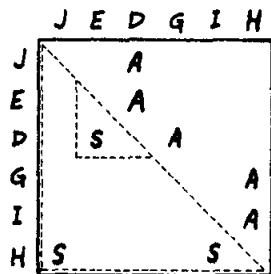


FIG. 18-6

Fig. 18-5(c) redrawn with triangles included to show the loops.

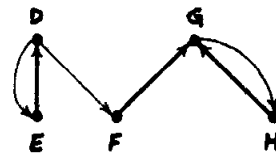
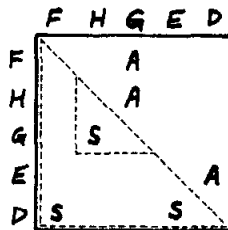
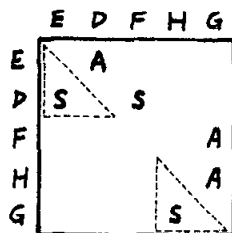


FIG. 18-7

Fig. 18-4 redrawn showing disjoint and nested loop arrangements, and corresponding graph.

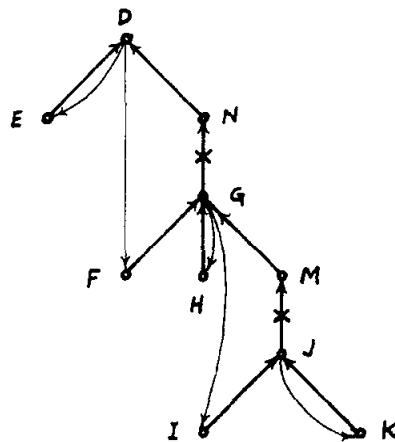
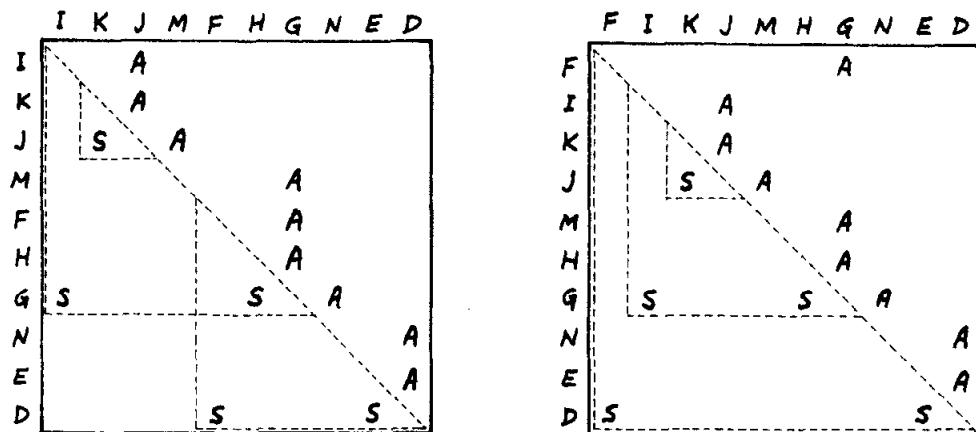


FIG. 18-8

Two arrangements of a chart, one showing improper nesting and the other showing proper nesting. Only the position of F differs between the two arrangements. The graph and source code are applicable to both arrangements of the chart. Source code:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F Eps G := ELEMENT
  (H Eps G = I Eps J := ELEMENT (K Eps J = I) Eps M Eps G)
  Eps N Eps D) Eps P Eps Q;
```

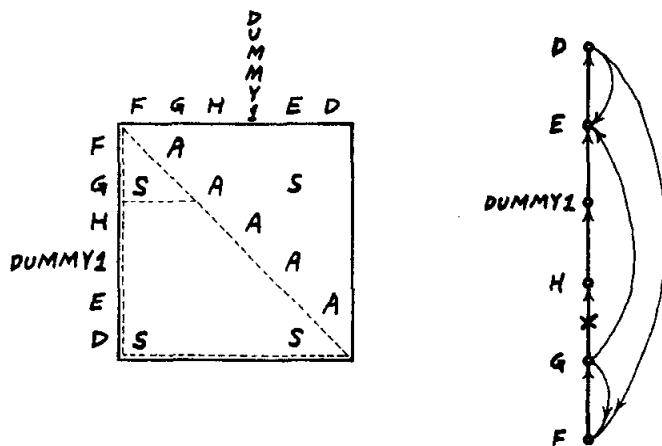


FIG. 18-9

Proper nesting where the contents of a structure-pointing atom is used both as data and as part of an access chain. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  (E Eps D = F Eps G := ELEMENT (E Eps I = F Eps G) Eps H Eps F)
  Eps I Eps J;
```

E and F are structure-pointing atoms, both of which must contain the name of the same structure after the selection has been made. DUMMY1 is the name of the structure in atom E of the selected element D.

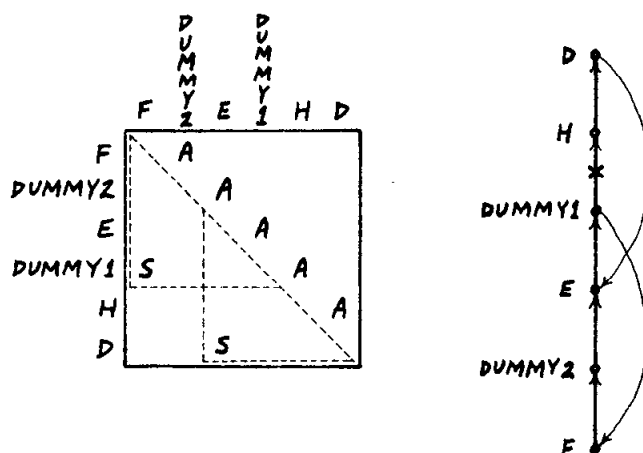


FIG. 18-10

Improper nesting where the contents of a structure-pointing atom is used both as data and as part of an access chain. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  {E Eps ELEMENT (F Eps E = G) Eps H Eps D = I)
  Eps J Eps K;
```

E is a structure-pointing atom, containing the name of structure DUMMY2. Element DUMMY1 is selected on the basis of the contents of atom F. The error in the source code is described in Section 18.5.

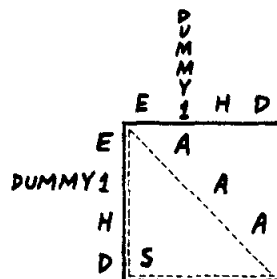
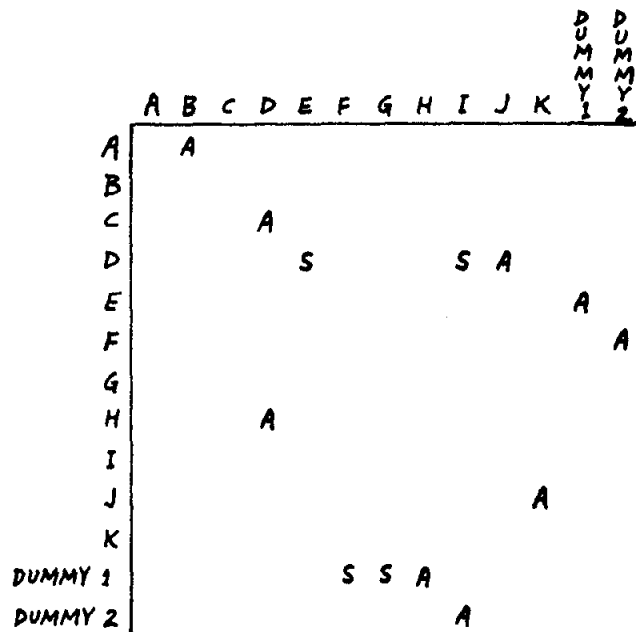


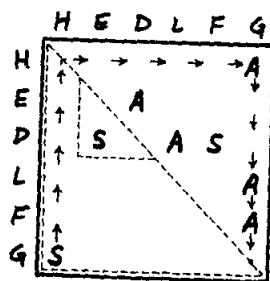
FIG. 18-11

Original chart and irreducible chart of the source code of Fig. 18-10, except that  $F \text{ Eps } E$  is rewritten as  $F \text{ Eps } I$ . Source code:

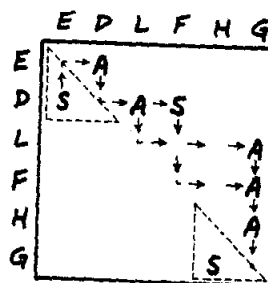
```
A Eps B <-- C Eps D := ELEMENT
  (E Eps ELEMENT (F Eps I = G) Eps H Eps I = I)
  Eps J Eps K;
```

The error is more apparent here, since the original chart shows two loops and the irreducible chart shows only one loop.

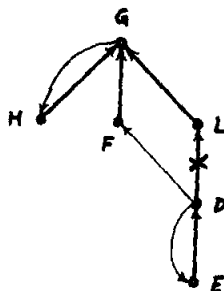




(a)



(b)



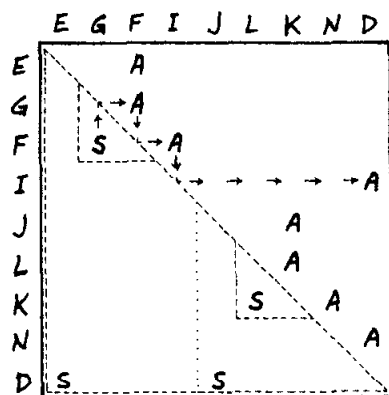
(c)

FIG. 18-15

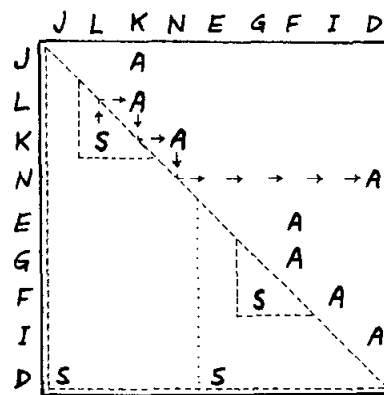
- (a) Nested chart arrangement due to the position of atom H.  
 (b) The upper loop depends on the lower loop via two paths.  
 (c) Graph.

Source code:

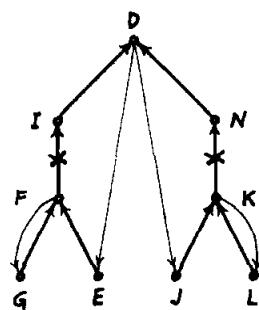
```
A Eps B <-- C Eps D := ELEMENT
  (E Eps D = F Eps G := ELEMENT (H Eps G = I) Eps J Eps K)
  Eps I Eps G;
```



(a)



(b)

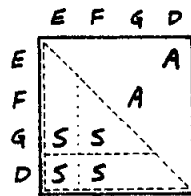


(c)

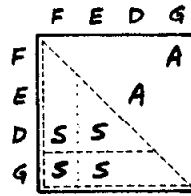
FIG. 18-16

Two chart arrangements and graph showing independent inner loops. Arrows in charts indicate propagation of dependency. Propagation stops at column D, since column D is to the right of the lower loop. Source code:

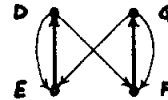
```
A Eps B <-- C Eps D := ELEMENT
  (E Eps F := ELEMENT (G Eps F = H) Eps I Eps D =
   J Eps K := ELEMENT (L Eps K = M) Eps N Eps D)
  Eps P Eps Q;
```



(a)



(b)



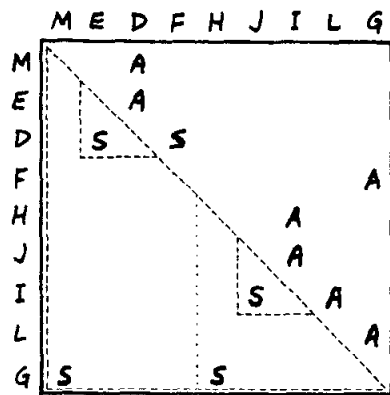
(c)

FIG. 18-17

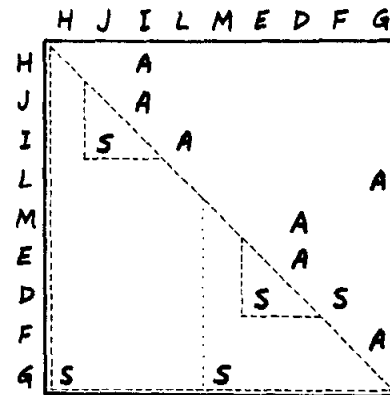
Two chart arrangements and graph showing mutual dependency. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  (E Eps D = F Eps G := ELEMENT (E Eps I = F Eps G) Eps H Eps I)
  Eps J Eps K;
```

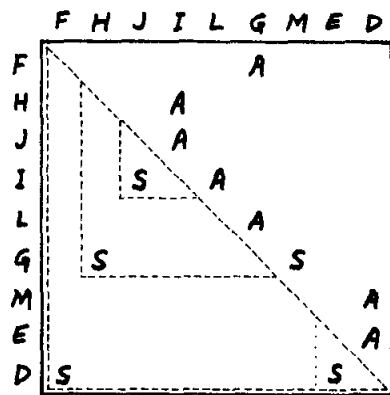




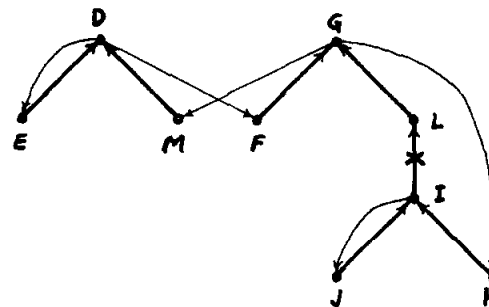
(a)



(b)



(c)

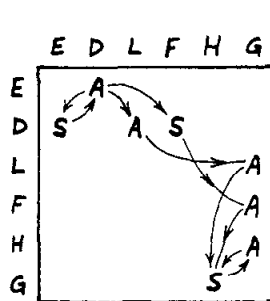


(d)

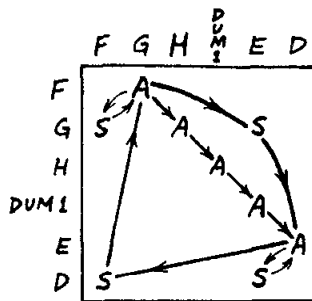
FIG. 18-19

Mutual dependency and independence. There is mutual dependency in the selection of elements D and G. In arrangements (a) and (b) the loop for element G is outermost, and the loops for elements D and I are independent. In arrangement (c) the loop for element D is outermost. Source code:

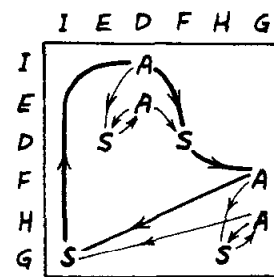
```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F Eps G := ELEMENT
  (H Eps I := ELEMENT (J Eps I = K) Eps L Eps G = M Eps D)
  Eps N Eps P) Eps Q Eps R;
```



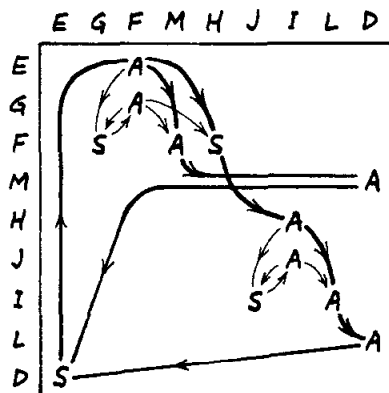
(a) Fig. 18-15(b)



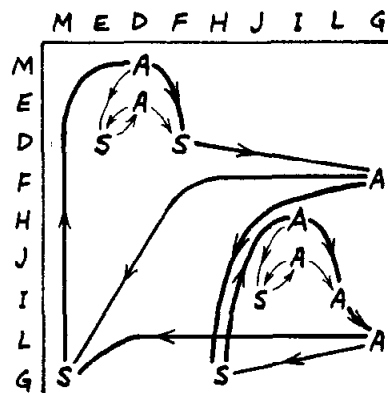
(b) Fig. 18-9



(c) Fig. 18-18(a)



(d) Fig. 18-14(a)



(e) Fig. 18-19(a)



(f) Fig. 18-17(a)

# FIG. 18-20

Some previous charts redrawn, showing detection of mutual dependency. Paths of special interest are emphasized.

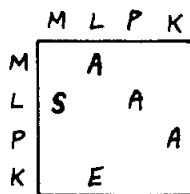
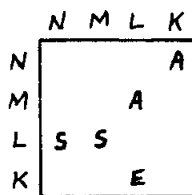


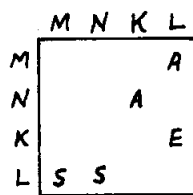
FIG. 18-21

Selection of element K depends on the existence of element L. Source code:

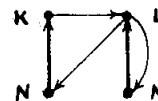
```
H Eps I <-- J Eps K := ELEMENT
  (EXISTS L := ELEMENT (M Eps L = N) Eps P Eps K)
  Eps Q Eps R;
```



(a)



(b)



(c)

FIG. 18-22

Selection of element K depends on the existence of element L. Source code:

```
H Eps I <-- J Eps K := ELEMENT
  (EXISTS L := ELEMENT (M Eps L = N Eps K) Eps P Eps T)
  Eps Q Eps R;
```

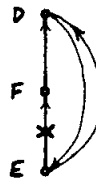
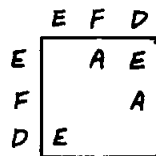


FIG. 18-23

Each loop depends on the previous selection of an element from the other loop. No first selection is possible. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  (EXISTS E := ELEMENT (EXISTS D) Eps F Eps D)
  Eps G Eps H;
```



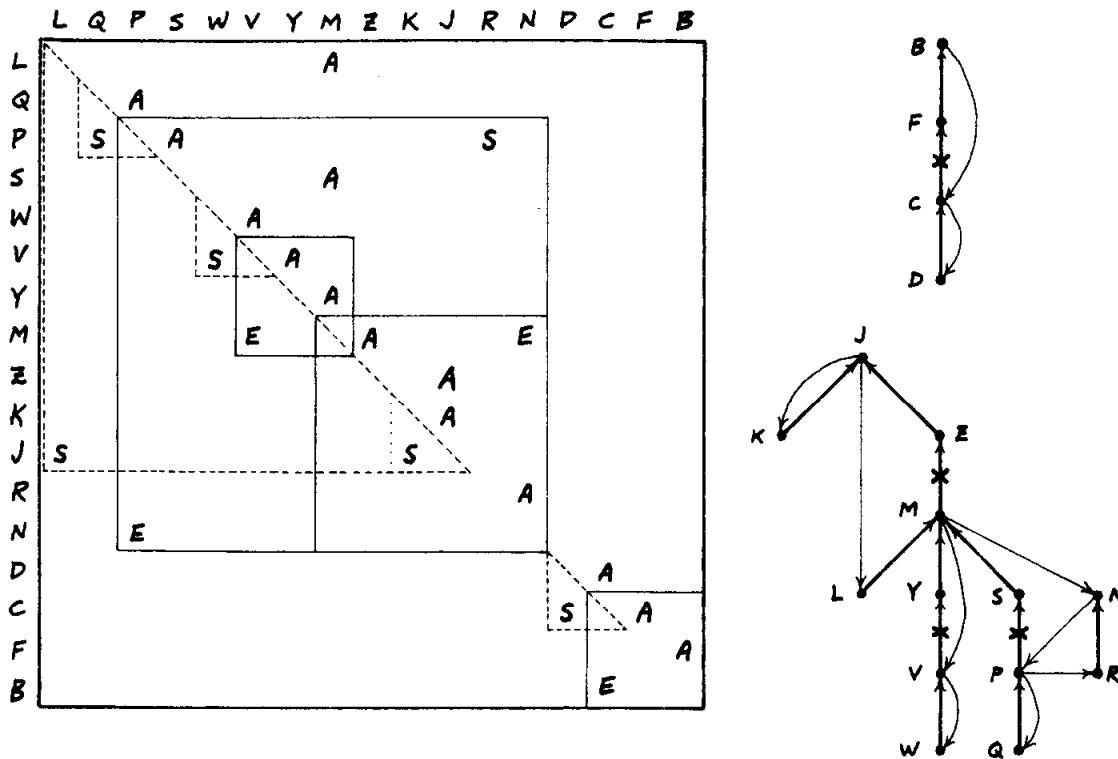


FIG. 18-24

Expanding the scope of loops containing E's. Several other chart arrangements are possible. Source code:

```
A Eps B := ELEMENT (EXISTS C := ELEMENT (D Eps C = E) Eps F Eps B)
  Eps G Eps H
<-- I Eps J := ELEMENT (K Eps J = L Eps M := ELEMENT
  ((EXISTS N := ELEMENT (EXISTS P := ELEMENT (Q Eps P = R Eps N)
    Eps S Eps M) Eps T Eps U) &
  (EXISTS V := ELEMENT (W Eps V = X) Eps Y Eps M)) Eps Z Eps J)
  Eps AA Eps AB;
```

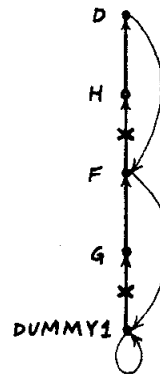
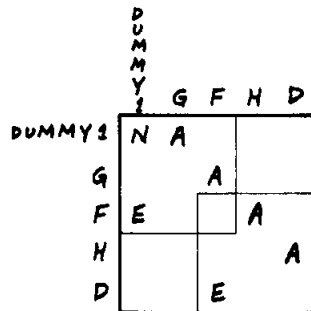


FIG. 18-25

Expanding the scope of loops containing E's. A numeric search and select loop provides the effective selection criterion. Source code:

```

A Eps B <-- C Eps D := ELEMENT
  {EXISTS F := ELEMENT (EXISTS ELEMENT (10) Eps G Eps F) Eps H Eps D)
  Eps I Eps J;

```

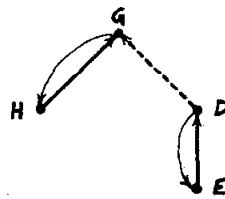
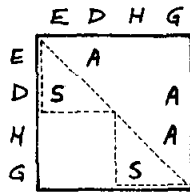
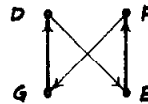
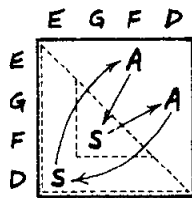


FIG. 18-26

The search for element D starts after selecting element G of the same complex. Source code:

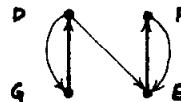
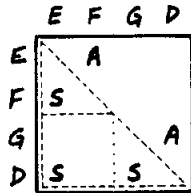
```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F)
  BACKWARD STARTING AT G := ELEMENT (H Eps G = I) Eps J Eps K;
```



**FIG. 18-27**

Logically independent loops coded to be mutually dependent. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  (E Eps F := ELEMENT (G Eps D = H) Eps I Eps J = K)
  Eps L Eps M;
```



**FIG. 18-28**

The second Boolean factor for selecting element D does not depend on any property of D. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  ((G Eps D = H) & (E Eps F := ELEMENT (E Eps F = K) Eps I Eps J = K))
  Eps L Eps M;
```

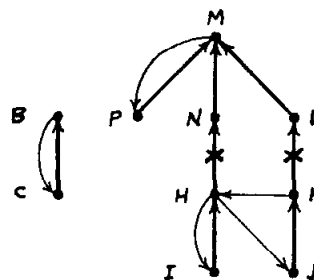
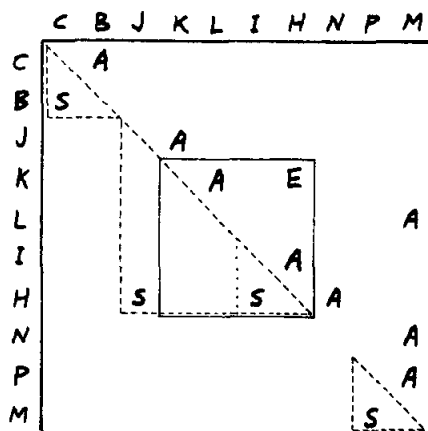


FIG. 18-29

One of the many possible chart arrangements and the graph of the source code. The numbers (1) and (2) in the source code specify that row M must be below row B. Source code:

```
A Eps B := (2) ELEMENT (C Eps B = D) Eps E Eps F
  <-- G Eps H := ELEMENT
    {I Eps B = J Eps K := ELEMENT {EXISTS H) Eps L Eps M)
    Eps N Eps M := (1) ELEMENT (P Eps M = Q) Eps R Eps S;
```

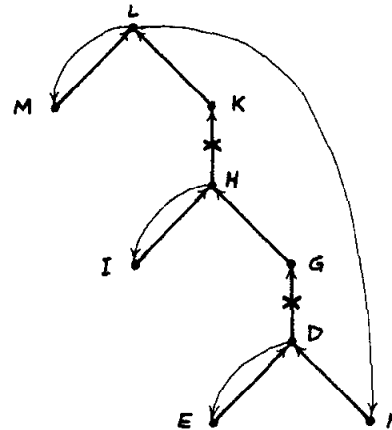
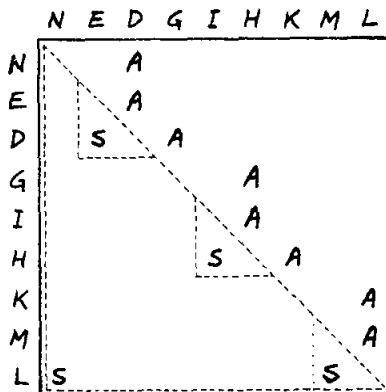


FIG. 18-30

Source code:

```
A Eps B <-- C Eps D := ELEMENT (E Eps D = F) Eps G Eps H := ELEMENT
(I Eps H = J) Eps K Eps L := ELEMENT (M Eps I = N Eps D) Eps P Eps Q;
```

Translated equivalent:

```
RESERVE D;
RESERVE H;
RESERVE L;
LOOP FOR ALL L := ELEMENT Eps P Eps Q
DC LOOP FOR ALL H := ELEMENT Eps K Eps L
DO IF I Eps H = J
THEN GO TO DUMMY1
END LOOP;
ERROR;
DUMMY1:
LOOP FOR ALL D := ELEMENT Eps G Eps H
DO IF E Eps D = F
THEN GO TO DUMMY2
END LOOP;
ERROR;
DUMMY2:
IF M Eps L = N Eps D
THEN GO TO DUMMY3
END LOOP;
ERROR;
DUMMY3:
A Eps B <-- C Eps D;
```

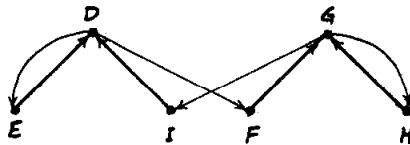
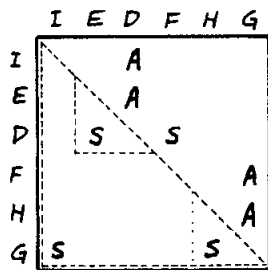


FIG. 18-31

Fig. 18-18(a) and (c) redrawn. Source code:

```
A Eps B <-- C Eps D := ELEMENT
  (E Eps D = F Eps G := ELEMENT (H Eps G = I Eps D) Eps J Eps K)
  Eps L Eps M;
```

Translated equivalent:

```
RESERVE D;
RESERVE G;
LOOP FOR ALL G := ELEMENT Eps J Eps K
DC LOOP FOR ALL D := ELEMENT Eps L Eps M
  DO IF (E Eps D = F Eps G) & (H Eps G = I Eps D)
    THEN GO TO DUMMY1
  END LOOP
END LOOP;
ERRGR;
DUMMY1:
A Eps B <-- C Eps D;
```

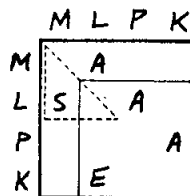


FIG. 18-32

Fig. 18-21 redrawn. Source code:

```
H Eps I <-- J Eps K := ELEMENT
  (EXISTS L := ELEMENT (M Eps L = N) Eps P Eps K)
  Eps Q Eps R;
```

Translated equivalent:

```
RESERVE K;
RESERVE L;
LOOP FOR ALL K := ELEMENT Eps Q Eps R
DC LOOP FOR ALL L := ELEMENT Eps P Eps K
  DO IF M Eps L = N
    THEN GO TO DUMMY1
  END LOOP
END LOOP;
ERROR;
DUMMY1:
H Eps I <-- J Eps K;
```



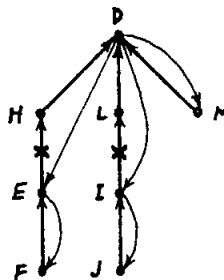
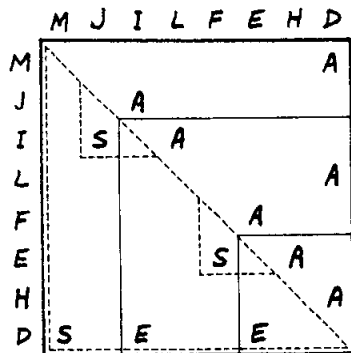


FIG. 18-33

Source code:

```
A Eps B <-- C Eps D := ELEMENT
  (EXISTS E := ELEMENT (F Eps E = G) Eps H Eps D |
   (EXISTS I := ELEMENT (J Eps I = K) Eps L Eps D & (M Eps D = N)))
  Eps P Eps Q;
```

Translated equivalent:

```
RESERVE D;
RESERVE E;
RESERVE I;
LOOP FOR ALL D := ELEMENT Eps P Eps Q
DO LOOP FOR ALL E := ELEMENT Eps H Eps D
  DO IF F Eps E = G
    THEN GO TO DUMMY1
  END LOOP;
  LOOP FOR ALL I := ELEMENT Eps L Eps D
  DO IF J Eps I = K
    THEN GO TO DUMMY2
  END LOOP;
  GO TO DUMMY3;
DUMMY2:
  IF M Eps D = N
  THEN GO TO DUMMY1;
DUMMY3:
END LOOP;
ERROR;
DUMMY1:
A Eps E <-- C Eps D;
```

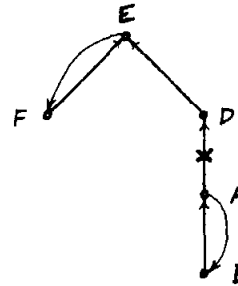
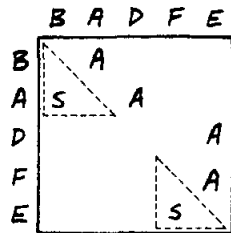


FIG. 18-34

Source code:

```
IF EXISTS A := ELEMENT (B Eps A = C) Eps D
  Eps E := ELEMENT (F Eps E = G) Eps H Eps I
THEN J Eps A <-- K Eps L
ELSE M Eps N <-- P Eps Q;
```

Translated equivalent:

```
RESERVE A;
RESERVE E;
LOOP FOR ALL E := ELEMENT Eps H Eps I
DO IF F Eps E = G
  THEN GO TO DUMMY1
END LOOP;
ERROR;
DUMMY1:
LOOP FOR ALL A := ELEMENT Eps D Eps E
DO IF B Eps A = C
  THEN GO TO DUMMY2
END LOOP;
M Eps N <-- P Eps Q;
GO TO DUMMY3;
DUMMY2:
J Eps A <-- K Eps L;
DUMMY3:
```

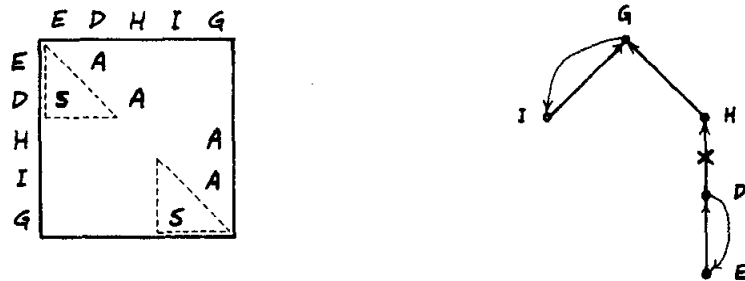


FIG. 18-35

A simple example of selecting all elements. Source code:

```
A Eps PREFACE ELEMENT Eps B
  <-- C Eps ALL D := ELEMENT (E Eps D = F) Eps H
    Eps ALL G := ELEMENT (I Eps G = J) Eps K Eps L;
```

Translated equivalent:

```
LOOP FOR ALL G := ELEMENT Eps K Eps L
DO IF I Eps G = J
  THEN LOOP FOR ALL D := ELEMENT Eps H Eps G
    DO IF E Eps D = F
      THEN A Eps PREFACE ELEMENT Eps B <-- C Eps D
    END LOOP
  END LOOP;
END LOOP;
```

FIG. 18-36 (On following pages.)

Charts and graph show use of ALL.

- (a) Graph shows source code.
- (b) Original chart.
- (c) Rearrangement of original chart with A's in upper-right triangle.
- (d) Irreducible chart. Several other arrangements are possible. Some paths of dependency propagation shown.
- (e) Irreducible chart with expanded scopes.

Source code:

```
A Eps PREFACE ELEMENT Eps B
<-- 2
+ C Eps D := ELEMENT (E Eps D = F) Eps G
    Eps H := ELEMENT (I Eps H = J) Eps K
    Eps L := ELEMENT (M Eps L = ANY OF N Eps ALL P := ELEMENT
        (Q Eps F = R) Eps S Eps T) Eps U
    Eps ALL T := ELEMENT (V Eps T = W) Eps X
    Eps Y := ELEMENT (Z Eps Y = AA Eps H) Eps AB Eps AC
* AD Eps AE := ELEMENT (AF Eps AE = AG) Eps AH
    Eps ALL AI := ELEMENT (AJ Eps AI = AK) Eps AM Eps AN;
```





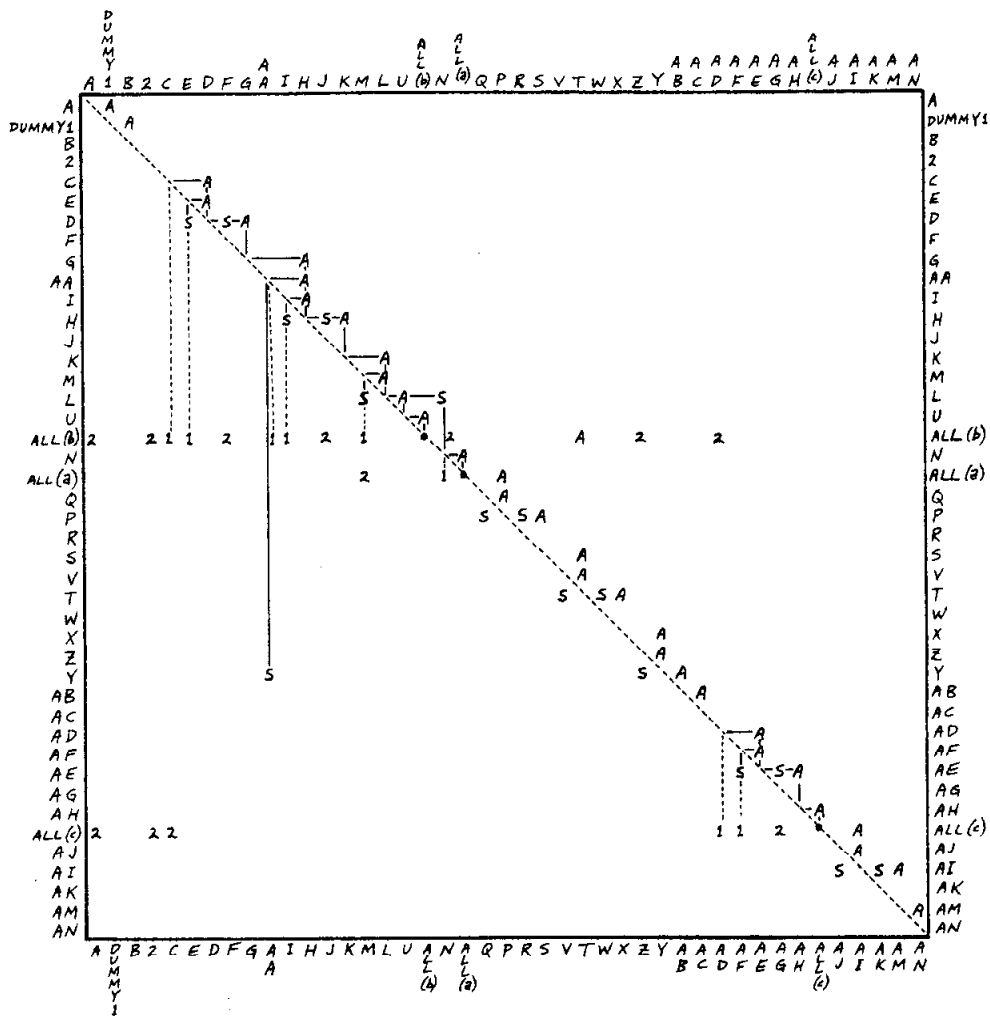


FIG. 18-36(c)

Rearrangement of original chart with A's in upper-right triangle.





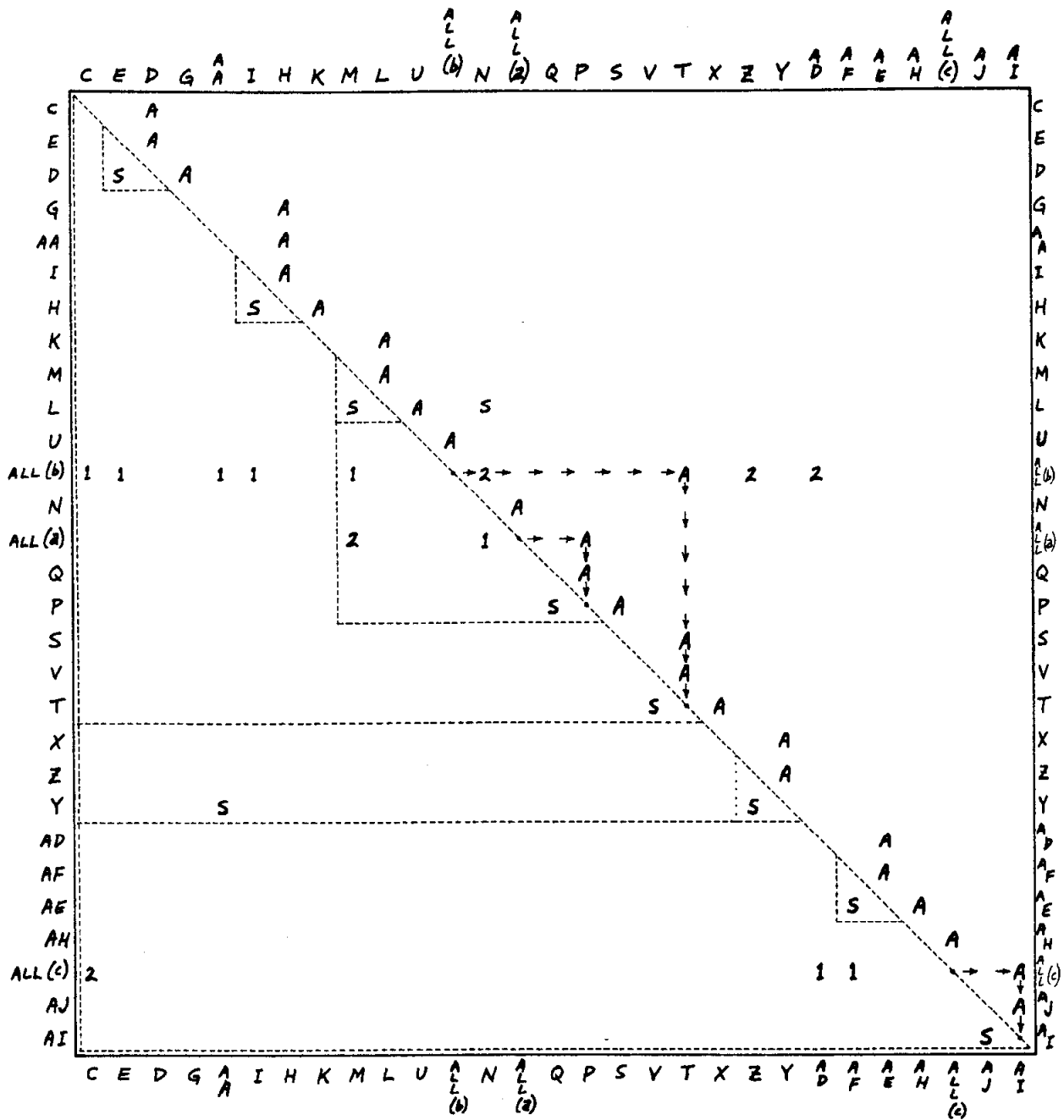


FIG. 18-36(e)

Irreducible chart with expanded scopes.

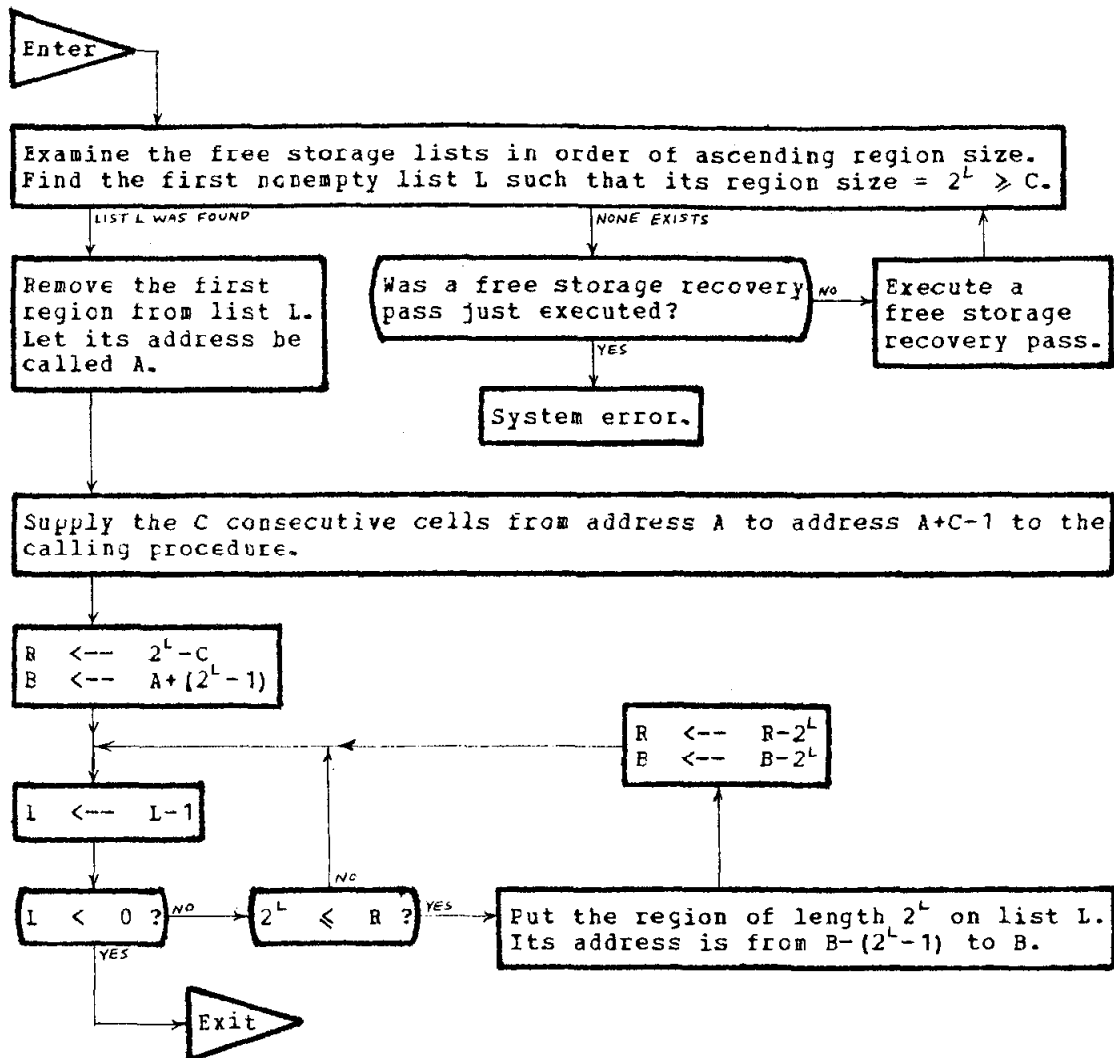
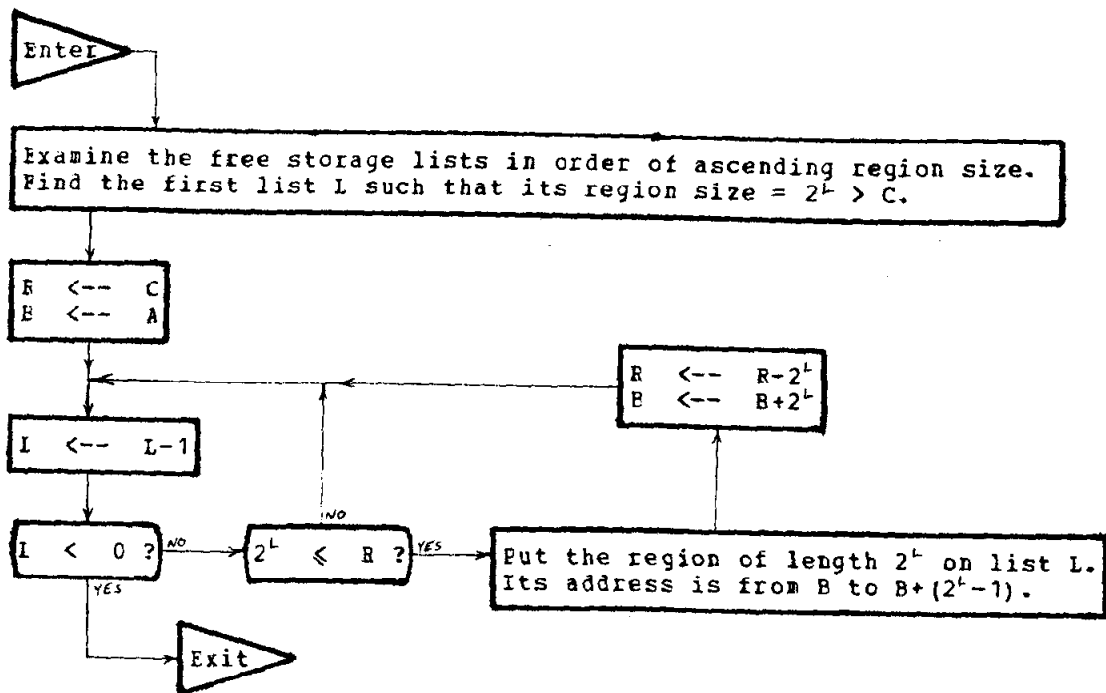


FIG. 29-1

Get a region of C consecutive cells.



**FIG. 29-2**

Free a region of C consecutive cells, from address A to address A+C-1.