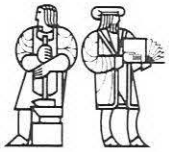
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-51

AN INVESTIGATION OF CURRENT LANGUAGE SUPPORT FOR THE DATA REQUIREMENTS OF STRUCTURED PROGRAMMING

Jack M. Aiello

September 1974

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

AN INVESTIGATION OF CURRENT LANGUAGE SUPPORT FOR THE DATA REQUIREMENTS OF STRUCTURED PROGRAMMING

by

Jack Michael Aiello

Submitted to the Department of Electrical Engineering on May 10, 1974 in partial fulfillment of the requirements for the Degrees of Master of Science and Electrical Engineering.

ABSTRACT

Structured programming is a new method for constructing reliable programs. Structured programming relies upon a systematic technique of top-down development which involves the refinement of both control structures and data structures. With possibly some limitations and extensions, existing languages can support control structure refinement. On the other hand, it is the belief of many that the representation of data structure refinement cannot be satified by present-day languages. Before accepting this view, it is wise to explore its validity. Therefore this thesis will investigate whether existing languages with possibly slight modifications are adequate for supporting the data requirements of structured programming.

THESIS SUPERVISOR: Barbara H. Liskov
TITLE: Assistant Professor of Electrical Engineering

MAC TECHNICAL MEMORANDUM 51

CSG MEMO 105

AN INVESTIGATION OF CURRENT LANGUAGE SUPPORT FOR THE DATA REQUIREMENTS OF STRUCTURED PROGRAMMING

Jack M. Aiello

September 1974

This research was supported by the National Science Foundation under research grant GJ-34671.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

My greatful thanks go to my thesis supervisor, Professor Barbara Liskov, for her guidance, constructive criticism, and patience throughout the preparation of this thesis.

I would also like to thank my colleagues Mark Laventhal and Dave Ellis for reading earlier drafts and suggesting many improvements.

A special note of thanks goes to Professor Thomas Cheatham at Harvard University for his personable manner and willingness to answer any of my questions.

Most importantly, I wish to thank my wife, Nell, for giving me her love and understanding plus doing my share of the dishes for the past two months, and my parents, for providing me with their confidence.

INTRODUCTION

Considering the major strides that have been made in twenty-five years in computer science and more specifically in software design, the techniques available for developing large reliable systems or even producing correct medium-sized programs are disappointing. We generally attack a problem by choosing some programming language and then proceed to design the program using an iterative process of coding and debugging. Unfortunately after completing these steps of program preparation, we still have no guarantee that our product is error-free. Instead a general method for developing programs which are assured to be correct is needed. This field of research comes under the heading of reliable programming.

Reliable programming can be defined into two general categories. The first is the "analytical" approach encouraged by Floyd(1), King(2), London(3), and others who attempt to prove a program correct after it has been written. This is accomplished by deriving assertions about values of variables and control patterns within the program. From these assertions, one then uses mathematical methods in trying to deduce the desired end result of the program.

The second approach, called "constructive" and primarily introduced by Dijkstra(4), is to develop a method for

constructing correct programs rather than proving the reliability of previously written programs. The goal of the constructive approach is writing programs that are easier to prove correct. This is accomplished by adhering to established programming restrictions while writing a program which, in turn, causes the program to be constructed as a hierarchy of layers. Guaranteeing the reliability of the program is then reduced to proving the correctness of each layer. The hope is that larger programs can be handled by the analytic approach.

The general constructive method that has been developed is structured programming (other names used are programming by levels of abstraction and programming by stepwise refinement). Although Dijkstra pioneered most of the early research in this area, several other papers have now followed including those written by Wirth(5), Parnas(6), Henderson and Snowdon(7), Woodger(8), Naur(9), and Liskov(10). Their works have all pointed out that what is really needed is a more systematic way to go about programming--a systematic programming method that leads to clear, readable, provable, and thus reliable programs.

Structured Programming

Structured programming is a design tool for building programs in a "top down" fashion by means of a process of

successive decomposition. When we say "top down" we mean the ability to introduce and use objects while designing a program before the objects are necessarily defined. On the other hand, a "bottom up" approach--required in most present-day programming languages--means that an object must be defined in the language before it can be used. The first step in structured programming is to write a program which solves the given problem. However, in general this initial program is not in a form understandable by the machine on which we are programming but instead only on some abstract machine. This abstract machine provides us with just those data objects and operations which have been generated by our program and are therefore suitable for solving the problem. This program is referred to as our top level abstraction. As was earlier noted, the fact is that names of operations and data structures may not be recognizable by the actual machine; so we refine these names by using real machine constructs as well as lower level abstractions which themselves, in turn, must be subsequently developed until our program is completely understandable by the machine on which the program is to be run.

Problem Overview

Most higher level languages today give the programmer the ability to design abstract operations through the usage

of procedural mechanisms. However, there seems to be no analogous method for handling abstract data objects. Obviously, if we hope to design programs by means of a structured approach, this second requirement is as necessary as the first.

Currently at the Massachusetts Institute of Technology research is being directed towards the development of a new programming language to be used specifically in conjunction with the structured programming methodology(11). While it is being assumed that current langugaes do not have suitable mechanisms for constructing data objects, it is not obvious that this opinion is justified. In an attempt to remedy this predicament, we will begin by analyzing the concept of data abstractions and how one would program them in some present day languages. Although our choice of languages becomes relevant to our successes and failures, it is important to recognize that we have chosen languages which have been all designed with different concepts in mind. The languages chosen are 1)PL/I--described by IBM as an "all-purpose language", 2)Pascal--a language with numerous data structuring facilities, 3)ELl--an extensible language, and 4)SIMULA67--a simulation language.

We do <u>not</u> intend to evaluate these languages in terms of the goals of their designers; instead these languages will be

judged by how well they meet our design criteria set forth by structured programming . We will pay particular emphasis on their abilities to represent data abstractions. As a result of the insights gained by this study, we will expect to answer the question of whether or not another programming language must be developed for structured programming. Furthermore, if there is a need for a new language of this type, what we have learned from this analysis should prove useful.

Before we explore these languages, we present a more detailed history of the problems of structured programming so that the reader will feel more competent when judging and trying to resolve the pros and cons of each language. Criteria upon which these assesments will be made will also be presented shortly.

HISTORY

Program reliability from the analytical point of view has encouraged much research from the latter part of the 1960's up till the present. These years of effort have produced much material on the subject ranging from improved resolution schemes for logic(12) to assertion-type languages(13). However, one fact becomes fairly obvious: the more unstructured--in the sense of wild control flow and enormous (but unecessary) size--a program is, the more difficult the problem of deriving assertions in order to prove the program correct. Furthermore, the increase in difficulty seems to be more on the order of exponential (and approaching impossible) than linear. Indeed, it seems that if only some straightforward design methodology for programming were adhered to during the development of the program, that finding these assertions would be much easier.

This has been one of the goals of the constructive approach and in particular structured programming. As we noted earlier (but explain now in fuller detail), by building a program in this fashion, we produce an arrangement of layers, corresponding to levels of abstraction, each with only one entry and one exit. The reliability of the program is based upon the reliability of each succeeding layer, and a change made to some higher level of abstraction (that is, an earlier conceived level) has no effect on the reliability of

lower levels. Thus proving the correctness of the entire program is reduced to proving the correctness of each layer in an orderly fashion. The correctness of each layer is not too difficult to prove as a result of making the following restriction: control sequencing is limited to the concatenation of assignment, IF-THEN-ELSE, DO WHILE, and possibly CASE statements; the correctness of each layer relies upon the fact that these statements correspond to proofs using mathematical methods of enumeration, case analysis, and induction.

An Example

At this point an example is presented to help clarify the method followed in structured programming and its associated reliability characteristics. Written in a PL/I-like language, the example will also display English phrases and undefined symbols at one level that must be refined at lower levels. Suppose the problem is to write a compiler for PL/I. Then the top level would be:

```
1  Write a compiler for PL/I programs;
```

Now if we had a machine capable of understanding level 1, then we would have no need to proceed further; however, we will assume this not to be the case.

Instead we will refine the above program thus obtaining:

```
l1 program:
     DECLARE source_program CHARACTER VARYING;
     source_program = INPUT;
     translate(source_program);
```

Level l1 consists of naming the variable source_program, which is to be type varying length character sequence, and two actions, INPUT and "translate". The assumption we will make here is that the machine understands DECLARE, CHARACTER, VARYING, and INPUT but not "translate". Assuming that INPUT sets the source_program to the program to be compiled and "translate" does indeed produce the correct object code, it is easy to see that level l1 solves the problem. The reliability of level l1 must be partially intuitive since the specifications of the action "translate" are not defined. This particular point concerning reliability should be well understood and is therefore reiterated here: at level l1 it is unnecessary to know how "translate" works but only what it is supposed to do; furthermore since we have no semantics to express what the function of "translate" is, our proof relies partly on enumeration (concatenation of statements) and partly on intuition.

The next refinement is a result of our actual machine not understanding the action translate.

```
III  translate(x);
       DECLARE x CHARACTER VARYING;
       DECLARE phrase_structure  tree;
       phrase_structure = recognize(x);
       generate_code(phrase_structure);
```

Level III introduces the parameter x, variable phrase_structure of type tree, and operations recognize and generate. Of these, only x is completely understood by the machine on which we will eventually run and the problem of what to refine next must be resolved. (The question of what to refine next has no simple solution--see discussion in following section.)

At this point I will stop because I feel the reader should have a reasonable understanding concerning the development of the compiler program. He should also be able to convince himself that if all the data types and operations introduced so far were recognizable by our machine, not only would the program be complete but it would also be correct. As was the case for level II, the way by which one can show the reliability for level III results from programming in a structured manner. For instance, to prove the reliability of level III, one needs to show nothing of the details concerning the operations recognize and generate and the data structure tree but only what the two actions are supposed to do and that the representation of the data structure tree will be refined at some lower level.

General Questions Concerning Structured Programming

The compiler example illustrates the general form a structured program takes. Two types of problems which are encountered in building programs structurally may be identified. First it is evident that decisions concerning the next refinement must be resolved but it is unclear as to how that particular choice is made. Making such decisions is not limited to structured programming, but is an integral part of constructing programs (although it may be that structured programming makes decisions more noticeable by identifying them with refinements). The second problem is how to represent the structured program as it develops. This involves representation of both structured control and structured data.

With regard to the second of these two issues, we remark that structured control can be broken down in two areas: control sequencing and action refinement. The requirements of control sequencing do not seem to present an implementation problem. They can easily be handled by a number of present day programming languages, for example PL/I (see Mills(14) and Sullivan(15)), with most likely some limitations and possibly some extensions, e.g. implementing the CASE statement in PL/I.

As was earlier recognized, refinement of action such as "translate" in the compiler example can be expressed by the combination of procedure-like structures and a good library system which supports linking of action names with action definitions. This would allow the programmer the advantage of being able to build the program in a top-down fashion. Indeed, the development of a language which provides the structured logic and library system to back up the compiler would provide a solid foundation for the development of a language specifically designed for structured programming.

On the other hand, the method in which data specifications in structured programming should be handled is unclear. For instance, in the compiler example how do we represent the abstraction "tree" and any associated operators (eg. we may wish to include the actions traverse, add_a_node, and delete_a_node operating on trees)? This issue is closely bound to the first problem of when and how one should define and refine data structures. Certainly the representation of the data structure will have a profound effect on the efficiency of the program and the ease of writing the functions which will operate on the data. But how is the programmer to decide upon the details of the structure while still conforming to a top-down development of the program?

The more generally held belief is to defer those

decisions concerning the details of data representation as long as possible which hopefully leads to refinement of program and data specifications in parallel. The following example will help the reader become more familiar with the questions of data format that the programmer must face in structured programming.

Wirth(5) develops a program to solve the 8-queens problem by a "successive decomposition or refinement of specification...(until) all instructions are expressed in terms of an underlying computer or programming language." The 8-queens problem is described as follows: given an 8x8 chessboard and 8 queens, find a position for each queen so that no queen can take another (i.e. such that every row, column, and diagonal contains at most one queen). The first data decision that Wirth must make is when to define the board. He decides upon a time when it actually becomes necessary to access and manipulate parts of the board.

Then the decision of how to represent the board becomes of primary importance. An obvious solution consists of introducing a boolean matrix B(1:8,1:8) such that B(i,j) = true denotes that square (i,j) is occupied. But instead Wirth chooses the representation:

```
integer j          (0<=j<=9)
integer array x(1:8)  (0<=xi<=8)
```

such that j is the index of the currently inspected column (allowing j to equal 9 will indicate that the solution is found), (xj,j) is the coordinate of the last inspected square, and the position of the queen in column k is given by the coordinate pair (xk,k) on the board. In justifying this decision, Wirth says "it is fairly evident even at this stage that the...(second) choice is more suitable than a boolean matrix in terms of simplicity of later instructions as well as of storage economy."

So it seems that although structured programming is a top-down process, a bottom-up justification has been given to support a choice of data representations. Unfortunately, if it is discovered that a wrong choice has been made, one will be forced to back up. An analysis of the trade-off between a strict top-down development and necessary backup might improve any decision policy on what to refine next and what format the refinement should take.

In principle there seem to be two ways of answering the first question of when to refine data. We can do it as soon as we realize that a particular data structure is needed some time in the program or we can postpone the definition until that data structure must be defined in order for us to be able to continue along our path of refinement.

The second way does seem to be more in the spirit of structured programming since it does not require looking ahead to see how a data structure is operated upon before defining it, i.e. Wirth uses the concept of "hoard" as long as possible. Also by postponing the decision we eliminate possible problems generated by awkward data format that has already been defined. On the other hand, waiting to choose a particular data representation can result in lack of compatibility and thus inefficiency if any presupposed operations on that format turn out to be clumsy. It also must be realized that we cannot postpone decisions forever and that the problem of backup will arise and have to be dealt with.

So now the problem becomes one of trying to formulate some rules or tactics concerning the refinement of data. Ideally, they would ensure the programmer of being able to tell not only when but exactly how to refine data. However, even before we can approach this problem, we must first be able to handle data representation and refinement in the language.

It is this latter problem to which this paper addresses itself with the belief that its solution will present us with both a firm understanding of data representation and a better comprehension of the concepts of structured programming. In

turn, we may then have hopes of being successful in solving the first problem of when and how to refine data. Before investigating different languages, let us explore precisely what criteria need be satisfied by a language acceptable for expressing abstract data and what examples we may wish to use in evaluating the different languages.

PROGRAMMING LANGUAGE CRITERIA FOR DATA ABSTRACTIONS

We noted earlier that as a result of using structured programming techniques, both abstract data objects and abstract operators are generated. These objects will henceforth be referred to as data abstractions and operator abstractions respectively. We have also established the fact that operator abstractions can be fairly well represented by procedure-like mechanisms common to most present-day programming languages. So it is at this point that we are most interested in establishing criteria for representation of a data abstraction the must be met by a programming language.

Some of the lesser data structuring requirements will become clear as we analyze the languages with the aim of producing programs through the use of structured programming. However, the major criteria should be made very precise at this point in the paper so that we have both set goals around which programs should be designed in our given language and a model for comparison of programs.

One obvious and possibly the most important requirement is that of being able to express data abstractions within the confines of the language. More explicitly, our view is that the data abstraction introduces some abstract data type, and variables of this type may only assume values corresponding

to this type (just as in ALGOL60, say, variables of type real may only assume values that are real numbers). The abstract data type is explicitly defined by some set of operations which may operate on variables declared of that type. The implementation of the abstraction may then be viewed as a form consisting of two parts: 1)the underlying representation of the abstract data type and 2)the actions defined to operate on variables of that data type by manipulating its underlying representation.

For example, a common abstraction in mathematics is the concept of a set which we might define by the operators union, membership-testing, and intersection. To implement the set abstraction, we choose some lower level representation for a set. Our choice of representation depends on the kind of operations which will predominate in the application expected. We could represent a set as an array or possibly as elements linked together by pointers. After choosing one of these underlying representations, we would program the operators given above. This would complete our implementation of the set abstraction although it may be the case that further refinement with respect to the representation chosen and the operators defined is necessary so that our program is completely understandable by the actual machine.

It is also a fact that at a higher level we need not nor should we have access to the lower level (underlying) representation of some data abstraction. This restriction has been implemented by most higher level programming languages for primitive types and there is no reason that we should ignore it while programming by levels of abstraction. For instance, a real number in a programming language is usually represented as a sequence of bits which gives the mantissa and exponent parts of the number. However, there is certainly no reason for the programmer to know which bits represent which part or even to allow the programmer access to any bit. Indeed, this lower level representation of real numbers is no concern to the programmer using the higher level tool of a programming language. This argument applies in a similar fashion to our example of the set abstraction. We are interested in declaring variables to represent sets and in accessing information about these variables through the operators associated with the set abstraction. But the data structure that was chosen to represent a set should be of no concern to us.

In summary, a data abstraction defines an abstract data type as a set of operators. Furthermore, the user is not permitted to know how this type is represented in the abstraction; instead, only the specified operators can manipulate variables of this type. It is these criteria

around which the language for structured programming, mentioned earlier, is being developed at M.I.T.

When judging the chosen programming languages, we must do so in terms of these requirements. Can we represent abstract data concepts and at the same time adher to the accessibility restrictions? If the answer is yes, then with what ease can this be accomplished? Does any syntactical format or semantic concept of the given language encourage the use of structured programming as the designer's tool or, on the other hand, cause any serious handicap while using the technique of structured programming?

Before leaving the topic of criteria for data abstractions, we should focus on one more issue upon which discussion ought to be based during the examination of a language. Suppose that we have conceptually formulated the model for a given abstract data type; that is, we have decided upon the representation of the type and the operations which can manipulate objects of that type. Our next step is to program this data abstraction in our chosen programming language. We should be concerned about the task of conceptualization versus the task of programming the data abstraction. Was the abstraction much more difficult to program than its conceptualization leads us to believe? If the language does not meet this criterion of

"conceptualization ease implying programming simplicity" then the language must be regarded as a disappointment from this viewpoint.

Stack Abstraction

Let us now examine a data abstraction example we have chosen to program in our languages. We will name it the "stack abstraction".

The stack is a linear list in which insertion, deletions, and accesses to values are made at one end of the list. The usages of stacks are many including the implementation of a polish-notation interpreter, the supporting of recursion, and the designing of parsing algorithms (eg. for operator precedence grammars we might wish to use two stacks called the "operator" and "operand" stacks).

When describing the concept of an abstraction earlier, we noted that it can be separated into two parts: 1)a representation of the data object (in this case the stack), and 2)operations on the object. The underlying representation we have chosen for a stack will be an array to hold the stack elements and a pointer which points to the top filled location of the stack. Operations on a stack that we will consider are the following: 1)"push"--adding an element

to the top of the stack, 2)"pop"--deleting the top element from the stack, 3)"top_element"--accessing the top element value from the stack, and 4)"initialization"--initializing the bottom element of the stack. Thus the stack abstraction is conceptually defined.

Linear List Abstraction

A second example we will look at will be referred as a linear_list abstraction. We wish to model three distinctive types of linear lists: a stack, a queue, and a dequeue (see Knuth(16)). At the top level of abstraction all linear lists can be represented as arrays of locations. At a lower level this representation may be further specified depending on what type of linear lists we are using. A stack abstraction would add as part of its underlying representation a pointer to the top of the stack and add operations that we have outlined earlier. A queue abstraction might add "front" and "rear" pointers to its representation and implement operation of "enter"--which would insert an element in the rear of the queue, and "remove"--which would delete an element from the front of the queue. A dequeue abstraction could add "leftmost" and "rightmost" pointers to its representation and operators--call them "insert" and "delete"--which make additions and subtractions to both ends of the list. Thus we have introduced the notion of hierarchical data types and it

is of interest to us to examine how our languages can represent this concept.

We will analyze each chosen language with two goals in mind: 1)we wish to see what can be learned from programming in each language while attempting to meet the specifications of structured programming, and 2)we want to determine if the language can be used as a structured programming language.

PL/I

The block structured language, PL/I(17), was developed by IBM during the mid 1960's well before the concepts of structured programming as a design mechanism for constructing programs were explicitly introduced by Dijkstra. PL/I was designed to cover as wide a range of programming applications as possible. The designers consider one of its prime features to be the ease with which modular programs are built, encouraged by the fact that a PL/I program is composed of blocks of statements called procedure and begin blocks.

It should also be mentioned that PL/I is an extremely large language and it is inconceivable to this writer how anyone could be knowledgeable of the complete language. The designers of the language recognized this as a possible drawback that would discourage programmers from learning PL/I (and companies from adopting the language) and have made programmers aware of the fact that one needs to learn only a small subset of PL/I in order to write most medium-difficult programs. (After all, this covers the bulk of written programs.) However, the language does contain such options as multi-tasking so that one is able to solve more-difficult problems using PL/I. As I present some of the language constructs and the examples, I may also be unaware of some "simpler" technique (although it is my belief that due to the enormous size of PL/I this is not my fault). However, I am

also quite sure that any "missed" feature could not change the overall outcome of the language with regards to data abstractions of structured programming.

One might argue that PL/I was not developed with structured programming in mind and therefore it is unfair to evaluate PL/I as a structured programming language. On the other hand, structured programming is intended to be a design method based upon refining levels of abstraction until they are expressed in terms of whichever language--may it be PL/I or any other--that the programmer is using. Thus it seems to be perfectly justified to examine PL/I in these terms.

Data Structuring Facilities of PL/I

We begin by describing some of the data structuring facilities of PL/I; however, it will be assumed that the reader is somewhat familiar with the language or at least some block structured language (eg. ALGOL60). Therefore we will describe only those data facilities that we might come into contact with when using the process of structured programming to write programs in PL/I.

We begin with variables and data attributes. Variables may be considered to be single elements, arrays, or structures. Associated with each variable is a symbolic name and a value that may change during the execution of the

program. The attributes of a variable consist of its basic type and storage class. Basic types in PL/I are FIXED(*), FLOAT, CHARACTER, BIT, and POINTER; storage classes of a variable are STATIC, AUTOMATIC, BASED, and CONTROLLED.

While basic types should be clear, an explanation of storage classes is warranted. The storage class attributes are used to specify the type of storage allocation to be used for a data variable. The default class is AUTOMATIC, which means that storage is allocated upon entering the block and is released (freed) upon exit from the block. The STATIC class, on the other hand, specifies that storage is to be allocated at load time and not released until program execution has been completed. BASED and CONTROLLED give the programmer two different ways of explicitly controlling the allocation and freeing of storage by using the ALLOCATE and FREE statements.

A number of other possibly useful data structuring facilities with respect to structured programming exist as part of the PL/I language. These include block structuring techniques (signified by BEGIN...END and PROCEDURE...END blocks) which partially allow us to think in terms of abstractions (associating an abstraction with a block)

(*)
Upper case letters will be used to signify keywords in PL/I.

especially with regard to operators. A second construct is the data attribute LIKE. Its function is to copy the structuring, names, and attributes of structures. For instance, suppose we wanted to declare a variable, s, to be a set (see Chapter 3). Assuming "set" were declared as a structure variable, we might write "DECLARE s LIKE set;". The result would be that s has the same structure as the variable set. A third facility of PL/I which we will investigate is ENTRY points into a procedure. This feature not only allows one to enter a procedure at some designated point (other than at the beginning) but also permits us to specify parameters and return attributes. Thus we may be able to set up numerous "operator" ENTRY points within a given procedure, which as a whole might represent a data abstraction.

Stack Abstraction in PL/I

Three stack abstraction examples in PL/I will be presented. This number of examples is due to language restrictions in conjuntion with program requirements. A detailed analysis of each example will follow the presentation of all three examples.

In PL/I the stack abstraction outlined in the previous chapter might be programmed as shown in fig. 4-1. For now, it is important that we understand how the procedure

create_stack works. Note that upon entering create_stack, not only is space allocated for the stack but also the bottom element is initialized. We should also be aware that this first example is capable of handling only one stack at a time in existence. The operators seem fairly straightforward although one should note that pop deletes the top element merely by decrementing top and does not return that element, while top_element performs the function of accessing the top element but not deleting it from the stack.

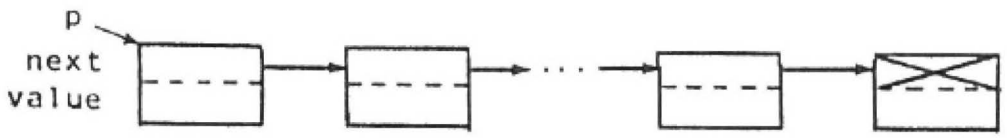
```
create_stack: PROCEDURE(n);
              DECLARE n FIXED BIN(15);     /* stack size */
              DECLARE stack(n) CHAR(1) CONTROLLED;
              DECLARE top FIXED BIN(15) STATIC;
                              /* topmost filled location */

              ALLOCATE stack;
              stack(1) = '!'; /* initial both the bottom*/
              top = 1;        /* element and the pointer*/
              RETURN;         /* to the stack           */

       push:  ENTRY(a);       /* insert the value of a  */
              DECLARE a CHAR(1);
              top = top+1;
              stack(top) = a;
              RETURN;

        pop:  ENTRY;          /* delete the top element */
              top = top-1;
              RETURN;

top_element:  ENTRY RETURNS(CHAR(1)); /* get top elem   */
              RETURN(stack(top));


              END create_stack;
```

fig. 4-1

Before proceeding further, we should also note that representing a stack as a controlled array is not the only representation one might think of. Indeed, the argument to represent a stack as a list of chained elements is solid and, in fact, this stack description will be programmed in a later example (see fig. 4-2). Our declaration for a stack might have been written as follows:

```
DECLARE 1 stack BASED(p),  /* p is as pointer */
          2 next PTR,
          2 value CHAR(1);
```

Pictorially our stack would have looked like:



So instead of allocating the whole stack at once, upon each call of the operation push (pop), a single element is allocated (freed).

The big limitation of fig. 4-1 is in trying to represent several different stacks at the same time. This problem of multiple stack usage can be solved by rewriting the stack example using the LIKE attribute. Before doing so, a further characteristic of LIKE should be revealed: to write "DECLARE a LIKE b;" requires that b be declared on the same or higher

block level (a global declaration) than a. Realizing this restriction, the example is programmed as shown in fig. 4-2.

```
DECLARE x FIXED BIN(15);

DECLARE 1 stack BASED(p),
          2 m FIXED BIN(15),  /* global decl. for stack */
          2 body CHAR(m REFER(x)),
          2 top FIXED BIN(15);


stack_ops: PROCEDURE;
           /* Contained within are the following oper-  */
           /* ations :  push, pop, and top_element plus */
           /* the oper of initialize.  The parameters   */
           /* stack_ref and elem refer to the stack in  */
           /* concern and hold the value of the element */
           /* to which we are referring respectively    */
           RETURN;  /* no reason to call stack_ops       */

     push: ENTRY(stack_ref, elem);
           DECLARE stack_ref PTR, elem CHAR(1);
           DECLARE 1 name BASED(stack_ref) LIKE stack;
           name.top = name.top + 1;
           name.body(name.top) = elem;
           RETURN;

      pop: ENTRY(stack_ref);
           DECLARE stack_ref PTR;
           DECLARE 1 name BASED(stack_ref) LIKE stack;
           name.top = name.top - 1;
           RETURN;

top_element:
           ENTRY(stack_ref)RETURNS(CHAR(1));
           DECLARE stack_ref PTR;
           DECLARE 1 name BASED(stack_ref) LIKE stack;
           RETURN(name.body(name.top));

initialize:
           ENTRY(stack_ref,elem);
           DECLARE stack_ref PTR, elem CHAR(1);
           DECLARE 1 name BASED(stack_ref) LIKE stack;
           name.body(1) = elem;
           name.top = 1;
           RETURN;


           END stack_ops;
```

fig. 4-2

In conjuction with fig. 4-2, at an outside block level (corresponding to a higher level of abstraction--although no higher than the one on which the stack is declared), we can now write the following lines of code:

```
DECLARE 1 operator BASED(p) LIKE stack;
                              /* operator stack */
DECLARE 1 operand BASED(p) LIKE stack;
                              /* operand  stack */
        .
        .
        .
x = 100;
ALLOCATE operator; /* the sizes of the bodies */
x = 10;             /* are now established for */

ALLOCATE operand; /* the operator and operand */
        .          /* stacks respectively      */
        .
        .
CALL initialize(p,'!');/*initialize each stack*/
CALL initialize(q,'$');
        .
        .
        .
IF...THEN CALL push(p,'a');
                /* push 'a' onto the opertor  */
     ELSE CALL pop(q);
                /* or pop the operand stack   */
        .
        .
        .
  y = top_element(q);   /* y gets set to the  */
              /* top element of the operand   */
        .
        .
        .
```

These statements in conjunction with the comments should be easily understood as to their meanings and results.

Referring back to fig. 4-1, we recognize the fact that

this first program would work only for a single stack usage. On the other hand, this last example allowed usage for a multiple number of stacks; however because we have used LIKE, we have had to separate the representation of a stack from its operations. Suppose we did not want to break up the abstraction and yet would like to take care of the case when we had the need to use more than one stack. The resulting program might resemble that in fig. 4-3.

```
stack:      PROCEDURE;
            DECLARE 1 stack_element BASED(sep), /* struc. */
                      2 last_stack_element PTR,
                      2 value CHAR(1);
            DECLARE sep PTR;
            DECLARE wp PTR;
            DECLARE get_stack_descriptor ENTRY(CHAR(32),PTR)
                    RETURNS(PTR);
            RETURN;

create_stack:
            ENTRY(stack_name1, init);
            DECLARE stack_name1 CHAR(32), init CHAR(1);
            DECLARE 1 stack_descriptor BASED(sp),
                      2 last_stack_descriptor PTR,
                      2 name CHAR(32),
                      2 top_pointer PTR;  /* descr. stack */
            DECLARE sp PTR;
            DECLARE stack_descriptor_top PTR STATIC
                                          INIT(NULL);
            wp = get_stack_descriptor(stack_name1,
                                      stack_descriptor_top);
            IF wp ^= NULL THEN SIGNAL error; /* stack has
                           never been previously created */
            ALLOCATE stack_descriptor;
            sp->last_stack_descriptor=stack_descriptor_top;
            stack_descriptor_top = sp;          /* new top */
            sp->name = stack_name1;        /* name of stack*/
            ALLOCATE stack_element;  /* get first element */
            sep->value = init;   /* and set to init value */
            sep->last_stack_element = NULL;
            sp->top_pointer = sep; /* point to top of stk */
            RETURN;

push:       ENTRY(evalue, stackname2);
            DECLARE evalue CHAR(1), stackname2 CHAR(32);
            wp = get_stack_descriptor(stackname2,
                                      stack_descriptor_top);
            IF wp = NULL THEN SIGNAL error; /* not found  */
            ALLOCATE stack_element;
            sep->value = evalue;
            sep->last_stack_element = wp->top_pointer;
            wp->top_pointer = sep;
            RETURN;
```

fig. 4-3(page 1)

```
pop:        ENTRY(stack_name3);
            DECLARE stack_name3 CHAR(32);
            wp = get_stack_descriptor(stackname3,
                                      stack_descriptor_top);
            IF wp = NULL then SIGNAL error;  /* not found */
            sep = wp->top_pointer;
            wp->top_pointer = last_stack_element;
            FREE stack_element;
            RETURN;

top_element:
            ENTRY(stack_name4)RETURNS(CHAR(1));
            DECLARE stack_name4 CHAR(32);
            DECLARE top_value CHAR(1);
            wp = get_stack_descriptor(stack_name4,
                                      stack_descriptor_top);
            IF wp = NULL THEN SIGNAL error;  /* not found */
            wp = wp->top_pointer;           /* indirection */
            IF wp = NULL THEN SIGNAL error; /* no element */
            top_value = wp->value;
            RETURN(top_value);


            END stack;
get_stack_descriptor:
            PROCEDURE(ename,q)RETURNS(PTR);
            DECLARE ename CHAR(32);
            DECLARE (q,mp) PTR;
            DECLARE not_found BIT INIT('1'B);
            mp = q;          /* call by ref tricks avoided */
            DO WHILE (not_found & mp ¬= NULL);
               IF mp->name = ename THEN not_found = '0'B;
               ELSE mp = mp->last_stack_descriptor;
            END;
            RETURN(mp);
            END get_stack_descriptor;
```

fig. 4-3(conclusion)

Obviously, fig. 4-3 needs some explanation as to what is going on. The overall intent of the program is to allow the creation and manipulations of more than one stack. This requirement is solved by construction a stack_descriptor list whose function it is is to keep track of both the stacks created and the elements of each stack. Suppose, for instance, that outer program began with:

```
CALL create_stack('operand', '$');
CALL create_stack('operator', '!');
```

Then fig.4-4 depicts the results of these calls. If the statements:

```
CALL push('operand', 'a');
CALL push('operand', 'b');
```

follow, the resulting modifications are illustrated in fig. 4-5. It should be fairly obvious to the reader that push, pop, and top_element correctly perform their intended functions.

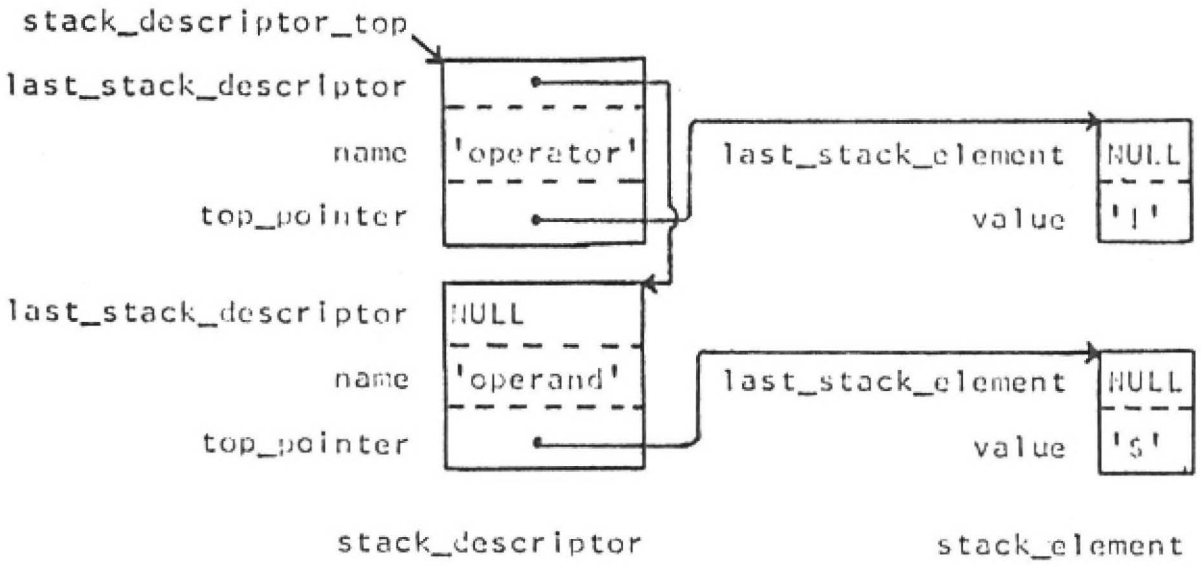

stack_descriptor_top
last_stack_descriptor
name
'operator'
top_pointer
last_stack_element
NULL
value
'!'
last_stack_descriptor
NULL
name
'operand'
top_pointer
last_stack_element
NULL
value
'$'
stack_descriptor
stack_element

fig. 4-4

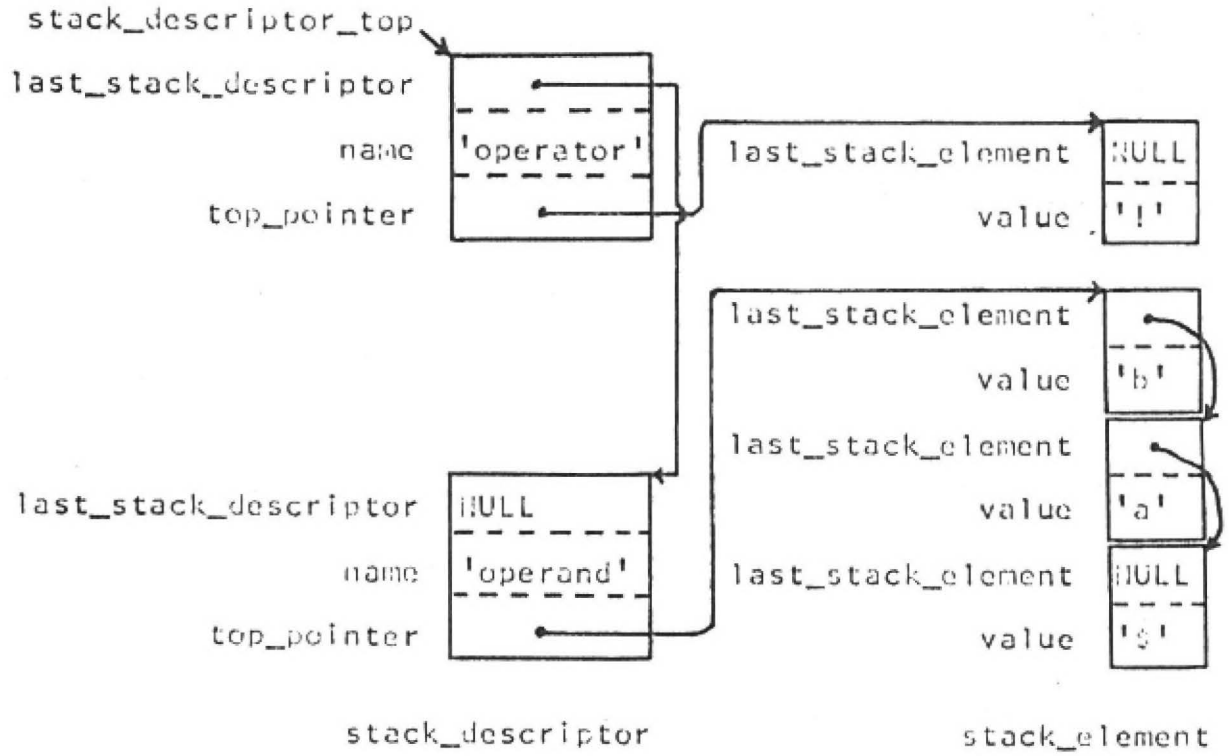

stack_descriptor_top
last_stack_descriptor
name
'operator'
top_pointer
last_stack_element
NULL
value
'!'
last_stack_element
value
'b'
last_stack_element
value
'a'
last_stack_descriptor
NULL
name
'operand'
last_stack_element
NULL
top_pointer
value
'$'
stack_descriptor
stack_element

fig. 4-5

We should make a note at this time concerning our awareness of the fact that instead of naming stacks one might merely have pointers to stacks. The usage of pointers would certainly decrease the stack access time since we would immediately have the location of the stack instead of having to go through the extra procedure get_stack_descriptor in order to find the stack. However, the stack would now be completely available to anyone programming at some level higher than the stack abstraction. He would not have to go through the operators to access the stack but merely use the pointer to the stack to change and information about the stack. For reasons of preventing these accessibility properties, the naming of stacks has been chosen as the method for maintainence of a multiple number of stacks.

Analysis of PL/I

We are now at the stage of analysis--that is, we are ready to judge the merits of PL/I with respect to the facilities it provides us for describing data abstractions. We will take each of the three examples (figures 4-1, 4-2, 4-3) and present first the good points and then the bad based upon the structured programming criteria that must be met.

Regarding the single stack example (fig. 4-1), probably the most important point in PL/I's favor is the fact that the representation of the stack and the operators we introduced

are well-contained within the procedure create_stack. Furthermore the stack, being CONTROLLED by the programmer, can be dynamically allocated and freed, thus allowing the designer a greater flexibility in determining the stack size. In addition, contrary to the usual hazards of flexibility, the requirement that the stack cannot be accessed from outside create_stack is satisfied. Programmer convenience is further increased by the variable attribute STATIC which designates that the last value of top is remembered upon reentering the procedure through create_stack or any one of several ENTRY points (corresponding to operators). And fortunately one is not permitted to access the variable top (thereby possibly changing its value) from outside create_stack. Overall then, an air of safety surrounds the abstraction create_stack.

Now we present the defects of PL/I that this example illustrates. First of all stack is not really a data type as we would prefer but instead a variable. That is, we cannot say that x is a stack since stack is not a data attribute. In our example stack is not only the abstraction we wish to define but also the name of the variable. Obviously to extend this program to handle a multiple number of stacks, we must use more declarations. For example, we would be forced to write something like "DECLARE (stack1(n1), stack2(n2), stack3(n3)) CONTROLLED CHAR(1);" and similarly "DECLARE

(top1, top2, top3) FIXED BIN(15) STATIC;" and finally rewrite the operations to make sure they will know on which stack they are operating. This really points out the fact that our operators have not been written to operate on the representation of some data type but instead on the variable itself. Unfortunately because there is also no syntactic connection between stack$_i$ and top$_i$, the housekeeping chores involved in implementing the usage of more than one stack in this manner become sizeable while the concept of a stack and operators push, pop, and top_element are still simple. The question to consider at this stage is, "Why should the implementation of multiple stacks be so much more difficult than implementation of a single stack when the concept of more than one stack and associated operators has not become more complicated?"

Also regarding the ENTRY statement, we see that one can simulate operation descriptions within a data abstraction. Howvere, a couple of minor criticisms with regard to syntax should be brought to the attention of the reader. We must include each operator section with a RETURN statement--which has no syntactic connection with the operator--or else execution will continue with the next ENTRY statement. It might make more sense to write "END <operation_name>;" (using BNF notation) to terminate each operation block. Secondly, implementation restrictions require us to declare procedures

and entry points at the level from which they are called (unless, of course, one wishes to try his luck with default conditions). This causes a syntactic confusion since the declaration takes the form DECLARE <name> ENTRY... . That is, there is really a second usage of the keyword ENTRY--a word which we depend on heavily.

In defense of PL/I, we must realize though that these last two points are basically implementation and language design considerations that had to be dealt with--although a more satisfactory way of handling these problems might have been invented. However, the first criticism involving the extension of example one to handle more than one stack and, in turn, realizing that this extension problem is caused by the fact that stack is not an abstract data type but the name of a variable seems to be a very serious defect of the language.

The necessity of a multiple number of stacks bring us to example two (fig. 4-2) which introduced a usage of the LIKE attribute. Presented in this example is the favorable impression that stack is now a data type in the sense that we can declare other variables to take the same structure as a stack. As a result, there is no need to even confront the question of programming one stack as opposed to many stacks as had to considered in the previous PL/I example.

However, we must also examine why the usage of LIKE in this manner is not as promising as it initially seems. The declaration of stack must occur outside the stack_ops procedure in order to be understood by declarations of the form "DECLARE...LIKE stack;" (eg. DECLARE operand BASED(q) LIKE stack;). A result of this non-structured requirement is that access to such elements as operand.top is possible at programming levels global to the stack_ops procedure (corresponding to higher levels of abstraction). In fact, one can even allocate storage for "stack" and manipulate its parts, say stack.top, although one would hope this not to be the case; after all, stack is merely supposed to play the role of a data type and not a variable which can take on different values. A part of the stack structure we should examine is the variable m. It is required simple to allow dynamic allocation of the body part of the stack. Even though when considering the representation of a stack m would not come into focus, as it now stands m is as much a part of the stack structure as are body and top.

Suppose we look a little more carefully at the stack_ops procedure. One soon realizes that calls to the different operators rely on the "overlay" implementation of PL/I. For instance if we write "CALL push(q, 'a');", push increments name.top (which is really operand.top) and assigns "a" to the top location of name (really operand.body(operand.top)).

This type of programming is both unclear and implementation-dependent. Note also that in order to group the operator definitions, we designate them as ENTRY points within the procedure stack_ops; however, we certainly hope that stack_ops, itself, will never be called.

The last unfavorable point to be made here is that, in general, one must refer to stacks by pointers rather than names (although the program could have been written using names--see fig. 4-3--but this is more difficult). A side issue to be discussed here is the usage of pointers. Pointers are with respect to data as gotos are to control sequencing--unstructured(18). Perusing a program full of gotos generally means that at some point one will come to a line in the program and not be able to tell how one got there because that spaghetti-like sequencing structure of gotos that has led one to that line has long since been tangled and retangled in his mind. The same holds true for any extensive use of pointers. Once a piece of data has been accesssed, it is often the case that one is really not sure which series of pointers has been used to retrieve this piece of information. These problems of multiple access-paths means that attempts to prove these programs correct will fail almost from the start.

It has already been noted that example three (fig. 4-3)

has gotten around the extensive usage of pointers by accessing stacks by name in conjunction with the procedure get_stack_descriptor. Example three has certainly proven that it is possible to describe in PL/I the stack abstraction (its representation and operations) within a single procedure (called stack in this case). Also we may not access stacks directly from outside the procedure but must go through the stack operations in order to manipulate stacks. In general the messiness within the stack procedure is hidden from the higher level and thus this major requirement of structured programming concerning data and operator abstractions is satisfied.

However, the process of designing the stack procedure of fig. 4-3, while conceptually clear, produces a relatively unclear and unreadable program. Keeping track of both stack descriptors and stack elements is indeed messy and confusing. Also due to our determination to use names instead of pointers we have incurred an additional lower level procedure get_stack_descriptor which was not part of our original stack abstraction. The point to be made here is that overall the complexity of this solution in PL/I in no way reflects the simplicity of concepts involved in solving the problem.

Suppose we now list the major favorable and unfavorable points of PL/I we found when attempting to use the language

for structured programming:

Pro

1) Single stack abstraction consisting of the stack representation and defined operations can be fairly well represented in PL/I (fig. 4-1).

2) Use of the ENTRY statement allow us to define operators as part of the stack abstraction.

3) The LIKE attribute can be used to simulate abstract data types (fig. 4-2).

4) The general stack abstraction can be programmed in PL/I (fig. 4-3).

5) Except for use of the LIKE attribute, the underlying representation of a stack can be made inaccessible from outside the stack abstraction.

Con

1) One is unable to define stack as an abstract data type.

2) Use of the LIKE attribute means that the programmer will be able to directly access the underlying representation of variables declared to have a structure LIKE stack (fig. 4-2).

3) Use of the LIKE attribute means extensive use of pointers (fig. 4-2).

4) In order to represent a general stack abstraction (ie. one that is able to handle a multiple number of stacks) a fairly complex program must be written. While the concepts of the stack abstraction are simple, the PL/I program required to simulate this abstraction is both difficult to write and difficult to read. (fig. 4-3).

Overall Critique of PL/I

We now give a general critique resulting from our analysis of using PL/I in conjunction with the design philosophy of structured programming. A quick glance at the

control features of PL/I shows that with a bit of fixing up it is perfectly reasonable to assume that the control requirements of structured programming can be adhered to. This fix-up would involve adding the CASE statement(4) and eliminating all other control constructs except for concatenation (";"), IF-THEN-ELSE, and DO-WHILE. Furthermore, its block structuring facility is definitely an improvement (over, say, FORTRAN) for describing control abstractions.

Certainly the strict top-down approach with respect to data of program development cannot be adhered to in PL/I and in fact the bottom-up approach is almost always taken. For example, to design the stack program we would first build the stack procedure and test its operators; then the program would be further developed around the stack procedure--creating stacks, pushing on or popping off elements when required for the solution to the problem.

With the ENTRY statement we can syntactically link operators to a data structure, although certainly the designers of PL/I did not intend for the ENTRY statement to be used in this manner. In the first place these operators would have been programmed as separate procedures while the ENTRY statement would be used to signify a secondary point within a procedure where the caller may wish to begin

execution.

The failure of PL/I to provide the programmer with the ability to create abstract data types and make use of these in lower level declarations substantially increases the difficulty of using PL/I in conjunction with structured programming. Certainly a basic concept of structured programming is the representation of programs in terms of abstract data and operators. Abstract operators can be simulated by procedural mechanisms but one is left without any credible facility for representing abstract data types. Although one might argue that the LIKE attribute is a substitute for abstract type declarations, the fact is that nothing could be more untrue. The LIKE attribute deals with the creation of a <u>similar</u> structure to one already defined; an abstract type declaration corresponds to the creation of a <u>new</u> data type (not necessarily a part of the language) and then the declaration of a variable whose range of assumed values and permissible operations are defined by that new type. The LIKE attribute hardly encourages the thought process of abstraction which is the basis of structured programming.

For further analysis of PL/I let us attempt to program the linear list problem outlined earlier in Chapter 3. We want to write a program the defines a linear list and, in

particular, stacks, queues, and dequeues. Each of these lists has a body associated with it and in fact the top level abstraction would simply consist of linear lists and the underlying representation of the body. At a lower level abstraction stacks, queues, and dequeues would be represented as refined linear lists together with particular operators associated with each type of list.

Without thinking too long, one might try to program this in PL/I as sketched in fig. 4-6. However, after a bit of review one should realize that this program is nowhere near a solution to our problem. For one thing, there is only one body and therefore only one list. (To extend this to more than one body would be a somewhat complex problem that conceptually does not appear to be so difficult.) Furthermore we can only access the stack, queue, or dequeue procedure from within the linear_list procedure. This requirement is not necessarily a bad restriction; it just is not what we intended to design. However, if we chose to get around this problem by writing linear_list, stack, queue, and dequeue as four separate procedures, we would be in disaggreement with the results of our abstraction process--that the latter three are lower level abstractions of linear lists.

```
linear_list: PROCEDURE(m, n);
             DECLARE body(m:n) CHAR(1) CONTROLLED;
                ...
             stack: PROCEDURE;
                    DECLARE top FIXED;
                       ...
              push: ENTRY
                       ...
               pop: ENTRY
                       ...
                    END stack;

             queue: PROCEDURE;
                    DECLARE (front, rear) FIXED;
                       ...
             enter: ENTRY
                       ...
            remove: ENTRY
                       ...
                    END queue;

           dequeue: PROCEDURE;
                    DECLARE (leftmost, rightmost) FIXED;
                       ...
            insert: ENTRY
                       ...
            delete: ENTRY
                       ...
                    END dequeue;

             END linear_list;
```

fig. 4-6

One point that has not been previously mentioned is the possibility of "conflicting names". It seems perfectly reasonable that we might have named the operations of queue in fig. 4-6 to be push and pop--although they would act differently that the push and pop operators of stack. (For instance the most recent element placed on a stack gets popped off (LIFO) while the element in the queue for the longest amount of time gets popped (FIFO).) So let us suppose that both the stack and queue procedures had ENTRY points named pop. Certainly at a higher level if we wanted to pop an element off a particular linear list, we would know which pop to refer to if we knew which type of list (a stack or a queue) we had. However, there is no way to do this in PL/I and in fact we would receive a naming conflict error message for having two ENTRY points (whether in the same procedure or not) designated by the same name. In all honesty, there are at least two reasons why we should not expect PL/I to handle this in any different manner: 1)a stack or a queue is not really an abstract data type that can be designated as a variable attribute, and 2)PL/I does not have the facility to allow the programmer to design a data abstraction consisting of a representation and associated operation in a manner that means the language is now extended to include this new abstract data object.

Overall, it should be clear that using structured programming as a design tool for writing programs in PL/I is nearly impossible. When one sees papers titled something like "Structured Programming in PL/I" one can be sure that this paper simply addresses itself to the issue of control and, in particular, leaving out the GOTO statement. This is a start but also close to the end regarding the extent that one can follow the structured programming approach to designing programs in PL/I. (I have never seen the usage of the ENTRY statement included in papers of this sort.) Due to the restrictions of the language, PL/I is far from being able to meet the criteria set forth by structured programming.

PASCAL

Pascal(19) was developed in 1969 by Niklaus Wirth. The design philosophy of the programming language was based on two principal aims: first, to create "a language suitable to teach programming as a systematic discipline," and second, to construct a language implementable as part of a reliable and efficient programming system.

In reference to the latter of these two points, by early 1973(20), Pascal had been successfully implemented on the CDC 6000 and the ICL 1900. Furthermore, implementations on the IBM 360, Sigma 6, CII 10070, and PDP-10 are in progress at various locations.

In order to evaluate the success of the first aim, we can refer to Wirth's recent book Systematic Programming: An Introduction(21) which uses Pascal as its programming language. In this book, Wirth is very much concerned with the explanation of systematic programming techniques, many of which are similar to concepts of structured programming. Wirth shows how programs designed in this fashion can be easily written in Pascal.

When analyzing this language with respect to structured programming and, in particular, the representation of abstract data objects, we should consider the following point. Pascal is the only one of the four languages

presented in this paper that was specifically designed around the concept of programming practices considered as a discipline.

Data Structuring Facilities of Pascal

In Pascal data are described by declarations and definitions. Each variable must be introduced by a variable declaration which associates a type and identifier with that variable. A data type defines the set of values that the associated variable may assume. A data type may be directly described in the variable declaration or by means of an explicit type definition.

It should be noted that Pascal is not a block-structured language in the sense that its predecessor ALGOL60 was. Variables are declared either at the beginning of the program--in which case they are local to the whole program, or in a procedure--in which case they are local to that procedure.

While variable declarations in Pascal are similar to those of PL/I, we should take a closer look at types and type definitions. Types are broken down into three categories: simple, structured, and pointer types.

The definition of a simple type indicates an ordered set of values. Simple types are divided into scalar and subrange

types. The standard scalar types in Pascal are Boolean(*), integer, char, and real. Other possible examples of scalar types are:

```
(mammal, reptile, bird, fish, amphibian)
(subcompact, compact, intermediate, full)
(a, e, i, o, u)
```

Consider the following variable declaration:

```
var demon:(subcompact, compact, intermediate, full)
```

Thus "demon" can be assigned precisely any of four possible values--compact, subcompact, intermediate, and full. Suppose, however, that we wish to construct several "car" variables. We might then construct the following type definition:

```
car = (subcompact, compact, intermediate, full)
```

and then program the following code:

```
var demon:car
```

so that "demon" is now of type "car". The subrange of a scalar type creates a new type defined by indicating the lower and upper bound values in the subrange. Two examples of this are (1..10) and (e..u).

(*)
Keywords in Pascal will be underlined.

Structured types in Pascal are characterized by associating a type (or types) with components and indicating a structuring method, of which there are four. The first of these is the array type which is comparable to PL/I arrays except that Pascal arrays are restricted from being dynamically allocated. Examples of array types are:

```
array [1..100] of char
array [1..5,1..5] of integer
```

The second structuring method to consider is called the record type. Each component of a record type is called a field and is designated by an identifier and its type. For example, letting "alfa" denote a character string type, we might write:

```
record name:alfa;
         age:0..99;
       ssnum:integer
```

So far this facility is similar to PL/I structures. Field access is accomplished by the dot (".") notation selection mechanism, also common to PL/I.

One added feature within the Pascal record types is that a tag field denoted by the keyword case may be indicated as part of a record. This powerful facility allows one to specify several optional variants. The actual variant to be used will be determined by the value of the tag field which

is specified by the programmer at or after the time the type is assigned.

For example, suppose that we have defined "list" to be a scalar type such that "list = (stack, queue, dequeue)". Then the following record type could define a linear list.

```
record body:array[-100..100] of char;
   case s:list of
      stack:(top:integer);
      queue:(rear:integer;
             front:integer);
    dequeue:(leftmost:integer;
             rightmost:integer)
```

Then assume at some later point in the program a variable x is declared to have the above type. Then if "x.s", say, evaluates to the constant "queue", associated with that corresponding record type variable is a "body", "rear", and "front". (Since x represents a queue, expressions such as "x.top" and "x.leftmost" are conceptually inaccessible. The precise details concerning storage allocation for the variable and manipulation of "inaccessible" record fields are left up to the implementor.)

Set types are a third method of structuring. Characterized by the keywords set of, set types define the range of values as the powerset of the specified base type. (In fact, in the first version of Pascal, the keyword was powerset.) If we write "set of(red, yellow, blue)", then the

possible values a variable of this type may have are { }, {red}, {yellow}, {blue}, {red, yellow}, {red, blue}, {yellow, blue}, and {red, yellow, blue}. In addition, the operations union, intersection, set difference, and membership are defined for all set types.

The file type, which is the fourth method of structuring, specifies a sequence of components all of the same type. For example, in order to construct the type character string we could write "file of char".

Finally, we mention pointer types of the Pascal language. Pointer types define an unbounded set of values (comparable to memory addresses) pointing to elements of the designated types. For instance, "↑integer" denotes that variables designated to be of this type may point to integers. Pointer types are commonly used in conjunction with the standard procedure new which generates a value of the specified type and returns a pointer to it.

This completes our description of the data facilities available in Pascal. We now give some instances of type definitions just to make clear how one uses them.

```
Vowel = (a, e, i, o, u)
List = (stack, queue, dequeue)
Board = array [1..8,1..8] of Boolean
Text = file of char
Linear_list=record body:array [-100..100] of char;
                   case s:List of
                      stack:(top:integer);
                      queue:(rear, front:integer);
                   dequeue:(leftmost,
                            rightmost:integer)
```

Before we proceed to understand the stack abstraction example given in the next section, we should examine the rules of parameter passing adopted by Pascal. Identifiers introduced in the procedure heading are called formal parameters, and the objects to be substituted for the formal parameters are called actual parameters. There exist four types of formal parameters in Pascal: 1)value parameters ("call by value"), in which case the actual parameter must evaluate to some expression and its value is substituted into the formal parameter (the default case for Pascal), 2)variable parameters ("call by reference"), in which case the actual parameter must be a variable and is substituted for the formal parameter, which must be preceeded by the symbol var, 3)procedure parameters, where the actual parameter must be a procedure identifier, and 4)function parameters, where the actual parameter must be a function identifier.

At this point in the chapter we are ready to read and understand the Pascal solution to the stack abstraction.

Stack Abstraction in Pascal

We would now like to construct a solution to the stack abstraction problem described in Chapter 3. Fig. 5-1 depicts a proposed solution to the problem written in Pascal. First of all we note that stack is a record type consisting of two fields: 1)a "body" which is an array of 100 characters, and 2)a "top" of type integer. To declare variables of type stack we would simply write at the beginning of the program:

```
var s,t:stack
```

Each of the four routines is simple. Three of them--push, pop, and initialize--are written as procedures while top_element is coded as a function. (Functions, as distinct from procedures are characterized by labeling the type of variable to be returned--in this case char.) In all four routines the char and stack parameters (where applicable) are passed by value and variable respectively.

```
stack = record body:array [1..100] of char;
                top:integer

procedure push (y:char; var x:stack);
begin
   x.top := x.top + 1;
   if x.top > 100 then error;
   x.body[x.top] := y
end;

procedure pop (var x:stack);
begin
   if x.top < 1 then error;
   x.top := x.top - 1
end;

function top_element (var x:stack):char;
begin
   if x.top > 100 then error;
   if x.top < 1 then top_element := 'e';
                else top_element := x.body[x.top]
end;

procedure initialize (y:char; var x:stack);
begin
   x.body[1] := y;
   x.top := 1
end;
```

fig. 5-1

The structure selection mechanism says that if x is of type stack then "x.top" and "x.body" are its constituents. Of course "body" is an array type and so in general we would write "x.body[i]" for some integer i such that 0<i<101.

It should be obvious that these routines in conjunction with the stack type definition correctly solve the problem of programming the stack abstraction. We now proceed to analyze this example and Pascal in general, paying particular attention to its data mechanisms with respect to structured programming.

Analysis of Pascal

We are now at the stage of analysis and are planning to examine both the Pascal language and the stack abstraction example. This examination will be based on the criteria established by the structured programming techniques.

Describing first the favorable points of Pascal, we must be impressed by the ease with which one can describe abstract data types using the different Pascal data facilities. This area of data types is such a central issue of the language that one is very much encouraged to think and program in terms of abstract data types.

Just as the data type "stack" was easily expressed within Pascal, so also were operations on a stack. Each of

the four routines is short, correct, and easy to understand. Thus as the operations were conceptually simple to define, so also were they easy to code within the Pascal langauge.

Another favorable point along the lines of the stack abstraction program is that obviously there are no difficulties incurred by attempting to use more than one stack (remember that multiple stack usage in PL/I caused us problems). A declaration such as "var operator, operand:stack;" means that we have declared two variables, "operator" and "operand", both to be of type stack. Furthermore, stack functions are easily expressible, e.g.

```
pop(operator)
push(y, operand)
```

Finally we must again emphasize the outstanding quality that Pascal possesses: clarity and simplicity. This feature combined with the ability to express abstract data types must commit us to be favorably impressed with Pascal.

However, Pascal does have some drawbacks concerning structured programming. We should be aware that a bottom-up approach is still required for writing programs. That is, we are not able to code in terms of stacks and its associated operations before the stack type and operators are defined. This point is minor, however, since it could be corrected with the implementation of a library system which keeps track

of undefined objects.

Pascal has a more serious fault that is well-illustrated by the stack abstraction example. Pascal provides no facility to syntactically link an abstract data type with its associated operators. In the example, the four routines are not defined as part of an encompassing stack type definition but are merely small routines within some larger program. It is furthermore unclear what happens if in a call to push, say, the second actual parameter is not of type stack--the type specified by the corresponding formal parameter. Perhaps some form of automatic conversion takes place. What we really want is for this call to give the programmer an error. In Wirth's book(21), he says that the "type of the actual parameter is determined by the type of the formal parameter, as specified in the procedure heading." This statement seems to imply that the type of the actual parameter need not be specified at the time of procedure invocation which hardly makes sense and furthermore does not answer our question.

A further criticism as a result of our inability to syntactically link a data type with its operators is that the lower level representation of a variable declared to be of that data type is accessible from anywhere in the program rather than only through the data type operations. For

instance in our stack example, if "operand" were declared to be of type stack then we could change the value of "operand.top" from any place in the program. No longer can we be sure that "top" will only be incremented and decremented by routines "push" and "pop" respectively. We cannot even say that "top" is only set within the routine "initialize". Similarly, the "body" of any stack variable can be manipulated from any point within the program.

A related problem is that of name duplication. We cannot have different operators with the same name associated with different data types since there is no syntactic association of operations to types.

One final criticism of Pascal concerns the designer's decision to disallow dynamic array allocation. The lack of this facility means that a great deal of flexibility is taken away from the programmer. For example, in our stack example we may wish to have different size stacks, whose sizes are to be determined at run time. However, this problem can not be solved within the Pascal language.

For the sake of reference, we list those major pro and con criticisms we have made about Pascal.

Pro

1) The programmer has the ability to represent abstract data types and code routines which operate on variables of particular types.

2) No difficulties are incurred from multiple stack usage.

3) Pascal is concise, simple, clear, and understandable--and yet a powerful langauge.

Con

1) Pascal provides no syntactic linkage mechanism to bind operators to their data type.

2) Lower level representation of abstract data types are completely accessible throughout a Pascal program.

3) Pascal does not provide dynamic array allocation.

Overall Critique of Pascal

The most striking observation one makes when learning Pascal is the compactness of the language and yet the richness of the facilities for the construction of data types. Thus it seems that a language need not be so complicated in order to permit the process of abstraction in writing programs. This process is certainly the foundation of structured programming, and for that reason Pascal, looked at in terms of a structured programming language, must be regarded as a step in the right direction.

The control structures of Pascal, which we have for the most part ignored, are perfectly adequate for meeting structured programming requirements. However, it does seem

odd that for all of Wirth's intentions of introducing programming as an art, the goto statement is a part of the language.

Suppose that we look at how one would code the linear list abstraction that was presented in Chapter 3. Fig. 5-2 presents an outline of the program. We notice first that the case construct is both a powerful and useful feature of the Pascal language. It plays the major role in discriminating among the lower level representation and, in turn, the appropriate operations. In terms of clarity and readability, the linear list Pascal program is certainly a success; however, we must be aware that the same structured programming criteria regarding data type and operator association and access restrictions are violated.

```
List = (stack, queue, dequeue)
Linear_list = record body:array[-100..100] of char;
                  case s:List of
                     stack:(top:integer);
                     queue:(rear, front:integer);
                   dequeue:(leftmost, rightmost:integer)

procedure put_on (y:char; var x:Linear_list);
begin
   case x.s of

      stack:begin                                  {push}
               x.top := x.top+1;
               x.body[x.top] := y
            end;

      queue:begin                                  {enter}
               ...
            end;

    dequeue:begin                                  {insert}
               ...
            end;
end put_on;

funtion take_off (var x:Linear_list):char;
begin
   case x.s of

      stack:begin                                  {pop}
               x.top := x.top-1;
               take_off := x.body[x.top+1]
            end;

      queue:begin                                  {remove}
               ...
            end;

    dequeue:begin                                  {delete}
               ...
            end;
end take_off;
```

fig. 5-2

An interesting paper we must consider is an article written by Habermann(22) which criticizes Pascal on several grounds. Habermann points out that one of Pascal's pitfalls concerns its failure to incorporate the ALGOL60 block structuring technique, reminding us that "a sound programming principle is to declare a variable at the place where it is used." One must agree with him on this point especially when looked at from the structured programming view. Block structuring seems to be a valuable technique when programming by levels of abstraction.

With respect to data representation, the main criticism of Pascal that Habermann makes is its failure to distinguish between types and structures. He makes the following definitions: 1)a type defines a domain for the objects declared of that type and determines the operations that can be performed on those objects, and 2)a structure defines a rule for connecting objects into larger units, but operations are not on the structure but are expressed in terms of individual elements of the structure.

It is my belief that Habermann's complaint is justified. Suppose that Pascal had actually provided a means for constructing a stack abstraction that satisfied our criteria. Then from outside the abstraction, stack would be considered a type defined by certain operations (e.g. push, pop);

however, from within the definition of the stack abstraction, stack would be a structure such that it is composed of a body and top, and the operations are coded in terms of these individual components. The fact is that Pascal fails to provide the programmer with the ability to construct the stack abstraction as a syntactic unit, thereby permitting stack to be viewed as both a type and a structure. In general, the language does not distinguish the use of a data abstraction from its implementation. Without such a distinction, the confusion between types and structures is inevitable.

Overall, Pascal falls short of being a suitable tool for structured programming. We must concur though that if our earlier criticisms plus a top-down programming mechanism were incorporated into the language then Pascal would be ideally suited for structured programming. Of course augmenting the language in this fashion is an ambitious step forward, and the Pascal designers knew just how far to proceed without incurring the major problems of designing a structured programming language. However, we must reiterate the point that this language was developed with the concept of systematic programming as its foundation, and from the results obtained it should be obvious that this specification is a beneficial design criterion for any successful language development.

EL1

The programming language EL1 is the work of Ben Wegbreit and was first described in his doctoral dissertation Studies in Extensible Languages(23) in June 1970. Although the philosophy of this language remains unchanged, a more recent description of EL1 can be found in the ECL Programmer's Manual(24) written in September 1972.

One can divide any extensible language into two parts: 1) the core language defined by some set of syntactic and semantic rules, and 2) extension facilities permitting the programmer to design a more powerful language from the small core language. The core language of EL1 does not differ significantly from ALGOL60 or, indeed, any other algorithmic language. Following along the lines of any extensible language, EL1 provides the programmer with a number of facilities for defining extensions so that the programmer can reshape the language to the problem at hand. These extension facilities exist in the following four areas: syntax, data types, operations, and control. Our investigation of EL1 will focus specifically in two of these areas: 1) data type extensions, which allow the programmer to define new data types and new information structures needed to model a particular problem, and 2) operator extensions, which permit the programmer to define new operations on new data types.

The following question must be answered. Using EL1, how easy is it to construct data abstractions, each of which consists of a representation and operations defined on that representation, and yet adhere to the established criteria of structured programming? Before attempting to resolve this question, a description of the relevant data structuring facilities of EL1 must be presented.

Data Structuring Facilities of EL1

We now give a rather detailed yet informal explanation of EL1's data structuring facilities. This description will include treatment of variables, modes, mode-producing operators, data generation, procedure and generic forms, and user-defined mode functions.

We begin as in PL/I with the description of variables. Associated with each variable is its name, mode, scope, and value. For example if we write "DECL one:INT BYVAL 1;"(*) then we can make the following deductions: 1) the variable name is "one", 2) the mode of one is INT, that is its value may take on any integer number, 3) its scope, although really relative to a program, is that block to which the declaration is internal, but excluding all contained blocks to which another explicit declaration of the same identifier is

---

(*)
Upper case letters will be used to signify keywords in EL1.

internal, and 4) its initial value is 1 by the action "BYVAL 1".

There are seven primitive modes in EL1 denoted by the mode-valued constants INT, REAL, LABEL, BOOL, CHAR, NONE, and REF. While the meanings of the first five are fairly obvious, the semantics of the last two should be described. NONE means that no type is associated and is the only means provided by EL1 for denoting that no variable is to be returned from a procedure. REF is equivalent to PTR in PL/I; that is a variable of mode REF is a pointer unrestricted as to the mode of an object to which it may point.

Just as one can define variables of mode INT, thereby restricting them to having INT values, so also can variables be declared to have mode MODE, to which only MODE values can be assigned. For instance, suppose we write:

```
DECL truthvalue:MODE;
truthvalue <- BOOL
```

Then "truthvalue" is of mode MODE and its associated value is BOOL. Thus the result of some conditional part "truthvalue = BOOL" would evaluate to TRUE. Furthermore, we are now able to use "truthvalue" as a data type in variable declarations, for example:

```
DECL marital_status:truthvalue
```

Thus "marital_status" is a variable which can accept values TRUE or FALSE.

However, the use of MODE would be very uninteresting if all we could do is use different variable names in place of our seven primitive modes. We would like to have the ability to create new data types by operating in some fashion on those primitive modes we already have. In order to meet this objective, EL1 provides the programmer with five mode-producing operators: SEQ, VECTOR, STRUCT, PTR, and ONEOF. These are discussed below. We will use m, m1, m2,...,mn to represent modes.

1) SEQ(m): The type result of this application to m is the construction of a <u>length-unresolved</u> row of components, each of mode m. For example, if we write:

```
DECL string:MODE;
string <- SEQ(CHAR)
```

then the mode "string" is defined as a row of any number of characters. The length of a variable declared to be of mode string must be resolved at the time of declaration.

2) VECTOR(i,m): For some integer i, the new type defined is a <u>length-resolved</u> row of i components each of mode

m. If we write:

```
DECL int_array:MODE;
int_array <- VECTOR(100,INT)
```

then the "int_array" is constructed to be type "row of 100 integers".

3) STRUCT(name1:m1,
name2:m2,
.
.
.
namen:mn): Given that name1,...,namen are symbolic names, the resultant mode is the type structure consisting of n fields (components) whose respective modes mi may differ from one another. As an example, suppose we wish to define a type named person as having a name, age and sex. Assuming the existence of our earlier definition of string, the following statements would construct the desired mode:

```
DECL person:MODE;
person <- STRUCT(name:string
                 age:INT
                 sex:BOOL)
```

4) PTR(m1,...,mn): The result of this application is the mode pointer restricted to point to variables of mode m1 or m2 or...or mn. For instance, if the type string were available to us, we could write:

```
DECL string_ptr:MODE;
string_ptr <- PTR(string)
```

Then any variable of type string_ptr may only point to elements of type string. Note the difference between PTR and REF: PTR operates on a mode to produce a new data type and restricts the range of that type; REF is itself a data type that does not make the above restriction.

5) ONEOF(m1,...,mn): The result is the mode union of alternative modes m1,...,mn where a variable of this type takes on a specific alternative based on its initial value. That is if we write:

```
DECL token:MODE;
token <- ONEOF(CHAR,INT)
```

then any variable of type token can have types CHAR or INT associated with it. We should note that a primary use of ONEOF is to describe acceptable types for formal parameters of procedures.

Thus the use of these five operations permits the programmer to construct modes suited for his purposes. In contrast with PL/I, this added flexibility provided by EL1 will become a central issue during the analysis of this language.

We now come to the area of creating objects of any mode--a process which comes under the heading of data generation. EL1 provides the programmer with two forms for generating data. The first is CONST which creates a new instance of some data class; the second is ALLOC which does the same thing as CONST but then returns a pointer to that new instance. For example, suppose we write:

```
DECL st:string BYVAL CONST (string SIZE 25)
```

here st is declared to have type string (unresolved length mode). The object named by st has been created and given a length (as a result of "SIZE 25"). We could also write:

```
DECL sp1:string_ptr;
sp1 <- ALLOC (string OF "Mark")
```

In this case the right hand side of the second statement creates a new object of mode "string", initializes that object to the literal string "Mark", and returns a pointer to the object; the assignment operation copies this pointer into sp1. Naturally the length has been automatically resolved to 4.

Procedures in EL1 are identified by the keyword EXPR (similar to PROCEDURE in PL/I). The procedure name is considered to have mode ROUTINE. We remark here that EL1 is an "expression-oriented" language, i.e. the final value

calculated within a block is considered to be the value of that block. The following procedure calculates the remainder when k is divided by j for integers j and k:

```
Rem<-EXPR(k:INT BYVAL,j:INT BYVAL; INT)(k-(k/j)*j);
```

Rem has parameters k and j which are given specific values whenever Rem is called (e.g. r <- Rem(5,4)). Writing BYVAL implies that the "call by value" implementation will be used. The INT following the ";" within the parameter list designates the type of value to be returned (namely the result of k-(k/j)*j).

Proceeding we might use the Rem procedure in the following (inefficient) routine which determines whether or not a number is prime:

```
Validprime <- EXPR (number:INT BYVAL; BOOL)
              BEGIN
                 DECL b:BOOL BYVAL TRUE;
                 DECL i:INT;
                 FOR i FROM 2 TO (number/2 + 1)
                    WHILE b = TRUE DO
                    [) Rem(number,i)=0 ->
                           b<-FALSE(];
                 b
              END;
```

Note that [)...(] is equivalent to the PL/I BEGIN...END construct and p -> q is equivalent to the PL/I statement IF p THEN q. The effect of the routine Validprime can be expressed as:

$$\text{Validprime}(p) = \begin{cases} \text{TRUE if } p \text{ is prime} \\ \text{FALSE otherwise} \end{cases}$$

where p is some integer.

We might be concerned about what would happen if a REAL number were used as an argument for Validprime. Instead of trying to determine the "standard" default mechanisms (which might prove insufficient for our purposes anyway), suppose we rewrite the routine, calling it Validprime2, in which FALSE will be returned if number is REAL or if it is not prime. This would be programmed as:

```
Validprime2<-EXPR(number:ONEOF(INT,REAL)BYVAL;BOOL)
                GENERIC(number)
                   [REAL] => FALSE;
                   [INT]  =>
                        BEGIN
                           DECL b:BOOL BYVAL TRUE;
                           DECL i:INT;
                              FOR i FROM 2 TO
                              (number/2 + 1)
                                 WHILE b = TRUE DO
                                    [)Rem(number,i)=0
                                       -> b<-FALSE(];
                              b
                        END;
             END;
```

The GENERIC form of EL1 provides us an efficient means for choosing a particular execution pattern within the GENERIC body determined by the mode(s) of the argument(s) of GENERIC (in Validprime2 we examined the mode of "number"). Also the EL1 form "p => q" is read as "if p is TRUE then exit the

block with a value q". For reasons of compilation efficiency, each statement within the GENERIC body must be of the form "p => q".

The last topic we will examine is the area of user-defined functions; we may best approach its explanation by example. Suppose we have defined the type "complex" in the following way:

```
DECL complex:MODE;
complex <- VECTOR(2,INT)
```

Furthermore suppose, after having defined some variable x to be type complex, we wish to access the first integer in x which will represent the real part of the complex variable x. We might write "x(1)"; on the other hand, for the sake of readability, we might wish to write "x.re". In order to use the latter notation, we can define a selection function for "complex" numbers which, in turn, will be called by the constructions x(i) and x.s where i is some integer and s is a symbol. The "complex selection" (coms) function would then be written as follows:

```
coms <- EXPR(a:complex,m:ONEOF(INT, SYMBOL); INT)
        GENERIC(m)
           [INT] => a(m);
           [SYMBOL] => [)m="re" => a(1);
                         m="im" => a(2)
                       (];
        END;
```

However, upon closer inspection one should realize that coms will not work satisfactorily. We said earlier that whenever we use the notation "x(i)" coms will be called. Unfortunately then, once coms is called, it will recurse forever since we are returning one of values a(m), a(1), or a(2)--all of which will force calls back to the selection routine coms.

We now analyze how the designer of EL1 has solved this problem of infinite recursion by introducing the operator "::". Instead of defining "complex" as we did, we should have defined it as:

```
complex <- QL("complex",comc,coma,coms,comp)
                ::VECTOR(2,INT)
```

This says that the mode complex has an underlying representation (UR) consisting of a VECTOR(2,INT) and associated with the name complex are the operations of conversion (comc), assignment (coma), selection (coms), printing (comp), and generation (comg). These operators are identified by position within the QL (quoted list)--that is, QL(1) gives the name of the mode, QL(2) identifies the conversion routine, and so on. These routines will be defined by the user (as we attempted to do for coms) and are to be invoked in place of any default mechanism when that

particular type of operation needs to be performed. For instance, the assignment routine would be invoked when a new value is assigned to an object of the particular mode for which the assignment routine was written. The assignment routine can be called either explicitly by the " <- " operator or implicitly by the OF or BYVAL constructions within ALLOC or CONST. Similar rules exist for the other four operations.

We now introduce LIFT and LOWER as general primitives which permit the user to attribute different modes to the same object. With this ability, user-defined mode funtions are able to manipulate objects of a given mode without recursively calling themselves.

Consider the following example. Let x be a particular complex number, say x = 9+7i. In ZL1, assuming that the assignment routine (coma) is previously defined, we can describe this by:

```
DECL x:complex CONST (complex OF 9,7)
```

Then we can talk about LOWER(x) which refers to the VECTOR(2,INT); specifically, LOWER(x)[1] = 9 and LOWER(x)[2] = 7. Thus use of the LOWER facility gives us the ability to present a corrected version of coms:

```
coms <- EXPR(a:complex,m:ONEOF(INT,SYMBOL); INT)
        GENERIC(m)
           [INT] => LOWER(a)[m];
           [SYMBOL] => [] m="re" => LOWER(a)[1];
                         m="im" => LOWER(a)[2];
                         [];
        END;
```

We thereby get around the recursion problem and successfully define the selection rountine for complex numbers. Note that coms will be called whenever we write x(1), x(2), x.re, or x.im. However, it would also be called if we wrote x(3) or x.sq and so obviously "breakpoints" signifying errors in selection should be placed in appropriate sections of coms. As one would expect, the LIFT function has exactly the opposite effect as the LOWER facility.

Although this description of EL1 has been rather informal, it should give the reader a fairly good idea of the concepts around which the language was developed. In addition, one should have gained a working knowledge of the relevant parts of EL1 in order to understand the examples to be presented (next section) and to accurately examine the language's usefulness with respect to structured programming.

Stack Abstraction in EL1

We now present the stack abstraction written in EL1. Obviously any example can be programmed in many different ways and, in fact, an EL1 program could be written which is

completely analogous to the Pascal solution to the stack abstraction we examined in the previous chapter. However, in analyzing this language or any other, we are attempting to adhere to structured programming requirements and use the language's fullest capabilities to do so. From my readings and discussions with people involved in the design of EL1, it is clear that the ability to program by abstraction was one of the goals that this language design was intended to meet. Thus this example will be written in a style such as the developers of the language would have programmed it.

We begin by constructing "stack" as a user-defined mode function:

```
stack <- MODE("stack", stc, sta, sts, ,stg)
          ::STRUCT(top:INT, body:SEQ(CHAR))
```

Thus "stack" will have an underlying representation of a structure consisting of fields "top", which will represent the topmost filled location of the stack, and "body", which will hold the elements of the stack. Note that the size of the body is unresolved and must, therefore, be resolved at the time of stack generation. Associated with a stack are operations of conversion (stc), assignment (sta), selection (sts), and generation (stg). (I have chosen to ignore the construction of a printing routine as it is not pertinent to this example.) These four operations will correspond to pop,

push, top_element, and initialize respectively.

The first of these routines that we will examine is stc:

```
stc <-
        EXPR(x:stack, fm:MODE; CHAR)
        BEGIN
            DECL f:INT BYVAL LOWER(x).top;
            DECL temp:CHAR;
            fm ~= CHAR  =>  TYPE_FAULT(stack,fm);
            f=0 => BREAK("stack empty");
            temp <- LOWER(x).body(f);
            LOWER(x).top <- f-1;
            temp
        END;
```

The routine, stc, will be used as a popping routine. Suppose j and s are variables of types CHAR and stack respectively. Then if we write the conditional statement "s -> j", the conversion routine, stc, will be invoked since s is not of type BOOL. All conversion routines take two arguments: the object to be converted and the desired mode of the converted result. Here the desired mode is the formal mode of the second parameter when it is equal to CHAR. (We should note here that the internal representation of all forms in EL1 is LISP(25) and so stc would be called by (-> s j) such that the "->" is responsible for calling stc with parameters s and j.) Now stc takes over by creating temporaries f, assigned the value "top of s", and temp, in which the value of the block will be stored and returned. Initially, two checks are made to make sure that j is type CHAR and s is not empty. If either of the outcomes is FALSE

then the corresponding system routine, either TYPE_FAULT or BREAK, will be invoked. If both outcomes are TRUE, we set temp, decrement the top of s, and return temp which gets stored into j. Thus we have popped off the top element of the stack s and placed it into the character variable j. We should make a note here that in the previous pop routine in PL/I, we simply decremented the top-of-the-stack pointer but did <u>not</u> return a value. The reason for this difference is only a matter of personel choice and independent of which language we use.

We now present and discuss the assignment routine:

```
sta <-
        EXPR(x:stack, y:CHAR; CHAR)
        BEGIN
           DECL f:INT BYVAL LOWER(x).top;
           f <- LOWER(x).top <- f+1;
           f > LENGTH(LOWER(x).body)
              => BREAK("stack overflow");
           LOWER(x).body(f) <- y
        END;
```

Let s and k be variables of modes stack and CHAR respectively. Then the assignment statement "s <- k" will have the effect of pushing the value of k onto the stack s. The internal representation (<- s k) will initiate a call to the routine sta which, int turn, begins by declaring the integer variable f and initializing its value to the top of the stack s. Then after incrementing top and f, a check is made to see whether or not the stack size of s, which can be

determined by use of the LENGTH function, has been exceeded. If not, then the value of k is pushed onto s.

Next we examine the selection routine:

```
sts <-
        EXPR(x:stack, fd:SYMBOL; CHAR)
        GENERIC ()
           fd = "top" => BEGIN
                            LOWER(x).top ¬= 0 =>
                               LOWER(x).body
                                (LOWER(x).top);
                            BREAK("stack empty")
                         END
           TRUE => SELECTION_FAULT (stack, fd)
        END;
```

If s and y are variables of modes stack and CHAR respectively, then the assignment "y <- s.top", having the internal form (<- y (. s top)), causes the invocation of sts in order to process (. s top). In sts we take advantage (of compilation efficiency) of the GENERIC form with no arguments which in this case has the effect of the PL/I IF-THEN-ELSE...IF-THEN format. First, sts makes sure that we are only selecting the "top" element. If this is TRUE, then upon confirming that the stack is not empty sts returns the value in location top of stack s, namely "LOWER(x).body(LOWER(x).top)". If we try to access any other element of the stack besides the "top" element, the system routine SELECTION_FAULT will be invoked. Thus the overall effect of sts is the same as the top_element procedure in PL/I.

We now take a look at the final routine, the generation procedure:

```
stg <-
        EXPR(b:BOOL, s:SYMBOL, l:FORM;
              ONEOF(stack, PTR(stack)))
        BEGIN
           b => ALLOC(ANY BYVAL stg(FALSE, s, 1));
           s = "SIZE" =>
                BEGIN
                   DECL n:INT BYVAL EVAL(l.CAR);
                   n LT 0 => BREAK("Cannot CONST
                      stack of negative size");
                   DECL r:stack.UR CONST(stack.UR
                      SIZE n);
                   r.body(1) <- "$";
                   r.top <- 1;
                   LIFT (r,stack)
                END
           BREAK("CONST FAULT--stack CONST only
                   by SIZE")
        END;
```

This routine is explicitly called by the CONST and ALLOC forms. Suppose we write: "DECL s:stack CONST(stack SIZE 100)". Then the job of stg is to generate a stack consisting of the "top" field and a "body" field of 100 CHAR locations and perform certain initializations associated with any stack. In greater detail, upon reading CONST, stg gets called. The parameter b, which the system provides, specifies heap generation if TRUE--in which case stg returns a pointer to the stack object--and stack generation if FALSE--in which case the actual stack object is returned. Then stg makes sure that we have used the SIZE generator (as opposed to the OF or BYVAL possibilities which we have

decided not to permit). Next, the temporary variable n is created and set to the value 100 ("1.CAR"). Given that n is not less than 0, we declare r and construct it as the UR (underlying representation) of a stack having a body of 100 CHAR locations and a top. At this point the generation procedure initializes the bottom element of r and sets top equal to 1. Finally "LIFT (r,stack)" returns the value of r by sharing with stack, thereby constructing s as we wanted.

The stack example is now complete with the possible exception of a printing routine which was felt irrelevant to this context. Thus we are at the point of analysis, taking both the stack abstraction example and the language EL1 and examining them as they relate to our structured programming criteria.

Analysis of EL1

This section, which is concerned with the analysis of EL1, will begin by describing the favorable features of the language with respect to structured programming. (In some situations, it will be easiest to refer back to analogous PL/I discussions to make explanations more clear.)

For instance, when constructing the stack abstraction in EL1 there was certainly no need to consider the problem in terms of one stack or a multiple number of stacks as was the

case in PL/I. The ability to go from concept to program was made easier to code in EL1 as a result of being able to consider stack as a data type rather than a variable. So one of our basic criteria--that of being able to talk about abstract data types--is satisfied within the EL1 language.

Continuing along the lines of data abstraction, it should be noted that one of Wegbreit's goals during the design of EL1 was to give the programmer the ability to construct new operations on new data types. Furthermore with the usage of the "::" form one can actually restrict access of new operations to some particular data type. In our example, for instance, those four operations are restricted to variables of type stack. The fact is that we were able to syntactically represent our complete stack abstraction consisting of a stack representation--STRUCT(top:INT, body:SEQ(CHAR))--and four operators--stc, sta, sts, stg--describing pop, push, top_element, and initialize routines respectively. Thus this structured programming criterion regarding the representation of a data abstraction seems to be met by EL1--at least in this example.

Another important area that must be given a favorable mark is that of access restrictions. For instance, the selection routine (sts, in our case) can restrict what parts of a data type are accessible from outside the definition of

the data type abstraction. Referring to our stack abstraction as it now stands, given that s is a variable of type stack at a higher level than the stack abstraction, we can talk about "s.top". However, if we try to access "s.body(i)" for some integer i, the sts routine will give us a SELECTION_FAULT error. Thus we can control the access of variables.

The last favorable point we should review concerns the introduction and usage of the "::" operator. While the reader has already seen its basic use, it should be noted that the "::" can be used for multiple level descriptions. For instance, we might write "list::stack::..." where the underlying representation of a list is a stack and the underlying representation of a stack is the structure we gave previously. Thus the "string of pearls"(4) description of structured programming commonly used by dijkstra can be programmed in EL1 using the "::" form.

We now proceed to examine the difficulties of using EL1 in conjunction with structured programming techniques. The first issue to discuss is that although we can talk about abstract data types, we must still program in a bottom-up fashion. That is, we must define "stack" as a mode-defined function before we can write "DECL s:stack...". However, the designers of EL1 view this programming restriction as an

implementation decision rather than a result of the language description(26).

A more important argument against the use of EL1 comes up during the examination of the stack abstraction we programmed earlier. One problem that always came up was the necessity of having to use the LOWER operator to avoid the problem of infinite recursion. In defense of this language construct, one might say that LOWER (or something like it) is necessary to solve the recursion problem; however, this problem is a result of the language design in that the only way to operate on a stack is to use some variable of type stack as a parameter of the routines. The fact is that we need not operate on this parameter but rather on the representation of the given stack. What we want is the ability to construct operations which operate on the representation of the abstract data type instead of constantly having to use the LOWER (or LIFT) facility.

A second issue to be taken up against EL1's method of defining a stack abstraction is one that is very obvious. What happens if there are more than five operations to be defined on some abstract data type? Or just as critical is the fact that there might be some operation that cannot be expressed in terms of having to be invoked by selection, assignment, conversion, generation, or printing mechanisms.

Then all one can do is program those "excess" operations or that particular operation as standard routines not to be associated with the particular data type for which it was conceptually defined. This programming restriction, however, introduces many of the same violations of structured programming criteria that occurred in Pascal.

The above defect, in itself, violates the structured programming criterion concerning the ability to associate any operation to a data type. But let us consider the five routines that are part of the user-defined mode function, and in particular suppose we examine the stack abstraction example. The conversion routine, stc, was written as a popping operator. First of all, this routine seems to violate the general meaning of conversion which normally means converting a value represented as one type to the corresponding value represented as another type. Although the stc routine is initially invoked by the requirement for type conversion, the function of the routine is much more than that. The fact is that the function of stc is not even related to type conversion. This type of programming can be exceedingly confusing and obscure. Next, we should note that a program which makes use of the stack abstraction might use "->" to mean pop and "<-" to mean push or top_element depending on the types of variables on which the arrows (<- or ->) operate. On the other hand, the arrows may refer to

assignment and conditional expresions repectively. These ambiguities make for code which can be difficult to understand.

Now suppose we reexamine the issue of accessibility. It was explained earlier that from outside the stack abstraction for some stack variable s, s.body(i) was inaccessible. We noted however that s.top was accessible from anywhere in the program because the selection routine, sts--programmed as part of the stack abstraction--defines the meaning of s.top. Then if we assume that "." is an operator particular to stack and invokes the operation, named sts, when s.top is written, then the program is expressed along the lines of structured programming requirements. A conflict, however, is bound to arise here since if we had not defined sts, then by system default s.top would produce the value of the highest location filled in the stack; on the other hand, with the definition of sts, s.top produces the value in the stack of that topmost location. Thus depending on whether or not "." is specifically user-defined for a given mode the effect of the operator can produce completely different results. But we know that "." is a legal operation on any structure mode and therefore the programmer must be completely aware of which modes have user-defined selection routines and which respond to the default routine--both invoked by the operator ".". The point to be made here is that if any of the five special

functions is not user-defined, then the system provides a meaning for these undefined functions; the programmer must be knowledgeable of which alternative is taken in each of the five cases for each mode used in the program.

To add further complexity to the above situation, a programmer might really decide that he wants to talk about "s.body(i)" from outside the stack abstraction. Consider the following style in which this program could be written:

```
DECL s1:stack.UR CONST(stack.UR SIZE 100);
   .
   .
   .
s1.body(i) <- "some character value";
s1.top <- "some integer value";
```

The mode of s1 is "stack.UR" which is equivalent to "STRUCT(top:INT,body:SEQ(CHAR))", the underlying representation (UR) of the user-defined mode "stack".

Thus although s1 is not really a stack, one might be perfectly happy with this type of programming since by using "UR", the program reads as if s1 is type stack but behaves differently. For instance, one can now write "s1.body(i)" and access the value or assign a value to the ith location in the body of s1. However, if s1 had instead been declared to have mode "stack" then the coding of "s1.body(i)" would have resulted in the invocation of the selection routine, sts, and mean a SELECTION_FAULT error. So the programmer can

seemingly circumvent the access rules.

However, the use of UR is not the only way that access restrictions can be avoided. Suppose that the variable s were declared to have mode stack, and let us examine another use of the LOWER facility. EL1 allows us to program expressions such as "LOWER(s).top" and "LOWER(s).body(i)" for some integer variable i. Each of these expressions invoke the system-defined routine for selection, signified by the dot ("."), instead of calling the user-defined routine, sts. Thus use of the LOWER facility also permits the user to get around access restrictions.

As a final criticism, let me remark that the routine definitions within an EL1 data abstraction can be confusing. For instance, suppose we look at stc in the stack abstraction. This routine was written as a popping procedure for some stack s and some character variable j. The fact is that if one were to write an instruction which pops the top element off s and places it into j, it would seem much more likely to use an assignment statement "j <- s" rather than a conditional statement "s -> j". The latter of these was used though to accomplish popping because conversion was required. It is not so simple as writing something like "pop(s,j)" if we want to keep our stack abstraction intact; thus we must be <u>completely</u> aware of our data routines in that it may

actually be a different (and possibly awkward) type of statement that performs the intended function. If we are not extremely careful in this manner, trouble can result that was not evident during our conceptualization of the problem.

Finally, we list the major pro and con criticisms made of EL1 in this section of analysis:

Pro

1) We can code abstract data types by using the EL1 mode-producing operators.

2) It is possible to associate operations with a particular data type by using the "::" operator.

3) Accessibility restrictions along the structured programming critieria can be established within the user-defined mode functions.

4) The "string of pearls" programming style described by Dijkstra can be represented by the "::" operator.

Con

1) Use of the LOWER facility within the definition of a type abstraction obscures the fact that operations are defined in terms of the underlying representation of the data type.

2)(i) An operation associated with a data type must be written in terms of one of five routines--conversion, assignment, selection, printing, and generation.

(ii) It is impossible to associate more than five operators with a data type.

(iii) The meaning of any of the five user-defined routines can be (and must be in certain cases) completely obscured.

(iv) Code which may invoke user-defined or system-defined routines can be seemingly ambiguous and difficult to understand.

(v) Expressions which invoke user-defined routines may make code difficult to read and understand.

3) Established access restrictions can be avoided by use of both of the UR and LOWER facilities of EL1.

Overall Critique of EL1

In judging EL1 with respect to structured programming, let us first give quick reference to its control structuring mechanisms. Obviously with its loop and routine facilities, EL1 can adequately lead to well-structured programs when considering only control. The GOTO statement should be eliminated especially since the "=>" form takes the place of any structured uses of the GOTO. Also the GENERIC form can be used to simulate the case statement and thus its use is approved by this writer.

Now with respect to data structuring, the first and most obvious point to be praised is EL1's facility which permits the creation of modes and definitions of operations on them. After all, that is a basic requirement of structured programming. However, for some reason, we are limited to five of these operations when in general the programmer may need more. Of course we could define another procedure to operate on some mode but that routine will no more be associated with a particular abstract mode than, say, addition is with integers, e.g. we can also add real numbers and matrices.

Along these lines, the point was brought up earlier that it might be difficult to write an operation of some abstract mode in terms of one of the five standard calling routines--selection, assignment, conversion, printing, and generation. It is also a fact that it is not always clear when these routines are called, explicitly or implicitly as is sometimes the case. One has to be completely aware of how the language implements the usage of these special routines and, in fact, the internal representation of any language form and the knowledge of LISP are required to be part of this awareness. My point here is that it is often the case that the programming of one of these data type routines is seemingly more difficult to accomplish than it ought to be.

Furthermore, one of the basic criteria of structured programming says that while the ability to use a data structure at some programming level higher than its definition is guaranteed, one should not be aware of the lower level details of the structure. An obvious corollary to this requirement is that in no way at the higher level should the programmer be able to access any of these details of description. Unfortunately the UR (underlying representation) and LOWER facilities, while useful in preventing the recursion problem mentioned earlier, contradict the above criterion. For instance, one can discover the representation of stack by writing "stack.UR".

Or if s were a variable of type stack, then one could write "LOWER(s).top" and LOWER(s).body" to avoid the selection routine, sts, and make assignments to the components of s.

Another issue that should be discussed is the view taken that one is able to represent a "string of pearls" using the "::" operator in EL1. Although this type of programming is possible, the difficulty of simulating levels of abstraction is yet another problem. Suppose we take the example of linear_list (given earlier in Chapter 3) which could be refined as either stacks, queues, or dequeues--each of which would have a set of associated operators. The problem is how to represent the type linear_list. Obviously we would appreciate the discriminated union facility (see Hoare(4)); then the linear_list abstraction might be written as "linear_list::(stack;queue;dequeue)" which implies that a linear_list is either a stack, queue, or dequeue. From a structured programming view, this type of facility is one that EL1 should incorporate into its language.

Our overall feeling about EL1 is that it approaches acceptability with respect to the techniques encouraged by structured programming but does not go far enough in satisfying the necessary structuring requirements. The issues that have led us to this criticism of EL1 have already been mentioned. However, before the discussion of EL1 is

ended, one more question must be posed. How easy or difficult is it to use the extension facilities provided by EL1? Because, if it becomes too awkward to use the extension mechanisms, the programmer simply will not attempt to use them. Instead he will do all his programming in the core language and most likely be successful in producing a finished product. However, the final product will not be one resulting from the application of structured programming techniques; thus the issue of whether or not to use an extensible language in conjunction with structured programming is defeated before it can even be considered. My experience was that the extension facilities of EL1 were somewhat difficult to learn, and I believe that the above problem would exist for programmers using EL1. However, the language is fairly young and in a state of change, and so it is still my hope that this problem will be remedied.

Finally, it is my impression that the goal upon which the language was developed--that of being able to construct a data abstraction consisting of an abstract data type and operations which operate on variables of that type--is well-founded. The mistakes that the designers made were to limit the programmer in the way these operations could be constructed and permit too much flexibility in programming around a data abstraction and its imposed restrictions. Thus, although most of the structured programming criteria

can be satisfied by the language, the above design errors mean that, in general, structured programming techniques need not be used to construct EL1 programs.

SIMULA67

Developed in 1967, SIMULA67(27) was designed as a general purpose simulation language by Ole-Johan Dahl, Bjorn Myhrarrg, and Kristen Nygaard at the Norwegian Computing Center. While SIMULA67 includes most features of ALGOL60 as a subset, its augmentations are directed toward the area of simulation. The hope of the designers was that SIMULA67 would be flexible and powerful enough to allow the programmer to orient the language towards specialized fields. To reach this goal, the concept of aggregates useful as building blocks for programming was introduced.

In dealing with large problems with many details, decomposition is of prime importance. The fundamental mechanism for decomposition in ALGOL60 is the block concept. A block contains local variables and procedures; as far as these local quantities are concerned, a block is completely independent from the rest of the program. SIMULA67 was able to extend this notion by considering that the execution of a block would result in a dynamic instance of the block being generated. Block instances provide the capability for generating several instances of a given block together with its local variables and procedures. If we consider that a block could be used for defining a data abstraction, where its local variables and procedures are to represent the lower level description and operators respectively, then already we

are aware of why SIMULA67 might be successful as a structured programming language.

We will examine how the above concepts of SIMULA67 apply not necessarily to simulations as was orginally intended, but to structured programming and the ability to represent data abstractions. However, we first describe some of the important language constructs.

Data Facilities of SIMULA67

The new concept of SIMULA67 in which we are interested is the instance of a block which is called an object. Each object has its own local data and actions defined by a class declaration. Our examination of class declarations will also include a description of object generation.

The general format of a class declaration is as follows(*):

class <main part> (**)

where the <main part> can be defined as:

(*)
Keywords in SIMULA67 will be underlined.

(**)
"<" and">" are BNF meta symbols.

```
<id> (<parameters v1,...,vn>);
   <declarations for v1,...vn>;
   begin
      <declarations for variables x1,...,xn>;
      <declarations for actions a1,...,an>;
      <class body>;
   end <id>;
```

Then <id> in the above definition is the name of the class. The generation of any object belonging to class <id> will have parameters v1,...,vn associated with that generation. The object's local attributes will consist of parameters v1,...,vn, variables x1,...,xn, and actions a1,...,an. When the object is generated, the <class body> will be performed.

To help clarify the preceeding definitions and explanations, consider the following example of a class declaration:

```
class square_matrix(n); integer n;
   begin integer array T[1:n,1:n];
         procedure transpose
            begin integer i,j,temp;
               i := 2;
               j := 1;
               for i to n do
                  for j to i-1 do
                     begin
                        temp := T(i,j);
                        T(i,j) := T(j,i);
                        T(j,i) := temp
                     end;
            end transpose;
         begin integer i,j;
            i := 1; j := 1;
            for i step 1 until n do
               begin for j step 1 until n do
                  T(i,j) := 0
               end;
         end;
end square_matrix;
```

The above declaration defines the class of square_matrix. The associated data of each object of this class are the parameter n, which represents the size of a matrix object, and the local variable T, which holds the elements of the matrix. Also described is the local action transpose which operates on T. Following the action transpose, we have written a section of code which sets the array elements of T to 0. This section, which is the <class body>, is executed upon each object generation.

Suppose that we want to generate two square_matrix objects--one of size 10 and the other of size 25. We would first declare two pointers to reference objects described by the class square_matrix:

```
ref(square_matrix)a,b;
```

(We remark here that the general statement "ref (<class id>) <pointer variables>" really means that the <pointer variables> are bound to an object denoted by the <class id> or any of its subclasses (subclasses will be explained shortly).) Generation of the desired objects is completed as follows:

```
a :- new square_matrix(10);
b :- new square_matrix(25);
```

The built-in function new creates an object of the specified class (in this case "square_matrix") and returns a reference to the object; the operator ":-" (read: "denotes") indicates the assignment of a reference to a reference type variable. Data belonging to the object may be referenced through use of the "dot notation" as the following expressions demonstrate:

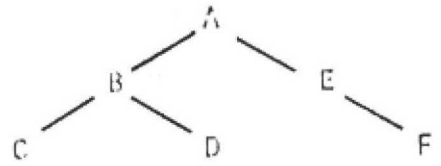
```
a.n
b.T(1,2)
a.transpose
```

The notion of subclasses is the next construct of SIMULA67 we will explore. First of all we extend the definition of a class to include an optional <prefix part>. The syntax now becomes:

```
<prefix part> class <main part>;
```

Semantically we mean that the class for which <prefix part> is the <id> contains the <main part> as a subclass. Then the attribute list of an object corresponding to the <main part> includes those attributes described in the <main part> as well as those described in the definition of the <prefix part>. The two parts are "concatenated" to form one compound object.

Take, for example, the following class hierarchy consisting of six classes A, B, C, D, E, and F.



This model is described in SIMULA67 by the following class declarations:

```
  class A...;
A class B...;
B class C...;
B class D...;
A class E...;
E class F...;
```

If we say that the corresponding lower case letter represents the attributes of the <main part> of an object belonging to that class, then the complete attribute list for each class is depicted in the following table:

class	attribute list
A	a
B	a, b
C	a, b, c
D	a, b, d
E	a, e
F	a, e, f

It should be noted here that if we write "ref(B)x" followed by "x :- new(D)", the attribute list corresponding to x (assuming that no attribute has been declared as virtual--a concept to be explained later) will be a, b and not a, b, d. This fact is a result of having bound x as a reference to objects of class B.

A more concrete example of subclasses might prove useful. Suppose a program is required to handle real and Gaussian numbers. (Gaussian numbers are that subset of complex numbers where the real and imaginary parts are restricted to be integers.) While each type of number may have its own attributes, there are certain characteristics common to both types by virtue of the fact that they are numbers. In outline form, we might code this program as follows:

```
class number;
    begin integer whole_part;
                .
                .
                .
    end number;

number class real_no;
    begin integer dec_part;
                .
                .
                .
    end real_no;

number class gaussian_no;
    begin integer im_part;
                .
                .
                .
    end gaussian_no;
```

Then, for example, the real_no 6.030 would be represented as having a whole_part equal to 6 and the dec_part equal to 030; the representation of the gaussian_no 2+4i would consist of the whole_part and the im_part equal to 2 and 4 respectively.

At this point it is useful to raise the question concerning the possibility of conflicting attribute names. This problem would result from defining two operations with the same name--one defined in a class and one defined in a subclass of that class. Then the attribute list of a reference variable bound to the subclass would contain two operations with the same name.

For instance, using the previous example, suppose that an action "add" were coded as part of the class number and

likewise another action "add" were programmed within the subclass gaussian_no. Then the question of two different "add" actions associated with a gaussian_no object must be resolved. The semantic rules of SIMULA67 stipulate that for any given class if this conflict arises, then only the most locally defined action is associated with that class. Referring to our example, the "add" action defined in gaussian_no is the only "add" operation in the attribute list of gaussian_no. Obviously this solution is what we would hope to be the result.

It should be mentioned, however, that the direction of this binding may be reversed by use of the virtual facility in SIMULA67. Consider the following situation where A is defined as a class and B is defined as a subclass of A. Suppose also that both A and B contain definitions for an operation named "twiddle" and that we have declared a variable x as "ref(A)x". Then if we write "x :- new(B)" followed by a reference to "x.twiddle", we would be referring to the definition of twiddle given in A (because x was bound to the class A). However, if we define twiddle in A to be virtual, the expression "x.twiddle" would access the definition for twiddle given by B. Thus, a top-down programming approach can be partially used for writing a program in SIMULA67. We say "partially" because only procedure labels and switches (see ALGOL60) may be bound as

virtual quantities.

Other features of SIMULA67 include the "group access" feature. Suppose that m references a square_matrix object and we are interested in examining the attributes of m. Using the inspect facility, we can write:

```
inspect m do begin
                .
                .
                .
            manipulations of n and T, use
            of transpose, all these part
            of the object referenced by m.
                .
                .
                .
            end
```

which eliminates the need to use the dot notation for every attribute of a given object. However, it should be clear that the inspect feature is merely "syntactic sugaring" to the language.

Another facility provided by SIMULA67 is instantaneous qualification. For instance, suppose that we have written:

```
ref(gaussian_no)x
ref(number)y
```

Then "x qua number.whole_part" and y qua real_no.dec_part" are legal expressions for accessing the attributes whole_part and dec_part respectively. Thus the qua facility provides an increased flexibility for referencing attributes of

concatenated class objects.

The ability to examine subclasses is also provided by SIMULA67. This feature and use of the when construct are displayed in the next example:

```
ref(number)q;
    .
    .
    .
inspect q when complex_no do begin...end
          when real_no    do begin...end
          otherwise...
```

We have now been exposed to most of the data facility features present in SIMULA67. So we are ready to discuss a SIMULA67 solution to the stack abstraction problem and analyze the language with respect to the structured programming view of data organization.

Stack Abstraction in SIMULA67

A solution to the stack abstraction problem, given in Chapter 3, is coded and presented in fig. 7-1. This representation of the class "stack" is fairly straightforward. An object generated of this class consists of the following attributes:

n--which represents the size of a stack;

body--an array of size n which holds the elements of the stack;

top--which holds the index of the topmost filled location in the body and is initially set equal to zero;

push, pop, top_element, and initialize--operations which perform the desired functions on objects of class stack.

```
class stack(n); integer n;

   begin char array body[1:n];
         integer top;

         procedure push(val); char val;
            begin;
               top := top + 1;
               if top > n then error
                          else body[top] := val;
            end push;

         procedure pop;
            begin;
               if top < 1 then error
                          else top := top - 1;
            end pop;

    char procedure top_element;
            begin;
               if top > 100 then error;
               if top < 1 then top_element := 'e'
                          else top_element := body[top];
            end top_element;

         procedure initialize(y); char y;
            begin;
               top := 1;
               body[1] := y;
            end initialize;

         top := 0;

   end stack;
```

fig. 7-1

A program making use of this class might begin as follows:

```
ref(stack) operator, operand;
    .
    .
    .
operator :- new stack(100);
operand :- new stack(75);
    .
    .
    .
```

Attributes of each stack might then be manipulated in the following style:

```
operator.initialize('!');
    .
    .
    .
operand.push('x');
    .
    .
    .
if operand.top > 100...
    .
    .
    .
inspect operator do begin
                        .
                        .
                        .
                        top
                        .
                        .
                        .
                        body[10]
                        .
                        .
                        .
                        end
    .
    .
    .
```

We will now examine SIMULA67 much more closely and focus our attention on some important issues that have been generated from the solution to the stack problem.

Analysis of SIMULA67

The most favorable feature that SIMULA67 offers us is the ability to syntactically construct a data abstraction as a well-organized cluster of information. This cluster consists of a lower level representation of a data type and operations which perform functions on objects of this type, plus an initialization ability represented by the <class body>. The ease with which this organizational feature can be coded in SIMULA67 is demonstrated by the stack abstraction example. In our SIMULA67 program solution to this problem, we were able to describe a stack type in terms of the "top" and "body" representation and four operations which operated on this lower level representation.

The ability to use parameters as part of a class definition, such as was used in the stack type definition, also proves useful. For instance, our use of the parameter n within the stack class provided us with the ability to allocate stack objects dynamically. The <class body> can also make use of class parameters for initialization purposes.

Continuing our presentation of the favorable features of SIMULA67, we note that this language also provides the programmer with the ability to partially program in a top-down fashion. First, the construction of subclasses encourages us to program in this style; second, the virtual facility permits the programmer to refine a procedure which, in turn, can be invoked by a line of code which initially invoked the older, "original" procedure.

One final point in favor of SIMULA67 should be mentioned. Obviously there is no need to be concerned about multiple stack problems incurred during our use of PL/I. This programming fortune is a result of being able to represent a "stack" as a data type (SIMULA67 calls it a class). Indeed, the simplicity with which the stack abstraction is expressed in SIMULA67 leaves the programmer with a favorable impression of the language.

We now present some problems one finds when using SIMULA67 as a structured programming language. One of the most glaring faults occurs in the area of class definitions. Using our stack abstraction example, we note that a stack can be syntactically represented by a class definition; however, some mathematical properties of stacks are violated. For instance, consider the relation between "top" and "body" such that "top" always points to the first available location of

the "body". In our stack abstraction, "body" and "top" can be accessed from outside the class definition. Not only can these variables be manipulated by invoking anyone of the four defined operations but also by simply writing such expressions as "operator.top" and "operand.body[i]" where operator and operand are references to stack objects and i is some integer. By allowing the latter of these two methods for attribute access, we permit the above mathematical relation to be destroyed.

Another issue which we commented on earlier (see Chapter 4) is the danger that can result from the extensive use of pointers. Unfortunately, SIMULA67 does not give the programmer the choice of whether or not to use pointers. All objects must be pointed to by reference variables, thereby implying that all object attributes are manipulated through the use of pointer variables.

The usage of pointers seems to be the reason for the implementation decision that a legal assignment between reference variables be carried out by "sharing". For example, consider variables x and y declared as references to the same class and x referring to some class object generated by the function new. Then the assignment "y:-x" assigns to y a reference to the object which is the value of x. Thus any change made to an attribute of y results in a "side-effect"

on that attribute of x. Sharing can be useful; however, as the default mechanism for reference assignments, the user must be on constant guard against side-effects.

Finally the pointer issue is further complicated by the fact that the statement "ref(m)n" says, in fact, that n may reference any subclass of m. This unrestrictive feature in conjunction with the qua facility means that any attribute of a class or its subclasses can be accessed, and furthermore that accessibility must proceed through a reference variable. Thus pointers are just too much an integral part of SIMULA67 for us not to strongly object to this mechanism of the language with respect to structured programming.

We now list the important pro and con criticisms we have made of SIMULA67.

Pro

1) The programmer possesses the ability to syntactically organize a data abstraction consisting of a lower level representation and operations operating on the representation.

2) Class parameters achieve a flexibility in generating objects.

3) Subclasses permit the organization of programs into levels of abstraction.

4) The virtual facility in conjuction with the subclass feature provides a partial top-down programming ability.

5) The programming of the stack abstraction in SIMULA67 corresponded in ease to the conceptualization of the solution to the problem.

Con

1) We can access attributes from outside their class definitions.

2) Class objects can be referred to only by pointers.

3) The ref and qua facilities complicate the pointer issue.

Overall Critique of SIMULA67

It is interesting to note that SIMULA67 was developed before most of the concepts propounded by Dijkstra on structured programming were well known; yet, this language captures one very important program design technique--that of providing the programmer with a means for developing an abstraction which consists of a data representation and actions manipulating the representation. In particular, within the syntax of the language, one develops a class consisting of attributes-- data and operations.

Unfortunately, SIMULA67 still falls short as a structured programming language in the following respect. Data attributes, which make up the lower level representation of an abstract data type, are accessible from outside the class definition. Thus the programmer can manipulate these attributes when in fact he should have no knowledge concerning the lower level representation of a data class. The point should be made, however, that due to the clean syntactic representation of a class, it seems that the above

fault could be corrected by signalling an error during compilation if one tries to access a data attribute from outside its respective class definition.

The unfortunate design decision to access objects through pointers is not so easily corrected. As was noted earlier during the discussion of PL/I (Chapter 4), a pointer is data's answer to the control's goto. Especially since ref does not explicitly say to what object a variable may point, it becomes difficult at times, for instance during the usage of qua, to tell where we came from in accessing a given object.

The virtual facility of SIMULA67 and the usage of subclasses prove useful in programming top-down by levels of abstraction. Concerning subclasses, fig. 7-2 demonstrates the ease by which we can code the linear list abstraction in SIMULA67. This program describes stacks, queues, and dequeues all as linear lists with bodies but with different list markers (e.g. top, front) and different operations.

Indeed, it seems that regarding the representation and organization of data abstractions, the designers of SIMULA67 should be praised for their insight. The language provides us with a good foundation for the development of a structured programming language.

```
class linear_list(n,m); integer n,m;
    begin char array body[n:m];
    end linear_list;

linear_list class stack;
    begin integer top;
          procedure push
              .
              .
              .
          procedure pop
              .
              .
              .
    end stack;

linear_list class queue;
    begin integer front,rear;
          procedure enter
              .
              .
              .
          procedure remove
              .
              .
              .
    end queue;

linear_list class dequeue;
    begin integer leftmost,rightmost;
          procedure insert
              .
              .
              .
          procedure delete
              .
              .
              .
    end dequeue;
```

fig. 7-2

CONCLUSION

The purpose of this thesis has been to study various programming languages for two reasons: 1)to see what can be learned from analyzing the use of these languages in conjunction with a structured programming methodology; 2)to determine if it is necessary to develop a new language for structured programming. The success of these languages has been measured in terms of their ability to represent a data abstraction.

Ideally, we wanted a language to support a type abstraction mechanism that would include: 1)the ability to describe an underlying representation which would be unknown to a user of that type; 2)a method by which operations could be defined within the abstraction. We also judged the languages on how well a program represented the conceptual solution to the problem at hand.

Summary

We begin this section by summarizing our findings for each language. PL/I was the first language we analyzed. PL/I, while described as an all-purpose language, limited the user's ability to construct data abstractions. The stack abstraction had to be programmed as a procedure consisting of declarations and entry points--corresponding to a lower level representation and operations. A general solution to the

stack abstraction (i.e. one able to handle a multiple number of stacks) became a complex program that was difficult both to write and to understand.

For all of its power, then, PL/I cannot be used as a structured programming language--its most basic fault being that of restricting the programmer to certain data types. An attempt to circumvent this limitation results in a program that does not reflect the conceptual simplicity of a problem solution.

The language Pascal was relatively simple to understand and use; furthermore, it provided us with the ability to construct data types. However, this construction was basically syntactic and not semantic because we could only describe a data abstraction in terms of a representation. The operations were defined separately from the type definition.

Due to this restriction, our stack abstraction could only be programmed as a definition of the data type stack followed by procedure definitions which operate on a stack. Unfortunately the result could not be considered a well-defined stack abstraction since the definition was not closed. The representation of the stack was totally accessible from anywhere in the program and thus, operations which manipulate the underlying representation of stacks

could be added by any user.

Pascal is an improvement over PL/I in that it allows the construction of data types. It is necessary to conclude, however, that Pascal is inappropriate as a structured programming language.

EL1 was a powerful and complex language that introduced several useful concepts. It certainly permitted us to construct data types and analyze these types with the GENERIC facility. The "::" operator, when used in conjunction with a data type definition, allowed for the construction of a data abstraction consisting of that definition and operations which manipulate the representation of that definition.

Using these features, we were able to construct a program solution to the stack abstraction. The unfortunate problem was that the operations which manipulated stacks had to be written in terms of certain system routines (e.g. conversion, selection). Each operation would then be invoked in place of the corresponding system-defined routine. This programming restriction made the program difficult both to construct and to understand. We also noted that by using the mysterious LOWER feature, the lower level representation of a stack variable was accessible from outside the stack abstraction definition.

The ability to construct data abstractions in EL1 is an issue that was considered by the EL1 designers. This fact is evidenced by their introduction of the "::" operator. However, the restriction regarding operations imposed on the user when constructing a data abstraction makes it difficult, if not impossible (i.e. when more than five operations exists), to design a given abstraction. EL1 in its present stage cannot be considered a language well-suited for structured programming; however, a serious study should be made to investigate how certain changes to EL1 could make it appropriate for structured programming.

SIMULA67 provided us with class definitions whereby objects of a given class could be created and referenced. Associated with each object were certain attributes defined by the class to which the object belonged. These attributes could be viewed in terms of a lower level representation of the object and operations which could manipulate the representation.

A program solving the stack abstraction problem was constructed in a straightforward way and reflected the conceptual solution to the problem. Stack objects could then be generated and referenced. Unfortunately, we found that the representation of stack objects was accessible from outside the class definition.

The concept of class definitions certainly seems analogous to the definition of data abstractions. It is interesting to note, though, that the designers of SIMULA67 must not have thought of a class definition in the same way that we view a data abstraction. The basic difference is that a class is viewed as being defined by its attribute list. This list includes parameter names, simple variable names, and procedure identifiers. A data abstraction, on the other hand, is thought of as consisting of a lower level representation and operations which manipulate the representation.

The SIMULA67 view of the class definition is the cause for our objection to SIMULA67 as a structured programming language. One is allowed to reference any object attribute; so not only is one permitted to invoke a class-defined operation, but one is also allowed to access any part of the lower level description of the object. The latter violates one of our structured programming criteria.

Concluding Remarks

Suppose that we consider the following question: If only operator attributes were made available outside the class definition, would SIMULA67 be suitable as a structured programming language? Our belief is that this restriction would be a simple change to the language, and that if the

answer were yes, then SIMULA67 would qualify as a structured programming language.

Morris(28) provides us with some insight into answering this question. One of the most serious difficulties which arises involves binary operations, for instance "equal". Consider the problem of determining whether or not two stacks are equal. We would like to define "equal" as an operator within the class definition for stack given in fig. 7-1. Without paying attention to the above access restriction, the program outlined in fig. 8-1 would solve the problem.

```
class stack(n); integer n;
   begin integer top;
         char array body[1:n];
         procedure push
            ...
         procedure pop
            ...
    char procedure top_element
            ...
 Boolean procedure equal(x); ref(stack) x;
         begin
            if top ≠ x.top then equal := false
            else begin
               Boolean eq; integer i;
               eq := true;
               i := 1;
               while i≤top & eq do
               begin
                  eq := (body[i] = x.body[i]);
                  i := i+1
               end
               equal := eq
            end
         end equal;
      ...
   end stack;
```

fig. 8-1

If s and t are defined as stack object references, then the expression "s.equal(t)" invokes "equal". The operator "equal", in turn, returns the Boolean value true if the stacks are equal and false otherwise. Now consider any solution to the "equal" operator which does not violate the above access restriction. In contrast to the solution just presented, we are not allowed to use any expressions involving "x.top" and "x.body[i] within the procedure "equal". Instead of writing "x.top", we could construct an operator named "size" such that when we write "x.size", the value of "x.top" is returned. In order to replace "x.body[i]", we might build an operator, calling it "copy"; then the statement "y := x.copy" would invoke "copy" which, in turn, would copy the "body" of x into the array variable "y". We could then proceed to analyze "y". It should be noted that the "copy" operator would also take advantage of the "size" operator. This method of solution, however, involves making changes to our original definition of the stack abstraction merely to facilitate the construction of "equal". The resulting program is also unsatisfactory because it is both more complicated and less efficient than the original program given in fig. 8-1.

Suppose that we relax our restriction by saying that from within a particular class, one can access all attributes corresponding to any object of that class. Then, using our

previous example, while we would not permit the use of "t.top" outside the stack class, it would be acceptable to write "x.top" from within. Of course, this feature would involve changing the implementation to handle this particular case. Even so, it seems that we could imagine problems where the original data abstraction would have to be augmented to compensate for the partial access restriction.

The point to be emphasized is that with the access restriction, operations within class definitions can only be defined as unary operators. Operations which are binary (such as "equal") become difficult, if not impossible, to program or require that additional rules be added just to handle them.

Another problem encountered in SIMULA67 is the following: We might wish for not all operations to be external, that is accessible from the outside of the class definition. Consider once again the stack abstraction. Within its class definition, we might consider defining an "error" procedure which could be called from push and pop if we try to refer to a stack body location which is out of bounds. While there is a need to refer to "error" from within the class definition of stack, there is certainly no reason to permit "error" from being accessed outside the class. We should be able to specify subroutines as being

external or internal to an abstraction.

Thus SIMULA67, even with the added restriction of limiting outside access to operators, is not flexible enough to be completely suitable as a structured programming language. While SIMULA67 may be a start in the right direction, a new language needs to be developed for structured programming.

However, this language should incorporate what we have learned from attempting to use these current languages--especially SIMULA67. Obviously, the ability to describe a data abstraction must be a central concern of the language design. Furthermore, operations declared within a data abstraction should be described as routines which manipulate the underlying representation of the type being defined. (In contrast to SIMULA67, "equal" would be constructed using two parameters corresponding to the underlying representations of stacks s and t.) Also within a type abstraction, one must be able to specify whether or not a given operation can be invoked from outside the abstraction. A language designed around such features is necessary for use in conjunction with structured programming.

BIBLIOGRAPHY

1. R.W.Floyd, "Assigning Meanings to Programs," Proceedings of a Symposium in Applied Mathematics, Vol. XIX, Mathematical Aspects of Computer Science, American Mathematical Society, J.T.Schwartz(ed.), Providence, R.I., 1967, 19-32.

2. J.C.King, A Program Verifier, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., September 1969.

3. R.L.London, "Software Reliability Through Proving Programs Correct," Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1971.

4. O-J.Dahl, E.W.Dijkstra, C.A.R.Hoare, Structured Programming, Acedemic Press, New York, N.Y., 1972.

5. N.Wirth, "Program Development by Stepwise Refinement," Communications ACM, Vol. 14, No. 4, April 1971, 221-227.

6. D.L.Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August 1971.

7. P.Henderson, R.A.Snowdon, "An Experiment in Structured Programming," BIT, 12, 1972, 38-53.

8. Woodger, "On Semantic Levels in Programming," Information Processing 71, 1971, 402-407.

9. P.Naur, "Programming by Action Clusters," BIT, 9, 1969, 250-258.

10. B.H.Liskov, "A Design Methodology for Reliable Software Systems," The MITRE Corporation, Paper 124, Bedford, Mass., 1972.

11. B.H.Liskov, S.Zilles, "Programming With Abstract Data Types," Computation Structures Group Memo 99, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., March 1974.

12. J.R.Slagle, "Automatic Theorem Proving With Renamable and Semantic Resolution," Journal ACM, Vol. 14, No. 4, October 1967.

13. B.L.Clark, J.J.Horning, "The System Language for Project SUE," Proceedings ACM Sigplan Symposium on Languages for System Implementations Sigplan Notices, Vol. 6, No. 9, October 1971, 79-85.

14. H.Mills, "Top Down Programming in Large Systems," Courant Computer Science Symposium 1--Debugging Techniques in Large Systems, June 29-July 1, 1970, 41-45.

15. J.E.Sullivan, "Extended PL/I for Structured Programming," The MITRE Corporation, No. 2353, Bedford, Mass., March 1972.

16. D.E.Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley Publishing Co., 1969, 234-239.

17. PL/I (F) Language Reference Manual, IBM Corporation, White Plains, N.Y. 1972.

18. C.A.R.Hoare, Personel discussions, July 23-August 3, 1973.

19. N.Wirth, "The Programming Language PASCAL," Acta Informatica 1, 1971, 35-63.

20. N.Wirth, Personel communications, January 1973.

21. N.Wirth, Systematic Programming: An Introduction, Prentice-Hall, Inc. 1973.

22. N.Habermann, "Critical Comments of the Programming Language PASCAL," The University of Newcastle Upon Tyne Computing Laboratory, Newcastle, England, February 1973.

23. B.Wegbreit, Studies in Exstensible Languages, Ph.D. dissertation, Applied Mathematics Department, Harvard University, Cambridge, Mass., June 1970.

24. B.Webreit, B.Brosgol, G.Holloway, L.Prenner, J.Spitzen, ECL Programmer's Manual, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., September 1972.

25. C.Weissman, LISP 1.5 Primer, Dickenson Publishing Co., Inc., Belmont, C., 1967.

26. T.Cheatham, Personel discussions, 1972-1974.

27. O-J.Dahl, B.Myhrhaug, K.Nygaard, Common Base Language, Norwegian Computing Center, Oslo, Norway, 1968.

28. J.Morris, "Types Are Not Sets," Conference Record of ACM Symposium on Principles of Programming Languages, Oct. 1-3, 1973.