CMU-CS-78-131

Survey of Scope Issues in Programming Languages

Bob Schwanke

June, 1978

This work was supported in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074.

ABSTRACT

In this paper we shall study scope issues in programming languages, from the standard binding techniques and philosophies of early languages, to the recent work in data encapsulation. First we will study the fundamental concepts of binding, then see how they appeared in early languages. The scope problems in these languages made clear the need for additional program structuring tools, leading to the development of data encapsulation mechanisms. We shall study the scope properties of data capsules, and compare the encapsulation philosophies of several modern languages. We shall use the notion of abstract data types to study modern scope issues, and to survey recent advances in several scope-related areas. Finally we shall compare and contrast several languages, both old and new, by studying solutions in each of them to a common programming problem.

1. Introduction

The scope mechanisms of a programming language are those features which describe and control the use of named entities, such as variables, procedures, and types. As such, they are the notation for describing the structure of programs. The particular scope mechanisms a language provides, by dictating what scope relationships a program may contain, profoundly influence the structure, and thus the quality, of that program. The scope properties of a program affect its understandability, its efficiency, its verifiability, its modifiability, and even the difficulty of finding its bugs. Thus studying scope can shed light both on programming languages and on programming itself.

1.1. Terminology

The term scope has been used to mean any of a large variety of loosely related concepts. In order to use it meaningfully in this paper, I shall assign a single, narrow meaning to it, and define two other terms, *extent* and *range*, to denote two other related concepts.

Before defining those terms, however, I need to make clear my frame of reference. In programming languages a *variable* consists of a name, an object, and a value, though one or more of these may be omitted. In the mainstream of language design, e.g. Fortran, Algol 60, Pascal, and Alphard, a variable is composed of all three. Specifically, a *name* is a program's way of denoting an *object*, which is a portion of memory containing one or more *values*, which may be integers, memory addresses, procedure bodies, or what have you. In languages like LISP and its descendants, however, an identifier denotes a value directly, though some values can be "modified" (CLU calls them *mutable* [31].) In Algol 68, a name is a constant equal to the address of an object. For the purposes of this paper, however, I use the terms variable, name, object, and value in the sense of Algol 60, Pascal, and Alphard. That sense is by far the most common one, and may be used to explain the phenomena of the LISP family and Algol 68 fairly well.

1.1.1. Definition of Scope

We define *scope* to be a property of names. The scope of a name is the portion of the

program text in which all uses of that name have the same meaning. In particular, if the name denotes an object, the scope of that name is the portion of the text in which the name denotes the same object. For example, in Algol 60, when an identifier which has been declared in an outer block is redeclared in an inner block, one says that the scope of the outer name does not include that inner block, and that a different name, spelled the same way, has a scope which is the inner block. As another example, one would be tempted to say that a reference or pointer variable name denotes different objects during the execution of the program. However, a pointer variable is actually a name denoting a single object whose *value* is a reference to (or address of) another object.

1.1.2. Definition of Extent

This concept is a property of objects. The *extent* of an object is its lifetime, that is, the portion of the execution time of the program during which the value contained in the object persists unless explicitly changed. For example, the extent of the object denoted by a local variable in Algol 60 is the period between entry and exit of the block in which it is declared. On the other hand, the extent of an own variable is the entire lifetime of the program, even though its scope is the same as that of a local variable declared at the same place.

1.1.3. Definition of Range

This term, borrowed from Algol 68, denotes language constructs for delimiting scopes and extents[1]. More precisely, a *range* is a portion of a program, delimited by some construct of the language, such that the scopes of names defined inside the program portion do not extend outside that portion unless explicitly "exported" (more on this later). Thus ranges can be thought of as the building blocks out of which scopes are constructed.

In Algol 60, procedures and blocks are the only range delimiters. In Algol 68, almost any statement sequence is a range, if it includes name declarations. In modern languages, a construct which bundles up a group of declarations into an *abstract data type* or *module*, delimits a range.

---

[1]The reader must not confuse this with the subrange concept of Pascal, which denotes an interval within the values of an enumerated type.

For the purposes of this paper, ranges never overlap. When one range is nested inside another, the outer range leaves off where the inner range begins. When one range provides names, objects, or values to a range it declares or invokes, we say only that the providing range is part of the *context* of the using range. Thus in Algol 60, when one block is nested inside another block, the range defined by the outer block does not include the range defined by the inner block, even though the *scope* of a variable declared in the outer block would include the inner block. The outer block is then part of the context of the inner block. Similarly, in languages with "dynamic scope", the range defined by one procedure does not include the bodies of the procedures it calls even though the *scopes* of the *variables* declared in the calling procedure might extend into the called procedures.

1.2. Overview

The history of programming language design, at least that part of it where scope has been an issue, can be divided into two major phases.

During the first phase, which extended from the introduction of Fortran through the late 1960's, languages plainly reflected the compiler technology and the machine architectures on which they were founded. Language comparisons were based on considerations of power and convenience. Usually they were done feature-by-feature. A language designer could ordinarily justify the inclusion of a particular feature simply by showing how conveniently it solved some particular programming problem.

In Chapters 2 and 3, I present a means for categorizing scope control mechanisms, and use it to describe and analyze The mechanisms developed during this phase. One of the major issues of the day was the choice of a parameter mechanism for permitting side effects on the actual parameters. None of the various proposals were completely satisfactory, but the debates served to clarify the nature of the *alias problem*, which is still a major issue in modern language designs.

Early language analysis was more coherent than early language design. Criticism included both analysis of individual features and discussion of medium-scale issues like side-effects and aliases. The more general analyses of the late 60's, combined with the programming methodology research just then emerging, formed the bases for modern languages.

The principal languages which have emerged since the late 60's have been designed based on explicit theories of programming. With a wide variety of suitable scope constructs already available, language designers have tried to choose features whose interaction harmonized with the underlying theory. This concern with interaction also motivated the introduction of several new scope features.

Chapter 4 surveys the major concerns of modern language designers, and explores how these ideas are supported and contradicted by the scope mechanisms of various modern languages. For example, concern for modularity has caused most modern languages to severely restrict the ways that data may be shared between various parts of a program. Similarly, concern for verification must be traded off against generality in the design of module constructs.

The first four chapters of this survey focus primarily on language constructs, and only secondarily on specific languages. However, the importance of the issues discussed cannot be fully appreciated without concrete examples. Therefore Chapter 5 presents a simple programming task, and solutions for it in several languages.These examples trace the development of various scope mechanisms, and show the strengths and weaknesses of the languages displayed.

1.3. Languages discussed in this survey

This paper is a survey of concepts, rather than of languages. Consequently, we will discuss specific languages only to illustrate concepts, and not to list all the languages having that concept. We will draw our examples from a small set of languages which together span the important concepts. However, the two different historical phases of language design require distinctly different kinds of spanning sets. (The references listed in this section are language manuals or language overviews.)

1.3.1. Early languages

During the first phase of language development, three well-known languages contained all the major ideas on scope control. Fortran [36] is important because it was conceived before scope was an issue, and because it is so often the object of ridicule. The mechanisms it

presents have only limited flexibility, but in their simplicity they avoid many pitfalls of more sophisticated constructs. Algol 60 [35] is the best known of a large group of similar languages, and captures the best thinking of a major segment of the computer science community of that period. LISP [55], designed around the mathematician's notion of a function, has engendered another large family of languages. It has a distinctly different set of powerful, general scope features, which have several unexpected properties.

1.3.2. Modern Languages

The modern languages reviewed in chapter 4 have more in common than the ones listed above, because many of the classical design problems of the early phase have been solved. The superficial similarities among these languages make it easier to see the various stands they have taken on a variety of unsolved modern problems. Consequently we will discuss more languages than in the early phase, but only those features of each language which are distinctive.

Pascal [57, 19] is important for its pioneer work in type definition and axiomatic description, as well as for being the basis for half a dozen recent languages. Algol 68 [56, 39] is a transition language, designed with lofty goals, but completed too early to incorporate several crucial modern ideas. Simula 67 [3, 4] is noteworthy as the first language to explicitly attach procedures to data types, as well as the first to provide a type-extension or subtype facility in a safe way. Euclid [26, 48] and Alphard [15, 60] provide abstract data type constructors within the Algol/Pascal line of languages. CLU [49, 29] also provides several modern abstraction mechanisms, within a LISP-like framework. Modula [58] provides module facilities tailored to concurrent programming.

2. A Set of Simple Binding Mechanisms

It would be tempting at this point to develop a formal basis for describing all possible bindings between names, objects, values, environments, et cetera. (Mark Elson has developed one such basis[10].) However, because most of the theoretically possible bindings are impractical, a formal basis would be too cumbersome for this survey. Instead, we shall use an informal basis. R. D. Tennent [54] has recently formulated a simple set of mechanisms for procedure parameters and local declarations, such that the same terms denote the same mechanisms in both contexts. The following sections expand his work into a set of mechanisms sufficiently rich to describe the wide variety of actual mechanisms found in programming languages, but not necessarily general enough to describe every conceivable mechanism.

2.1. Notation

In order not to prejudice the reader by using too-familiar delimiters in my examples, I have created a neutral syntax for example programs. Each example will consist of three columns. The left column contains the range in which new bindings are being made. The right column lists the ranges which form the context from which the new range may obtain names, objects, or values for some of its new bindings. The keywords insert, declare, and invoke mark the exact points where the new range touches its context. Invoke marks the point where control transfers from the context to the range, and returns when the range terminates. Declare marks the point where the name of the range is declared. Insert simultaneously declares and invokes the range. The middle column defines the interface between the new range and its context. Explicit relationships use the symbol :: (double colon) to relate a variable from the new range, on the left, to some piece of the context, on the right. Names occurring free in the left column, and bound by a context in the right column, are listed in the interface column, between angle-brackets.

2.2. Explicit Binding Mechanisms

The mechanisms described in this section each present a way of defining the relationship of an identifier being declared in one range to the objects and identifiers in the range's

context (as defined in section 1.1.3), whether that be a surrounding block (in the case of declarations), or the calling context for a procedure (in the case of parameters).

2.2.1. NEW Variables

A NEW variable is one which has no explicit relationship to the context of its defining range. It consists of a name bound to an object, possibly initialized to a locally computed value. Both the name and the object are normally only accessible within the range in which they are declared. Thus executing range R below would print the value 3:

```
range S =
    new A
    A := 4
endrange S
```

```
interface
new A :: (nothing)
```

```
range R =
    new A
    A := 3
    insert range S
    print A
endrange R
```

2.2.2. VAL Variables

A VAL variable brings into its defining range the value of an object found in the range's context. A VAL variable may not be assigned to. Thus it cannot be used to modify the object from which its value comes. I purposely leave unspecified whether the name is bound to the object or directly to the value. In simple cases (i.e. no aliases or parallelism), it doesn't matter. In early programming languages the VAL mechanism only shows up as a copied value, used as a building block for other mechanisms. In later sections I discuss the perils of aliases and parallelism in some detail. Executing range R in the following example also prints the value 3:

```
range S =
    val A
    ...
    ...
    ...
endrange S
```

```
interface
val A :: B
```

```
range R=
    new B
    B := 3
    insert range S
    print B
endrange R
```

No matter what the range S does, it can't tamper with the object named by B (at least not via the interface mechanism).

2.2.3. VAR variables

A VAR variable brings into its defining range an entire object from the range's context. It consists of a local name bound to that object. This means that any assignment to a VAR variable is also an assignment to the variable (from the surrounding context) which provided the object the VAR name is bound to.

```
range S =
    var B
    print B
    B := B - 1
    print B
endrange S
```

```
interface
var B :: A
```

```
range R=
    new A
    A := 3
    Insert range S
    A := A - 1
    print A
endrange R
```

Executing range R above prints the sequence of values 3, 2, 1, because the names A and B are defined by the interface to denote the same object.

2.2.4. EXPR variables

An EXPR variable actually brings into its range a piece of text from the surrounding context. It consists of a name bound to an expression, which can be any expression which would be legal in the enclosing range. The value of a EXPR variable at any time is the value which would be obtained by evaluating the corresponding expression *in the enclosing range!* If the expression would also be legal as the destination of an assignment statement, an assignment to the EXPR variable becomes an assignment to the object described by the expression. Otherwise, assignment to an EXPR variable has no effect.

```
range S =
    expr A
    expr C
    A := A * 2
    C := C * 3
    print A,C
endrange S
```

```
interface
expr A :: B
expr C :: B+3
```

```
range R =
    new B
    B := 2
    insert range S
    print B
endrange R
```

When the code of range S multiplies A by 2 and stores it back into A, it is manipulating the same object that B is bound to. When it tries to triple B+3, via C, nothing happens. Then when it prints A and C, it gets A's value from that same object, and computes C's value as the current value of B, plus 3. So the program prints 4,7,4.

PROC variables are very similar to EXPR variables, in that they both transport a piece of program text into a range from its context. The only difference is that for an EXPR variable the text is written out explicitly at the binding site, whereas a PROC variable is bound to a procedure previously declared in the surrounding context. That procedure can then be invoked inside the PROC variable's defining range. Most of what I will say about EXPR or PROC variables applies equally to either.

2.2.5. LABEL variables

A LABEL variable brings into its defining range a statement label from the range's context. The LABEL variable is treated exactly as if it were an ordinary label. For instance, GOTO <label variable> causes control to transfer to the statement named when the variable was bound. LABEL variables, like EXPR and PROC variables, permit more complicated interaction between a range and its context. Unlike other mechanisms, however, LABEL variables may be used to affect the control flow, rather than the data, of the range's context.

2.3. Free Name Binding

In mathematics, a variable occurring in a particular context without being defined in that context is said to be free in that context. Many programming languages permit a name to be used in a range in which it is not explicitly defined. To provide a meaning for the name, the language specifies a rule for searching through related ranges to find the declaration which defines it. There are four main concepts involved in such searching:

2.3.1. CLOSED ranges

A CLOSED range is one that cannot contain any free names. Such occurrences would be flagged as errors. This means that all of the interactions between a CLOSED range and its environment will be through explicit bindings. A CLOSED range with no VAR, EXPR or LABEL in its interface would have absolutely no way to cause any external side effects when executed. In the example use of the VAL mechanism in section 2.2.2, I qualified my assertion that the object paired with the VAL variable was safe. With CLOSED ranges, we can remove that qualification. By adding the word CLOSED to the example in that section, we get a range

guaranteed not to cause side effects:

```
closed range S =
    val A
    ...
    ...
    ...
endrange S
```

```
interface
val A :: B
```

```
range R=
    new B
    B := 3
    insert range S
    print B
endrange R
```

Executing range R above must print the value 3.

2.3.2. OPEN ranges

An OPEN range inherits all of the names accessible in the range's context, except for any names it redeclares.

```
open range S =
    new B
    B := A
    A := B * 2
endrange S
```

```
interface
< A >
```

```
range R=
    new A
    A := 3
    print A
    insert range S
    print A
endrange R
```

The expressions involving A in range S refer to the identifier declared at the beginning of range R. Thus the program above prints 3, 6. In general, if the scope of a name N includes a range R, it also includes any OPEN ranges found within R.

2.3.3. DYNAMIC and STATIC ranges

The context of a range actually consists of two parts. The *static* part is the name environment in which the range is *declared*. The *dynamic* part is the name environment in which the range is *invoked*. When a range is *inserted* (see section 1.1.3), the two parts of its context coincide. In the last example, for instance, range R is both the static and dynamic context for range S, because the insert statement in range R both declares and invokes range S.

Free name binding may be done in either the dynamic or static context of a range; we shall label an open range as either STATIC or DYNAMIC according to which context shall be used to bind free names. In the following program, if range S had been marked STATIC, it would

have printed the value 3, found in the variable B in range R, where S is *declared*. Because range S is DYNAMIC, however, the name B inside it refers to the variable declared in range T, where range S is invoked; thus the program prints the value 4.

```
range R=
    declare range S
    new B
    B := 3
endrange R

dynamic range S =          interface
    print B                < B >
endrange S

range T=
    new B
    B := 4
    invoke range S
endrange T
```

2.4. Summary of Mechanisms

The scope mechanisms defined in this chapter are tabulated here for convenient reference:

Mechanism	Shares with context
NEW	nothing
VAL	value
VAR	object
EXPR	part of the execution environment of the context
PROC	procedure, and its free variables.
CLOSED	nothing
OPEN	all variables occurring free in inner range
STATIC	variables in declaration context
DYNAMIC	variables in invocation context

The mechanisms defined in this chapter represent the major concepts behind the scope mechanisms found in early programming languages. In the following chapter I survey those actual mechanisms in detail.

3. Scope Mechanisms in Early Languages

The scope mechanisms in Chapter 2 were carefully defined to be independent of whether the mechanisms were to be used for procedure parameters or variable declarations. However, in the standard early languages, most of the variety in binding mechanisms showed up in procedure parameters. Declarations were ordinarily similar to the NEW mechanism (section 2.2.1). It wasn't until later languages began to worry about initialization that variety in declaration binding mechanisms began to appear.

In this chapter, when I contrast LISP with Algol and Fortran, I will sometimes refer to so-called "pure LISP". LISP was built around the mathematician's notion of a function as a straightforward mapping from one set of values to another, such that one only need think about values and expressions, never about modifiable objects. However, LISP programmers apparently found that the conventional notion of objects was a useful one, because most of the languages in the LISP family have some sort of destructive operations, i.e. operations which modify existing "values" (in other words, objects) instead of creating new ones. These operations form the "impure" part of LISP systems. Without them, the language is free of the notion of object, and thus free from a number of problems.

In the following sections we shall survey and compare the parameter mechanisms and free name mechanisms in Algol, Fortran, and LISP. Then we shall study the notion of a *side effect*, including the Alias Problem, and its manifestation in those languages.

3.1. Free Name Mechanisms

Fortran has no free variables. Every name occurring in a procedure and not explicitly declared in that procedure, is implicitly declared to be a variable with attributes derived from its spelling and the number of subscripts occurring with it. Fortran procedures (main programs and subprograms) are its only range delimiters. The names used in each procedure are private to it. The only mechanism for statically sharing a set of objects among procedures is the COMMON mechanism, which permits sharing of storage areas, but does not require that different procedures refer to the same location in the same way. Indeed, the declarations which name the objects in a COMMON area must be repeated for each procedure, with no check for consistency between procedures. Thus, what to one procedure looks like a sequence of characters might look to another like integers. This quirk has its uses, but is

prone to errors as well. The labelled common mechanism is powerful for a second reason: it is the only mechanism in Algol, Fortran, or LISP which permits an arbitrary set of procedures to share a set of objects without having to make those objects available to other procedures as well. A colleague, on reading an early draft, pointed out to me that this permits one to write Parnas modules [45] in Fortran.

In LISP, all variables are either global variables or formal parameters, and all ranges are DYNAMIC. Free names are handled in it the same way they are in mathematics: names free in one expression are subject to bindings occurring in the next enclosing expression. If no binding can be found in this manner, the name in mathematics is left unbound, (implicitly quantified "for all"). Analogously, a name which is free in one LISP routine is left unbound until the routine is invoked in some context; then the names are bound to the definitions provided for them by the calling environment.

Algol 60 ranges are STATIC . Procedures are defined in terms of blocks, which can be textually nested in other blocks. A name free in one block derives its meaning from the textually enclosing block. If it is also free in that block, the one enclosing it is checked next, and so on.

By avoiding free names altogether, Fortran also avoided some of the pitfalls of Algol 60 and LISP. The named COMMON mechanism, although permitting sharing of objects, provides no assistance in maintaining the integrity of those objects. (To its credit, Fortran was designed before dividing up programs into lots of conceptual units was a serious concern.) The LISP mechanism makes sense in a pure mathematical context, where the "meaning" of a function is independent of the values put into it, whether they are put there explicitly or implicitly. Furthermore, LISP programmers for the most part do not define functions inside other functions, even though they could, so STATIC free name resolution wouldn't be very useful. However, when a procedure can modify variables and not just obtain values from them, the procedure can hardly be understood without knowing which variables its free names denote. Since the parameter mechanism provides a flexible means for obtaining objects from the dynamic context, static inheritance of names, as in Algol, seems to make more sense.

3.2. Parameter mechanisms

3.2.1. Fortran: REFERENCE parameters

The Fortran parameter mechanism was derived from assembly language programming practice: all parameters are passed by address. This has the interesting property that labels and procedure names can be passed as parameters as easily as variables; since no safety checks are made on parameters, no other information need be passed. Thus the Fortran mechanism, often called a REFERENCE parameter mechanism, corresponds to the VAR mechanism in section 2.2.3. (Of course, the power and speed obtained by omitting checking is very unsafe. If a programmer should pass a constant, say 3, to a procedure which expected to store values into its formal parameter, many Fortran systems would have that procedure changing the value of the "constant" 3.)

3.2.2. Algol: VALUE and NAME parameters

Most early languages had no facilities for defining constants, so it isn't surprising that most didn't have pure VAL parameters either. Algol's VALUE parameter, however, is closely related. A VALUE parameter may be thought of as a NEW variable which is initialized at procedure entry with the value of the corresponding actual parameter. It may be assigned to, like any other variable, but because it is a NEW variable, the assignment does not affect the calling context.

Algol has a second kind of parameter, the NAME parameter. It is very much like the EXPR mechanism defined in section 2.2.4. The actual parameter can be any expression which would be legal in the caller's context, with the exception that if the formal parameter occurs inside the procedure as the destination of an assignment statement, the actual parameter must be an expression which would be a legal destination in the caller's context. Thus one could write a procedure for zeroing vectors:

```
begin
procedure zero(vecelement, index, lbound, hbound);
   value lbound, hbound;
   begin
   for index := lbound step 1 until hbound do
      vecelement := 0
   end zero;

integer i;
integer array a[1:10], b[1:10,1:10];
zero(a[i], i, 1, 10);
zero(b[i,i],i, 1, 10);
end
```

The first call to zero in the example above would put 0 in each element of array a. The second call would put 0 in each diagonal element of array b.

3.2.3. LISP: VAL, VALUE, or REFERENCE?

LISP 1.5 differs from pure LISP by including two sets of modifying operators: the SET group, and the RPLACA group. The SET operators change the name-value binding of a variable so that the name is bound to a new value, abandoning the old value. The RPLACA operators modify an existing value, instead of copying parts of it and constructing a new one. Consider the following program (the syntax is contrived):

```
SET ( A, 3 )
SET ( B, A )
SET ( C, 4 )
SET ( D, C )
SET ( B, 5 )
RPLACA ( D, 6)
PRINT ( A, B, C, D )
```

The second operation binds B to the same "3" that A is bound to. The fourth operation binds D to the same "4" that C is bound to. The fifth operation binds B to the value "5", leaving A bound to 3. But the RPLACA operation changes the "4" to "6", so that both C and D are bound to the value "6". The print statement prints 3, 5, 6, 6. Observe that the SET operators do not introduce the notion of objects into LISP. SETting one variable will never change the value of another. It is only the RPLACA group which makes the notion of object distinguishable from that of value.

LISP programmers are well aware of the implications of the SET and RPLACA groups, and will often refrain from using one or both groups in large sections of their programs.

Therefore, I will describe the LISP parameter mechanism as it behaves in each of three versions of the language.

Parameters in pure LISP are VAL parameters. In fact, VAR or EXPR parameters in pure LISP would behave identically with VAL parameters, because only assignment distinguishes VAR from VAL, and only variables with changing values distinguish EXPR from VAL.

Add SET operators to pure LISP, and the parameters become VALUE parameters (sec 3.2.2). That is, the name of the formal parameter is bound initially to the value of the actual parameter, but may later be re-bound (i.e. SET to a different value), without affecting the calling context. Note, however, that SETting a free variable is more confusing in a DYNAMIC range than in a STATIC one.

Full LISP 1.5 makes the parameter mechanism behave somewhat like a REFERENCE mechanism. The RPLACA operator makes it possible to modify the value of the actual parameter, unless SET rebinds it first. But what it really amounts to is that all names are bound to pointers to objects, and all parameters are pointers, passed by VALUE.

What I have described above is the underlying mechanism. LISP values may actually be expressions or functions, which may or may not contain free variables. Ultimately, however, the programmer must always be aware that he is dealing with pointers to objects.

3.2.4. Analysis

Larry Snyder's thesis [53] contains an exhaustive analysis of the computational power of various parameter mechanisms. It shows that VAL, VALUE, REFERENCE, and COPY mechanisms are all equivalent in power, by giving simple rewrite rules for implementing any one of them in terms of any other. The NAME parameter is the only mechanism in his study which could not be rewritten in terms of the others, because of the repeated re-evaluation feature. So the following analysis is based more on considerations such as convenience and efficiency, rather than power.

Each of the parameter mechanisms in Algol, Fortran, and LISP is well suited for certain kinds of computations, independent of the language in which it occurs. As mentioned before, the VALUE mechanism is side effect free, except when the value is a pointer. It also turns out to be cheaper to execute than the others. The mechanisms of LISP are well suited to

symbol manipulation, especially list processing. In addition, by distinguishing between CONS, SET, and RPLACA, the programmer can tell which "assignments" can cause non-local side-effects. The relative merits of REFERENCE and NAME are not as clear. The NAME mechanism permits one to write procedures whose primary purpose is to express some control structure, such as the array sequencing example above. The generalization to other array operations, such as inner and outer products, should be obvious. It also has the delayed evaluation property. That is, the actual parameter will not be evaluated until it is needed. This permits one to pass as parameters expressions which would produce runtime errors if evaluated (e.g. subscript out of bounds), provided that the procedure receives enough information to deduce that it can avoid using the potentially invalid expression. But the fact that the name parameter imports a whole environment into the procedure range implies that the associated overhead must be somewhat high. The REFERENCE parameter is thus appealing because it allows side effects on parameters in a simpler way, and generalizes well to passing procedures, arrays, and labels, although not expressions.

The NAME mechanism turned out to be more powerful than its designers thought. Indeed, it is so powerful that there are some very simple things it cannot do. The most famous example is the Exchange procedure: it is impossible to write a procedure in Algol 60 which exchanges its (integer) arguments, for all possible actual parameters.

The obvious algorithm using a temporary variable will fail when one of the actual parameters is an index to the other:

```
begin
procedure EXCH ( A, B );
   begin
   integer temp;
   temp := B;
   B := A;
   A := TEMP
   end;

integer array A[1:10];
integer I;
I := 1;
A[1] := 2;
EXCH ( A[I], I )
end
```

When the exchange routine puts the value of A[I] into I, A[I] is no longer A[1]. If the routine happens to do its operations in the right order to handle the above case correctly, it will fail

on EXCH(I,A[I]).

The simple array index exchange problem was solved in the late 60's using a non-obvious feature of the assignment statement, namely that the destination address is computed before the source expression is evaluated. Consider the following procedure:

```
procedure EXCH(A,B);
   begin
   integer procedure EX1(M,N);
      begin
      EX1 := M;
      M := N;
      end EX1;

A := EX1(B,A)
end EXCH;
```

This procedure first computes the address of A, then invokes EX1 which saves the value of B, stores the value of A into B, and returns the value of B, which is then stored into A. The critical property is that the addresses of both variables are computed before either is assigned to. But even the above doesn't work, because it is legal to write an actual parameter in Algol 60 which evaluates to a different address every time it is accessed [11]:

```
begin
integer array A[1:10];
integer J;
integer procedure I;
   begin
   J := J + 1;
   I := J;
   end I;

J := 0;
EXCH (A[I],A[I]);
end
```

If this last call were to the "clever" solution above, it would have the effect of copying A[2] into A[1], and A[4] into A[3]!

The problem with the NAME parameter, then, is precisely its strength: the actual parameter must be recomputed on absolutely every examine and store operation. In contrast to that, the address of a REFERENCE parameter is computed exactly once, which makes many situations clearer and simpler.

These difficulties with the NAME parameter are examples of problems with "side effects", which we examine next.

3.3. Side effects

A side effect is any non-obvious effect of executing a statement of a program. There are basically two categories of side effects: a) evaluating an expression may modify variables, and b) an explicit modification may also effect changes to variables not named in the statement.

The principal issue with side-effects is not whether they occur, but how unexpected they are, and how well they can be described. Evaluation side effects permit compact code, but raise serious semantic problems and efficiency issues. Implicit consequences of explicit effects, on the other hand, can often be documented unambiguously.

3.3.1. The Alias Problem

The Alias Problem is the class of problems that comes up when one tries to explain a program in which two names, occurring in the same range, may or may not be simultaneously bound to the same object. In such situations, modifying one variable may affect the value of another. We shall study the problem in the context of aliases created by parameter binding, then see how aliases can occur via free name binding or pointer variables.

3.3.2. Parameter Aliases

In the procedure call EXCH(A[I],I) in section 3.2.4, the object of the actual parameter I was also referenced in the evaluation of A[I]. Thus the exchange routine would unwittingly "modify" one of its parameters when assigning to the other. The REFERENCE mechanism is free of these more sophisticated problems, but still exhibits the very basic problem intrinsic to the notion of multiple names for an object. Consider the following procedure, which divides each of its two parameters by their greatest common divisor, leaving them in their "simplest ratio":

```
procedure simpleratio(A,B);
begin
Integer C,D;
C := A;
D := B;
While C ≠ D do
      If C > D then C := C - D      ! This loop reduces C and D to
      else D := D - C;              !their greatest common
                                    ! denominator

B := B / C;
A := A / C;
end
```

One would hope that a procedure computing the simplest ratio of equal numbers would leave them both equal to 1. But SIMPLERATIO(N, N) would set N equal to zero, assuming N was greater than 1 beforehand. Once again, the reason is that the assignment to B would change the value of A, because both would be bound to the same object.

Algol W [52] has a procedure mechanism, called VALUE RESULT , which avoids the problem of side effects during computation. The VALUE part of the mechanism is the same as in standard Algol: a NEW variable named with the formal parameter name is initialized from the actual parameter. The RESULT mechanism also mandates the creation of a NEW variable, and in addition specifies that when the procedure terminates, the value of the RESULT variable must be copied into the corresponding actual parameter. A VALUE RESULT parameter, sometimes called a COPY parameter, would thus create a NEW variable, initialize it from the actual parameter, execute the procedure, and store its final value back into the actual parameter. Thus in the simplest ratio procedure, the assignment to B would not affect the value of A ever, because A and B would denote separate, local objects. The final copying of A and B back into the same actual parameter would be harmless, because they would have the same value.

Unfortunately, the Algol W form of COPY has a serious flaw, namely that the address of the actual parameter variable is calculated twice: once before entering the routine, to obtain the parameter's value, and again after leaving the routine, to store the resulting value. When an actual parameter is an element selected from an array, side effects on the index variable will change the destination of the value copied out on routine exit. Thus that mechanism still does not solve the Exchange problem, even though the REFERENCE mechanism does so nicely. What's worse, if the same procedure has more than one VALUE RESULT parameter, the order in which the final copies are done is not specified, so that the effect of a problematic call cannot be determined at all. Thus, although call by VALUE RESULT handles SIMPLERATIO(N,N)

correctly, it would fail on something like SIMPLERATIO(A(I),I).

None of the major languages of this period took the obvious step of defining a reference-style COPY mechanism, which would save the address used to obtain the initial value and use it as the destination for copying back the final value of the variable.

3.3.3. Free Name Aliases

Free name binding mechanisms can also create aliases. Consider the following Algol 60 program skeleton:

```
begin
integer I;
procedure P ( A );
integer A;
   begin
    ...
   I := ( ... );
    ...
   I := ( ... A ... );
   end;

  ...
p ( I );
P ( 3 );
  ...
end
```

During the invocation P(I), A and I would be bound to the same object, so that assignments to I would change A. During the invocation P(3), assignments to I would not change A. Thus we see that the meaning of procedure P depends heavily on how it is invoked.

3.3.4. Pointer Aliases

LISP 1.5 has facilities for building rather general graphs, using pointers. Graph manipulations are particularly susceptible to alias problems, because two pointers into a graph may point to the same node, or to a father-son pair, or to two nodes related in some other important way. Several modern languages have attacked this problem; see section 4.4.2.1 for details.

3.3.5. Evaluation side-effects

These come about when one of the components of an expression is a function call (or an EXPR variable bound to an expression containing a function call). There are two problems with such side-effects: a) the effect of the function call on the value of the expression can be obscure, and b) if that effect is precisely defined, the definition forces the code generated to be inefficient.

A function call causes a side effect whenever it modifies a variable whose extent is longer than the function call. (This does not include the pseudo-variable, with the same name as the function, used in some languages to contain the result of the function.) The variable modified may be a parameter to the function, a free variable, or an own variable.

Mathematicians often object to the whole idea of permitting a function to produce side-effects, since this is contrary to the mathematical notion of a function. They argue that a function should always produce the same value for a given set of input values. (They include the values of free variables as inputs). Accordingly, some languages distinguish between procedures, which have side-effects, and functions, which do not. However, we are interested here in functions which both produce side effects and return a value.

The effects of a function call on its parameters and free variables are not in themselves hard to specify. Specifying effects on OWN variables is somewhat difficult, because the effects are only manifested in the results of subsequent calls to the function. The real difficulty comes from the fact that the expression containing the function call may contain other occurrences of the variables and functions involved. Consider the expression

A + F(A)

where F is a function with a VAR parameter. The value of the expression depends on whether A is evaluated before or after F(A). This makes addition be non-commutative, as well as preventing a number of useful optimizations. For example, consider the optimization of Boolean expressions. It is well known that in many such expressions the value of one subexpression can determine the value of the entire expression. For example, (X or true) always evaluates to true. If evaluating X will produce a side-effect, the compiler must produce code to evaluate it, even though the value of the main expression is known at compile time.

A third kind of anomaly occurs when the side effect is a transfer of control, e.g. a GOTO to

a statement outside the function. Which of the effects of the statement containing the function call actually occur?

4. Modern Language Designs

4.1. Overview

In previous chapters we have studied the basic elements of scope mechanisms, and how they show up in early languages. Up to this point, language design has mostly been of an experimental and pragmatic type (i.e. let's try this construct and see if it works any better, is more efficient, etc.) Recent developments in programming theory, however, have provided much firmer foundations for coherent language design. The languages discussed in this chapter have each been based on a formalized theory of programming, with a clearly understood set of concerns motivating the design. We will explore the scope constructs of these languages in the light of the concerns motivating them.

4.1.1. The Software Crisis

In the late 1960's, the computing community became increasingly alarmed over the regularity with which software projects ran past deadlines and over budgets. Contrary to earlier expectations, debugging and modification had come to be an enormous, unpredictable part of the cost of a project (often 50 per cent or more of the total cost [13, 2]). Dijkstra [7] was one of the first to realize that this cost came from the fact that programs were getting too complex to be fully understood, and that better methods for controlling complexity were needed. This idea stimulated research into "structured programming", seeking methods of programming which provide structure strong enough to support the weight of very large, complicated programs. This work had two main themes. One was oriented toward coding, looking at the syntactic characteristics of clear programs. Dijkstra's GOTO letter [5] was a landmark in this area, leading to fruitful discussions about disciplined control flow [25]. The other theme, which emerged from work on program design methods [6], sought organizational tools for dividing up a large program into semi-independent parts [41, 42, 44, 45, 46].

By the end of the decade the combined maturity of language analysis and programming experience was enough to precipitate a new generation of languages rooted both in theoretical ideals and practical experience.

4.1.2. Modern concerns of language designers

The rapid development of new ideas in programming has produced an avalanche of new terms. In order to make the rest of this chapter clear, I present here a brief glossary of the major concerns voiced by researchers studying the quality of programming.

4.1.2.1. Programming Methodologies

The "software crisis" has prompted a variety of attempts to develop methods of programming based on formal principles.

- Structured Programming: Designing a program so that the interrelationships among its parts may be clearly grasped [6, 16]. Note that this is different from a structured program, which is one whose structured design is embedded in the code [60]. Both of the above are different from structured coding, which is a programming standard restricting the ways in which certain language constructs may be used, in order to produce programs with simpler patterns of control flow [1].

- Incremental Development: Constructing a large program a piece at a time, such that each new piece can be written and tested based only on the pieces that have already been constructed.

- Understandability: When referring to programs, the extent to which the program author's intentions are made apparent to the reader. When referring to languages, the extent to which the language provides constructs which permit programmers to express the structure of their programs directly in the code, in natural ways.

4.1.2.2. Overall Structure

The following terms all relate to the structure of programs:

- Modularity: This term has been used in a variety of related senses. In general, modularity is simply the quality of being divided up into coherent pieces. Often this means that each module of a program must be compilable separately. But Parnas [45] has proposed a slightly different notion of modularity, which is even more desirable in a system of programs. He proposes that a modularization of a system of programs be done along conceptual lines, rather than by compilation units. In particular, he advocates that each module make a very small number of assumptions about other modules, and each hard design decision be contained by a single module, so that a design decision, and the corresponding module, may be changed without affecting other modules, so that major design changes cause a minimum of program changes. Put in the terminology of graph theory, Parnas would have us divide a software system into a weakly connected set of strongly connected subsystems. Remember, however, that the division is only at the source code level, not at the machine code level. An appropriate

macro-definition facility can permit an actual machine code routine to be composed of code from several different modules. Parnas's concept implies that in most cases a complex data structure will be accessible within only one module, contrary to the prior practice of spreading knowledge of the format of complex data structures over several different modules. For the remainder of this paper we will ordinarily use the term modularity in Parnas's sense.

- Modifiability: The ease with which a maintainer can locate the set of places where the program text must be changed to accommodate an intended modification. In a highly modular program, most design changes will only affect one module, because the decision being changed pertains only to that module.

- Abstraction: Representing a group of related things by a single term which expresses their alikeness and suppresses their differences. For instance, a procedure with parameters actually describes a large set of possible computations, one for each different set of parameter values. Thus the procedure would be an abstraction denoting the common properties of all the different computations. Abstraction is the principal means by which one can control the complexity of large programs. For instance, the procedural abstraction just mentioned may be invoked in many different places in a program. Each place it is used it will represent the same abstract computation (e.g. binary search), but with different parameters to indicate exactly which version of the computation is meant (e.g. which item is being sought).

- Specification: Independent, concise, precise description of the external properties of a program or subprogram. If a module is to isolate a design decision or conceal the implementation of an abstraction, it must be possible to *specify* exactly what that module does, independently of how it does it [43, 30].

4.1.2.3. Fulfilling Requirements

A program is correct if it completely satisfies its intended purpose [8]. But since intentions are hard to quantify, several other concepts have emerged.

- Verification: proving that a program meets its *specifications*. Ordinarily such a proof must be based solely on the program text, and not on test data [32].

- Validation: Proving properties of a program by executing it on test data, or by embedding executable tests at various points in the program text.

4.1.2.4. Robustness of Programs and Languages

The reliability of a program is the subjective confidence level of its users. But several factors contribute to this confidence.

- Protection: Controlling the rights of different programs to access various data objects and other programs. This should not be confused with security, which is concerned with controlling the flow of information, in the military sense.

- Safety: Invoking an operation whose semantics are not well-defined should not

violate the integrity of user or run-time-system data structures (or programs!).

4.1.2.5. Efficiency

Ideally, a programming language should permit compilers for it to generate generally efficient code, while also allowing the programmer to control code generation and space allocation fairly explicitly in portions of the program where time and space are critical.

4.1.3. Goals of Modern Languages

Algol 68 and Simula 67 are transition languages between the early and modern phases of language design. Algol 68 was designed to be a small, understandable language consisting of a small set of constructs which meshed smoothly to produce a powerful, expressive language [56]. It succeeded well at these goals, but several common combinations of its features proved to be hard to understand, and prone to error. Simula 67 introduced the class and subclass concepts, which were the forerunners of modern data abstraction mechanisms; however, the language appeared too soon to incorporate modern theories of programming.

Wirth brought out Pascal in 1969 with the stated purpose of providing a language which was easy to explain, easily compiled into efficient code, and which encouraged "transparent" (i.e. understandable) programming [57]. CLU, Mesa, and Gypsy were all designed to support structured programming, particularly through data abstraction. Modula and Concurrent Pascal came out as languages which supported modular decomposition of programs, particularly in the realm of multiprogramming. Alphard and Euclid, along with some of the goals mentioned above, were specifically designed to support verification.

4.1.4. Scope Control and Modern Concerns

Modern programming theory has guided the development of scope in languages in a variety of ways. We observe four major areas of influence, which are the subject of the remainder of this chapter.

1. Simple Binding Mechanisms. Modern language designers have reached a consensus on the appropriate uses of each of the standard binding mechanisms. Much of this agreement grew out of studies of side effect problems, as well as out of modern concerns over understandability, verifiability, and programming style.

2. Data Abstraction. Semi-independent results in programming methodology, modularity, specification, verification, and language extensibility have all pointed to the need for a new kind of range which encapsulates a group of related procedures, type declarations, and data objects. Data abstraction mechanisms provide a more powerful, coherent means of describing the structure of a program, and simultaneously take pressure off other scope control mechanisms which would otherwise be used for description purposes for which they were not designed.

3. Relationships among objects, types, and ranges. Previously, ranges coincided with control structures. However, data encapsulation ranges do not, and thus add a new dimension of complexity of possible relationships.

4. Classical Problems. Besides the well known scope problems, e.g. Aliases and Dangling Reference, there are several well-known programming problems which have become clearer and sometimes easier when viewed as scope problems in the light of modern programming theory. Data encapsulation has been a powerful tool for tackling these problems.

4.2. Modern Binding Mechanisms

Early languages, we have seen, contained a variety of interesting binding mechanisms for parameters and declarations. In modern languages we find a substantial degree of uniformity in the selection and use of these mechanisms. In this section we review the historical and theoretical bases for the use ... or disuse ... of the various base mechanisms.

4.2.1. VAL mechanisms

Early languages tended to treat a variable name as always denoting a memory location. Thus when the language designer wanted a parameter mechanism which passed only the value of the actual parameter, he provided a place to put that value, and made it available to the programmer. Similarly, programmer-defined constants had to be stored in memory locations anyway, so no special mechanisms were provided to distinguish constants from variables. But modern theory has recognized the usefulness of truly constant "variables", and at the same time unified the notions of constant declarations and constant parameters. Named constants are a means of localizing design decisions, such as the size of tables and the numeric representation of non-numeric information. Marking a parameter as constant makes clear the point that it will not be modified, thus simplifying understanding. Optimizing compilers can use the constant property to great advantage (e.g. in constant folding, code

motion, and indexing). A constant can be broadcast widely without fear of side-effects. Consequently, modern languages usually provide a means for declaring a variable as having a constant value, and for declaring a parameter either as being a constant whose value is provided by the caller, or as being a read-only reference to a caller-provided object. In most languages only one of these two alternatives are provided; however, Euclid is designed in such a way that the two alternatives are exactly equivalent, so neither is specified exclusively. Many modern languages permit free names for constants, while prohibiting free names for variables, because the latter practice invites side-effects.

4.2.2. VAR mechanisms

We have already pointed out in section 3.2.4 that the VAR parameter mechanism is often cheaper to implement than other side-effect-permitting mechanisms. We also saw by studying the exchange problem that VAR parameters were conceptually simpler than EXPR and VALUE-RESULT parameters. This simplicity translates directly into verifiability. Consequently VAR mechanisms are the dominant form of side-effect permitting parameter mechanism in modern languages. Once again Euclid, which has eliminated parameter-related alias problems, does not distinguish between VAR and VALUE-RESULT, since the two are equivalent in the absence of aliases. This non-specification permits the compiler to choose, on a case-by-case basis, which implementation is most efficient.

4.2.3. EXPR and PROC mechanisms

EXPR mechanisms have largely disappeared from modern language designs. Algol 68 was the last major mainstream language to include them (as a special sort of procedure constant). Experiences with the NAME mechanisms in Algol 60 showed that it was difficult to understand, as well as being expensive to implement. We have already seen that the VAR mechanism is both simpler and cheaper. EXPR's chief enduring value is as a tool for constructing control abstractions, and research in Alphard and CLU (see sections 4.3 and 4.5.1), as well as elsewhere, is seeking to fill the gap there. Similarly, PROC mechanisms have become an endangered species. Algol 68's version is much cleaner than Algol 60's, for it requires that the parameter and result specifications for the PROC parameter be included in the parameter specification for the procedure receiving the PROC parameter. Moses [34] has

unified the implementation problems of name parameters, functional parameters, and functional values into what he calls the Environment Problem, which is the problem of keeping track of the name binding environment in which the parameter or functional value originated. The cost of maintaining this information pervades the entire language system, slowing down compilation and execution even in programs which never use it. In addition, functional values pose a dangling reference problem (see section 4.4.2.1) if the function contains free variables which must be bound in its STATIC context. Functional arguments and values, like name parameters, continue to have value as control abstractions. However, to separate control from data, this writer believes that "funargs" and "funvals" should not be permitted to contain free variables. Without free variables, there are no scope problems to argue against them.

4.2.4. Free Name Mechanisms

DYNAMIC free name binding has been eliminated from the most recent languages, because of its limited utility. Some languages also eliminate free name binding altogether in ranges whose DYNAMIC and STATIC contexts do not coincide, to enhance modularity and reduce opportunities for side-effects.

4.3. Data Encapsulation

The ability to group together and isolate a data structure and the operations defined upon it, has been the single most important recent development in programming languages. In this section we will trace some of the origins of the idea, describe its essential components, and survey its manifestation in modern programming languages.

4.3.1. Origins

Data encapsulation has emerged in response to a variety of modern programming concerns. Each of the concerns has evoked a slightly different notion of what the construct should look like; it is not yet clear whether a single construct can satisfy all the concerns.

One line of research has sought to generalize the notion of a data type to make it possible for a programmer to define his own types. Early languages had only a few base types (e.g. integers, reals, booleans) and only one or two structuring methods (e.g. arrays, records).

Pascal, Simula 67, and Algol 68 pursued data structuring as a reasonable notion of type, and provided facilities for creating and *naming* data structuring templates, so that the same structure could be used for objects created in various places, especially as formal parameters to procedures. This notion drew its theoretical basis from mathematics, which defines a type as a set of values [18]. Algol 68 and Simula 67 both had facilities for associating procedures directly with their types (*classes* in Simula). However, the full impact of this notion was not realized for several years. The last step in this line of development was the notion of restricting access to the representation of a type to the collection of procedures associated with it. We shall discuss this more in the next section.

Parnas, in his studies of large software projects, discovered that decomposing a program according to compilation units was not a conceptually natural method [45]. That is, such decompositions tend to spread across several modules the code implementing a single design decision. In particular, Parnas realized that often times the main concepts in a system are best characterized by data structures, rather than by algorithms alone. Therefore, he proposed decomposing a system into modules which each include both data and procedures, taking special care to conceal all but the most basic design decisions inside modules. This proposal implied that the interfaces between modules had to be as "narrow" as possible, and in particular would not include any elaborate data structures, control blocks, or such things. Parnas modules, then, can be thought of as data structures which can only be accessed via operations defined right along with them in the same module. Note that Parnas intended that his modules only be separate in source form, and that the accessing procedures could be expanded in line when appropriate, rather than incurring the cost of a procedure call each time

Research in program verification has shown data encapsulation to be a powerful tool for simplifying the verification of a program [60]. When the data contained in a module may only be modified by procedures defined in that module, many properties of that data may be verified by regarding the procedures as *predicate transformers*, and doing induction on the number of procedures applied to the data [17]. Verification has brought into focus the concept that the initial values of variables can be vital to the integrity of programs.

4.3.2. A Data Capsule

I shall now define a simple data abstraction mechanism, showing the essential components of such mechanisms, and why those components are useful. In the next section, I'll compare the actual mechanisms found in real languages.

A capsule is a data structuring template, in many respects similar to any of the Pascal structured types, or Algol 68 modes. In particular, a capsule can be the template for creating an unlimited number of objects, each of whose "type" is the name of the capsule. Hereafter, when I use the term *type*, I include capsules as well as other structured and primitive types. A capsule defines a range consisting of three major parts: a representation, a set of operations, and supporting declarations. The representation section contains a set of declarations of data structures which will contain the data of the capsule variable. That is, each variable created from the capsule will contain an instance of each of the data structures declared in the representation. The operations are procedures which may be called by the user of the capsule variable, to examine and modify it. The capsule may define an operation with the reserved name INIT, which is to be automatically called whenever a variable is created from the capsule definition. This routine can then initialize the representation structures so that they have reasonable values in them the first time any operation is applied to the variable. The supporting declarations include whatever procedures, data types, and capsule definitions the capsule creator requires to implement the representation and operations. The documentation of a capsule will always include a set of specifications sufficient to let the capsule be used without inspecting its implementation, and to verify that the implementation satisfies the external specifications. For further introduction to abstract data types, see [28].

A capsule definition may appear anywhere a type definition may appear, and variables may be declared to be of the type defined by the capsule, anywhere the capsule name is accessible.

```
type queue = capsule
   operations size, insert, remove, full, circulate
   representation
      integer array a[1:100]
      integer front, back
   endrepresentation

procedure modincr(i:integer) =
   i := (i mod 100) + 1

procedure init(q:queue) = q.front := q.back := 1

procedure size(q:queue) returns count:integer =
   count := (q.back - q.front) mod 100

procedure insert(q:queue, item:integer) =
   if size(q) equals 99 then fail else
      q.a[q.back] := item
      modincr(q.back)

procedure remove(q:queue) returns item:integer =
   if size(q) equals 0 then fail else
      i := q.a[q.front]
      modincr(q.front)

procedure full(q:queue) returns b:boolean =
   q.size equals 99

procedure circulate (q:queue) returns i:integer =
   insert(q,remove(q))
   i := q.a[q.front]

endcapsule
```

In this example, the capsule ... endcapsule pair delimit a range, so that each of the declarations in it may make use of each of the other declarations. The representation variable names become field selectors for the queue variables passed to the operations. The procedure circulate uses other procedure definitions as well. The operations clause lists the procedures declared inside the capsule which may be used outside it. *These operations defined for the capsule are the only means by which a variable of that type may be manipulated by code outside the capsule!* Thus the writer or reader of a capsule may be assured that he has before him all of the code which is relevant to the data structures in the representation. For example, a program using a queue named M could include the statement

```
if not full(m) then insert(m,3)
```

but that program could not contain the expression

```
m.a[m.front-3]
```

because "a" and "front" are not exported from the queue capsule. Conversely, the only variables which may be manipulated by the code inside a capsule are the variables passed as parameters to it, or declared in supporting declarations. *A capsule may not contain any occurrences of free variables!* Thus the programs which use capsule variables are immune to changes in the implementation of the capsule, so long as the implementation satisfies the capsule's specifications.

In summary, capsules have these important properties:

- Modularity. Many times a design decision will only affect one data abstraction. A capsule gathers into a single range all of the code pertaining to a particular data abstraction.

- Modifiability. A capsule is sufficiently isolated that changes in the design decisions contained in it usually have no effect on any other code.

- Efficiency. Capsules need not be compiled separately. A compiler is free to expand any operation in line at its call sites, if efficiency so dictates.

- Understandability. A capsule variable may be used as if it really were a primitive type in the language, without reference to its implementation. Conversely, its implementation may be understood without reference to how it will be used.

- Verifiability. A capsule correctly implements the abstract data type described in its specification if a) the initial value of a capsule variable represents a legitimate abstract value, b) every operation on a capsule variable transforms legitimate values into legitimate values, and c) the transformation on the concrete representation coincides with the specified transformation on the abstract variable. The two critical properties of capsules here are the concealment of the data structures to prohibit outside access, and the INIT routine to assure initial consistency.

4.3.3. Mechanisms in Modern Languages

Simula 67 and Algol 68 both had mechanisms for associating procedures with data structures (classes and modes, respectively). Algol 68's mechanism seems to have been an accident of its generality; the syntax required to use such a procedure seems exceedingly awkward. Simula 67, however, plainly intended that programmers would ordinarily associate procedures manipulating a class object directly with the class. It also provided a convenient initialization mechanism. Not until recently [40], however, has Simula 67 added a protection facility to conceal some of the names declared in a class definition from code outside the class.

Modula, Euclid, and Alphard are typical of modern languages implementing capsules. Their origins, and resulting mechanisms, however, are very different. Modula modules are very similar to Parnas's module concept, with primary emphasis on isolating a module from its context. Alphard forms were designed to support the design of abstract data types, based on Hoare's verification methodology. Euclid's module facility is a generalization of Modula's, with support for verification.

The module in Modula is not an abstract data type; it is simply a collection of declarations, plus a piece of initialization code for the variables declared there. The interface between a module and its context is completely (except for an initialization problem, discussed later) under the programmer's control. A module is a CLOSED range. Any identifiers brought in from its context must be listed in its uses clause. Any identifiers occurring within it which are to be available outside, must be named in an exports clause. However, not all of the attributes of identifiers may be exported. In particular, a *type* definition may be exported, but its field selectors may not, so that objects of that type may be declared outside the module and passed around as parameters, but their contents may not be examined or modified except within the module. Similarly, variable names may be exported; however, they can only be read outside the module, not modified. Thus a module is responsible for all of the objects it declares, and for the use of all the types it declares. It may release information via exported variables, but need have no fear of side-effects on them. Modula has thus achieved a great deal of flexibility for its mechanism while still maintaining sharply defined boundaries between modules. In particular, although a Modula module is not a type, a module which consists only of one type declaration and associated procedures, would correspond directly to a capsule.

Modula types are not completely protected by its modules. First of all, Modula type definitions do not provide for initialization. Consequently no procedure provided for an exported type may be sure that the variable passed to it has been initialized. Secondly, the language report [58] is ambiguous about whether exported types are forgeable, i.e. whether a procedure expecting a parameter of an exported type will accept a parameter of any type having the same structure. However, Wirth [59] did not intend that Modula should prohibit bad style, but only that it encourage good style, so he probably doesn't care whether exported types are forgeable. He would simply say that the verification of a module assumes faithful initialization for exported types, and no forgery of them. Given those programming

conventions, such an exported type is definitely a data abstraction, and a module containing only one such type and its procedures, would be a fully general capsule.

A Euclid module, though similar to a Modula module, may be used either as a simple collection of declarations, or as a template for a new data type (i.e., as a capsule). When the module is used as a type, the initialization code becomes the initialization procedure for the type, and the variables become the representation of the type. Exported names may be used only in conjunction with the name of the module or the name of a variable whose type is the module. This applies uniformly to exported constants, variables, types, and procedures.

Euclid also has a conventional type mechanism, with initialization based on parameters to the type. If such a type is exported from a module, none of its field selectors are available outside the module, unless exported with it. Even assignment and tests for equality are concealed unless explicitly exported.

Alphard forms are designed to model data types directly. A form definition defines both the representation and the operations for the type being defined, and provides convenient means for incorporating the specifications for the type, such that it may be understood without reference to its implementation. The supporting declarations in an Alphard form may include other form declarations, which may also be exported.

The crucial difference between the types modeled by forms and those modeled by Euclid modules is that form operations may operate on several instances of the form simultaneously, whereas a module operation may only operate on one instance of the module. This latter view is perfectly adequate for many purposes, e.g. stacks and queues. However, consider a capsule implementing sorted lists, which must include an operation to merge two lists. A Euclid module defining a sorted list couldn't do it, except by repeated remove and insert operations, which wouldn't be very efficient. An Alphard form for sorted lists, on the other hand, could easily include an operation which took two such lists as parameters, and accessed the representations of both. This difference in mechanism might well be due to the view taken by Modula, and partially adopted by Euclid, that a module exists to manage resources. It would be strange indeed to merge two lists of objects built from different resource pools. What has in fact happened in Euclid and Modula is that the principal data type definition facility has been separated from the encapsulation mechanism. In Alphard, the usual programming paradigm is to define one type per form, giving abstract and concrete specifications for it. Any data shared among instances of the form is declared specially.

Procedures which access the common data must do so through an instance of the form. In Modula, the normal case is to create a module which declares both variables and types, and procedures which operate on both. Procedures which take parameters of those types may be thought of as operations on them; those that don't are just operations on the main variables of the module. Euclid tried to blend the two ideas, and only partially succeeded. Since a module's variables represent a resource pool, of which there may be several instances, the module definition may be used as the definition for a type. If the module also defines and exports conventional types, they must be accessed through the name of the parent module, e.g.

```
SpaceManager.BlockType
```

If that module is actually a type definition, the BlockType must be accessed through a particular variable of type SpaceManager, so that the block is allocated from the right storage pool. Unfortunately, this means that even types which are not part of resource managers, must still accessed through a module name every time, if they are to be protected. The language report even gives an example of a module implementing floating point numbers, which forces every operation on floating point numbers to give the name of the floating point module as well as the name of the routine -- even though floating point numbers do not share data!

4.4. Relationships among objects, types, and ranges

Scope, after all, is concerned not primarily with individual objects, names, and ranges, but with the interactions between them. In this section we study the ways in which objects, types, and ranges can interact in modern languages. We explore the role of initialization in data abstraction and data integrity. We study the problems of describing and verifying relationships between objects. Finally, we study relationships among types, both when the types are almost unrelated, as with generic types, and when they are closely related, as with exported types.

4.4.1. Initialization

Initialization has become very important in language design, because of issues of safety and verifiability. Uninitialized pointer variables are unsafe, because a dereference for

assignment via such a variable will modify some arbitrary storage location. Initialization is important for verification, because the proof that a data object faithfully represents its abstraction ordinarily starts from the assumption that the object starts out with a legal value [17]. (The alternative is to show that it receives a legal value prior to the first time it is read.)

Initialization facilities in early languages were fairly weak. Of the three studied earlier, none had any facilities whatsoever. In particular, lack of initialization was one of the fatal weaknesses of the OWN construct in Algol 60. An OWN variable is supposed to retain its value between invocations of the block in which it is declared. To use this facility, the code of the block must assume that the variable already has a legal value when execution of the block commences. But during the first invocation, this assumption will be false. So the programmer using an OWN variable had to add a mechanism to check, on every invocation, whether or not it was the first.

Three main strategies have been developed to handle initialization:

- Default Values. This scheme inserts a value in every variable when it is created. It may be an ordinary value, like zero for integers, or it may be the special value UNDEFINED, which causes the program to halt if it is ever examined. Both schemes incur the initialization cost for all variables. The former conceals many of the bugs caused by omitted initialization. The latter requires special processing on every fetch operation, which requires special hardware support to avoid being excessively expensive.

- Explicit initialization. Alphard, Euclid, and Algol 68 all provide explicit facilities for specifying the initial value of a variable. Algol 68 permits the declaration to be the left hand side of an assignment statement, and permits the intermingling of statements and declarations, as long as each variable is declared before it is used. The *initial* operations in Euclid and Alphard can set up the initial value at the time the object is created. Both languages require the programmer to either provide an initial value, or somehow prove that none is required.

- Virgin scopes. Dijkstra [9] has suggested that special syntactic support be provided so that the programmer may separate the declaration of a variable from its initialization, yet still have the compiler check that the variable is initialized before it is used. This separation is important because the initial value of a variable might not be known upon entry to the block in which it is declared. Inserting a dummy value would be distracting. Instead he proposes that the statement sequence comprising the range in which a variable is declared be partitioned into three subsequences: the initial sequence of statements in which the name does not appear, the statement in which the variable is initialized, and the sequence of statements in which the value of the variable may be used. If the initializing statement happens to be a compound statement, the variable must be imported into it as a *virgin* variable, and the statements comprising the inner range must be partitioned in the same manner as the top level. Dijkstra then

applies the restriction that initializing statements may not be repetitive statements, and thus guarantees that the initialization is not performed more than once. Furthermore, he requires that if the initializing statement is an alternative statement[1], that all alternatives be initializing statements. This guarantees that the variable is initialized exactly once. Algol 68's mechanism corresponds somewhat to Dijkstra's proposal. Because it permits intermingling of declarations and statements, it overcomes the problem of meaningless initial values. However, it provides no syntactic assistance for preventing multiple initialization, nor does it permit initialization inside alternative or compound statements, since the scope of the variable would then be limited to that statement.

Euclid's approach to initialization relies on the assumption that programs will be verified before they are run [48]. From this assumption one may conclude that the only time a variable may be read before it has been assigned to, is when the value *doesn't matter!* More precisely, the specifications for an operation may state that the variable must be in some particular state when the operation is applied. Thus, if the capsule did not provide an explicit initial value, it could simply provide some operations which did not examine the value of the variable, but did set it, and specify that one of them must be applied to the variable before applying any of the operations which do make use of the current value of the variable.

Alphard's approach to initialization is the same as Euclid's; thus both languages achieve by verification what Dijkstra would do syntactically.

4.4.2. Object-Object Relationships

Most of the data structuring facilities in modern languages, including data abstraction mechanisms, have been oriented toward simple composition of related objects. That is, one type is *composed of* objects of another type, and one module may be *composed of* other modules. Thus, the relationships among objects and among modules form trees. However, many programs require more general, graph-like relations among objects. Here we describe old and new mechanisms for such programming, and the perils therein.

4.4.2.1. The Problem of Pointers

The general pointer variable was a direct descendant from assembly language

[1] e.g. if-then-else

programming. When one data object needed to refer to another, it simply recorded the address of the other. LISP and PL/I adopted the notion unchanged. Programmers quickly found it both powerful and dangerous, because of the possibility of treating an uninitialized pointer variable as if it contained a legitimate address, and because of the possibility of undetected mistakes concerning the type of the object pointed to. The transition languages, Algol W, Pascal and Algol 68, all required that a pointer variable be declared to only point to one type of object. This solved the data misinterpretation problem, but not the initialization problem.

A second problem with pointers involves those objects which can be created and deleted independently from the block structure of program control. Because such objects provide the potential for graph-like structures which grow and shrink arbitrarily, they introduce the possibility that an object might be deleted while some variables are still pointing to it. This is called the dangling reference problem . If the space formerly occupied by the deleted object is now reused for some other object, one again has the potential for very obscure bugs. Most language systems now handle this problem by retaining a count of all pointers to an object (reference count), and not deleting the object until the reference count becomes zero.

However, the dangling reference problem recurs when some of the objects and pointer variables are allocated from a stack. Then one has the possibility of the stack discipline *forcing* the deletion of an object with outstanding references. Algol 68 comes very close to running afoul of this problem. In that language, all variable names are pointer (ref) constants. Thus any stack object may be referred to by ref variables of the appropriate type. To keep the problem from being unmanageable, Algol 68 requires that the extent of a ref variable must fall entirely within the extent of any object assigned to it. Since Algol 68's dynamically allocated (**heap**) objects are reference counted, it is safe for a stack variable to refer to a **heap** object, and illegal for a heap object to refer to a stack object. Furthermore, it is illegal for a stack variable to refer to a more recently allocated stack object.

Algol 68, however, confuses the language user with two rules involving stacks and scope. First, if an expression could evaluate to a reference to any of several stack objects, with different scopes, and at least one of those objects has a scope which would be legal for the context in which the expression occurs, the language permits the scope checking to be deferred until run-time, on the chance that the legal object might be selected. Second, Algol 68 permits the programmer to allocate objects from the stack without naming them, but

defines the scope of such objects to be the smallest enclosing statement which includes named stack-allocated objects. Thus the scope of the stack allocated objects depends on the presence or absence of possibly unrelated declarations.

Graph-like structures have an intrinsic problem with aliases. If two pointer variables in a particular range have the same type, it is in general impossible to prove that they don't refer to the same object. But more importantly, graph-like structures are useful precisely because they often *do* incorporate more than one way of referring to an object.

Euclid has tackled the first of these two problems directly, by introducing *collection variables* , which partition the space of objects of a given type. The type of a pointer in Euclid includes both the type of the object it will point to, and the collection from which the object will come. Thus two objects from different graphs will ordinarily also come from different collections, and pointers to those objects may be shown syntactically not to be aliases for one another. The second alias problem mentioned above, however, is intrinsic to the data structure being described, and is the source of the dilemma discussed in the next section.

4.4.2.2. VAR Parameters To Capsules

We have seen that general graphs provide little assistance in managing the complexity of a data structure. On the other hand, there are several more restricted classes of graphs which humans can understand well, such as lists and trees. Data capsules very naturally describe tree-like relations among objects, where the relation is "is composed of". For instance, a capsule might define a symbol table entry to be composed of a string, an address, and a value. Similarly (but not quite the same), a tree is composed of a left son, a right son, and a value, where the sons are references to trees. This second example is somewhat more tenuous, because one could envision operations which could cause the left son of a tree to be the tree itself. However, if a tree can only acquire a son by "growing" one, and can only lose a son by deleting it, such irregularities cannot occur.

Nonetheless, there are many cases where a programmer would like to construct graph structures containing cycles, without permitting the full generality and unmanageability of general graphs. The chief mechanism proposed for achieving this in modern languages is the VAR parameter to capsule definitions. An object passed as a VAR parameter to a capsule

variable instantiation is accessible within any operation applied to that variable, throughout the lifetime of the variable, in the same way that an initial value for a pointer field in a record creates a graph edge. Note that the parameter object might well be stack allocated, opening up opportunities for dangling references. It also provides an alias of sorts for the parameter variable, since any operation on the capsule variable may modify the original parameter variable. (The alias could become more explicit if the parameter name were also exported by the capsule!) Conversely, the object passed as a parameter to the declaration might also be a parameter to some operation on the capsule variable, creating an alias problem inside the capsule range. Because of the difficulties listed above, the designers of Alphard have still not settled on the right set of restrictions to place upon VAR parameters to capsules.

Euclid has a novel parameter mechanism which bears a superficial similarity to VAR capsule parameters, but serves a very different purpose, and thereby avoids some of the conceptual difficulties. Instead of permitting VAR parameters to a capsule, Euclid provides an *imports* clause, which lists a set of identifiers from the context of the capsule which are to be available inside every instance of the capsule. A procedure body in Euclid may also have an imports clause. The variables in an imports clause must be available at *both* the definition site and the invocation site of the range (procedure or capsule) to which the clause is attached. (A variable is considered available at an invocation site even if it is a concealed field of a variable which is actually visible in the invocation context.) Thus those identifiers are roughly equivalent to normal parameters, except that the actual parameter is specified at the definition site instead of the invocation site. The motivation for this construct is that Euclid's capsules and procedures are both closed ranges, and may not contain any free variables. The imports clause provides most of the same functionality as inherited names, but with two important differences: the inherited names are specified explicitly in the range header, and the names are bound both statically and dynamically to the same variables. The static-dynamic rule for imports gives imported objects full status as candidates for side-effects. That is, it guarantees that any object available within a capsule operation can be treated as if it were a parameter to that operation. This is in contrast to a VAR capsule parameter, which might not be available at the site of every operation invocation on the capsule variable, and thus not considered when noting side-effects.

Despite the potential complexity of VAR parameters to capsules, the following two examples show their importance.

4.4.2.3. Binary Trees

Shaw *et al* [50] have written and verified a capsule which defines a binary tree. Their tree definition actually defines two data abstractions: a tree and a node. Every node belongs to at most one tree; each tree may contain many nodes. One means of modifying a tree is by "growing" a son for one of its nodes. Such a growing operation affects both the original node and the tree to which it belongs. The most natural way to express the relationships involved is to permit a tree to refer to its nodes, and also to permit a node to refer to its tree. Otherwise the "grow" operation is hard to define. Consider:

- Grow(tree, node): does the node really belong to the tree?
- Grow(node): unless the node refers to the tree, how can this operation update the node count for the tree?
- Grow(tree, path): what if the path from the root of the tree to the desired node isn't known?

Shaw et al use a VAR parameter to the node capsule to let the node refer to its parent tree. The Alphard group is contemplating restricting VAR parameters to capsules to be of the type of the smallest containing capsule. Euclid's *imports* clause would permit this kind of relationship. The node capsule would be defined inside the tree capsule, and would import the name of the tree, or of the appropriate components of the tree. Then any operation to create a node would have to select the "node capsule" field from a particular tree, and that particular tree would be imported into the node being created.

4.4.2.4. Resource Problem

Resource consumption is an aspect of program behavior which until fairly recently has not been treated with the tools of program verification. In many high-level language systems it is of no concern, because the language system conceals the finiteness of resources from the user. However, in programs which implement operating systems, resource consumption is a vital concern. Nonetheless, it is usually separable from other correctness concerns, and often should be treated separately, although with the same tools.

Consider a symbol table in a language translator. The capsules which implement the types symbol table and symbol table entry will ordinarily be considered correct if they faithfully represent the information stored in them. But what if the symbol table overflows? Is the

program still correct? In many contexts it would be, because the user would simply reconfigure the translator with a larger symbol table, and try again. But if the symbol table were storing airplanes in an air traffic control system, symbol table overflow (i.e. too many airplanes) would be a fatal error. So consumption of symbol table resources must be considered in the verification of such a system.

VAR parameters to capsules have been proposed as a vehicle for propagating access to resources. We have already seen the side-effect problem inherent in such relationships. It becomes particularly critical here, since capsules which otherwise have nothing to do with each other might draw resources from the same pool, when neither is aware that he is consuming resources at all, because the consumption is hidden in the capsules it uses.

The Resource Problem is a topic of ongoing research.

4.4.3. Generic Types

VAL parameters to variable declarations provide information for two forms of initialization: initial values, and structure selection. In early languages, array declarations included "parameters" which indicated size and index bounds of the array. In transition and modern languages, a VAL declaration parameter might also select one of a finite set of alternative structures for objects of the specified type. Such alternative structures, usually called variant records or variant types, are a powerful means of grouping related types. For instance, a factory inventory program might like to use the same procedures for processing all requisition forms, except for small pieces of the program which specialized in office requisitions or maintenance supplies. The programmer could declare a type "requisition" to be a record with a set of fields for requisitioner, account number, date, etc., and then a different set of fields for each category of requisitions. The declaration of a variable would then supply a parameter to indicate whether it would handle all kinds of requisitions, or only some particular kind.

A *generic type* is a data capsule in which some of the component types of the representation are provided by the user of the capsule. Similarly, a generic procedure is one for which the types of some of the parameters are likewise provided by the caller of the procedure. Thus a generic type defines a whole set of capsules, one for each possible set of user-provided types, and a generic procedure defines a set of actual procedures, one for

each possible set of parameter types.

That ubiquitous example, the stack, is also suitable for illustrating generics. A stack capsule might well be defined independently from the type of object being stacked. Such a definition might look something like the following:

```
capsule stack(T:type) operations push,pop,top, empty =
   begin
   proc push(s:stack(T),item:T)=
      ...

   proc top(s:stack(T)) returns item:T =
      ...

end stack
```

The stack defined above can stack any sort of object, provided that (a) all objects are of the same type as specified at stack declaration time via the parameter T, and (b) the type of the actual parameter provided for T must have an assignment operation defined for it. The procedures for the stack might or might not be considered generic procedures. At the site of the procedure definition, the type T is a bona fide type. However, since T is defined parametrically at the capsule head, the procedures defined will have many different versions, depending on what parameters are provided for various stack variable declarations. Here is a simpler version of a generic procedure:

```
proc equal(T:type, a,b:[1..10] array of T) =
      begin
      for i = 1 .. 10 do if a[i] notequal b[i] then return false;
      return true;
      end
```

This procedure can test for the equality of the values of any two arrays with indices between 1 and 10, provided that both arrays contain the same type of value, and that type has a "notequal" operator defined on it.

Generics are a logical generalization of the abstraction method introduced by capsules. They represent the notion that a particular body of code may be written based only on the specifications of the data types used in it, without reference to the implementation, or even the true identity, of those types. Generic types are particularly useful for describing types whose principal purpose is organization. In the stack example, there is no reason why the code implementing stacks should have access to the representation of the objects being stacked. Conversely, operations on an element of a set should not necessarily have access to

the link or tag fields which connect it with other elements of a set.

Generic types and procedures take over one of the functions previously provided by PROC parameters. Procedures which otherwise might be passed explicitly as separate parameters may sometimes be passed implicitly as one of the operations defined on the type of some parameter. In particular, the chief complaint about procedures as parameters was the cost and confusion involved in free name binding; with generic types the data involved is passed explicitly, with procedure attached. There is no opportunity for side-effects other than on the actual parameters, or via whatever other side-effect mechanisms are present in the parameter type. (The free name argument against procedures as arguments applies equally well to returning a procedure as the value of another procedure. The chief complaint against them is the complexity of free name binding; in such cases an abstract data object as the procedure value, with the appropriate operation defined on it, makes the data passing explicit and well controlled.)

Simula 67 had a simple generic type facility which was a generalization of the notion of variant records. Its *subclass* mechanism made it possible to extend a class with another class, producing an object which was eligible for operations defined on either class, with operations on the base class ignorant of the existence of the extension. The base class could be extended by different class in the same program; each extending class is called a *subclass* of the base class (class). When Simula 67 added facilities for concealing representations, it included facilities for permitting a base class to conceal parts of its representation from any subclasses defined on it. Note that the base class (corresponding to a generic type) need not make any assumptions about properties of the subclasses (parameter types).

CLU has always included types as parameters to capsules, requiring only that the parameter type have operations with specified names and parameter types. Euclid omitted generic types primarily due to skepticism about the cost of implementation. Alphard's work in generic types is one of its major contributions to language design; it provides very convenient mechanisms for specifying a wide variety of properties of a type passed as a parameter, without tying down the implementation of those properties. For an example, see [33].

4.4.4. Closely Related Types

We have already seen in previous sections two kinds of close relationships between types. We have seen that the relation "is composed of" is central to the methodology of data abstraction. We saw in the binary tree example that a type might want to export one of the types of which it is composed, because that type provided a different view of the same object. From that notion we may generalize to the possibility of defining two types which are intimately related by common design decisions, shared data, or mixed-mode operators (e.g. "compute the area covered by this square and this circle, even if they overlap"). To handle such situations one would like to be able to access the representation of two types simultaneously. Such access is quite convenient in Euclid and Modula, since a module defining more than one type provides access to the representation of each type to all procedures in the module. Alphard provides even finer control over such overlapping, by permitting the specifications exported with a form to be more abstract than the specifications used inside the parent form.

4.5. Applying Data Abstraction To Several Scope-related Problems

The notion of data abstraction has revolutionized the entire field of language design. It has produced new insights into a variety of problem domains. New implications continue to emerge. The following problem areas have received significant benefits.

4.5.1. Loops as ranges

One of the more famous shortcomings of Algol 60 was its iteration statement definition. The rewrite rule used to define it implied that the quantities used to compute the steps of an iteration would be computed as many as three times for each iteration. Knuth [23] has described the debate over what was really intended, in great detail.

Algol 60's problems arose from the fact that the iteration variable, step-control expressions, and loop body were all considered to be in the same range as the surrounding statements. Thus it was perfectly permissible, if not altogether reasonable, to include statements in the procedure body which would change the step size of the iteration, or even change the value of the control variable. Languages like Pascal, Algol 68, and Bliss have taken some variant of the position that the iteration variable is a NEW variable, implicitly

imported as a VAL variable into the loop body, and changed only by the stepping code. Similarly, they view the control expressions as values computed at loop entry and constant thereafter. (Of course, many compilers do not enforce these rules.)

The above constraints make it trivial to prove that a for-loop statement terminates, independent of what the loop body does. However, they also restricts the kinds of iterations which the counted loop may describe. Those which have been excluded must be described by the while-loop.

Complex data structures often require correspondingly complex iteration sequences over their elements. One common operation on trees, for instance, is printing them in order. Searching for an element with a particular property is another common operation on large data structures. For this reason, languages which permit definition of large structures also define iteration methods for them. Arrays, for instance, may be easily traversed in subscript order. Euclid has a special variant of its iteration construct for iterating over the elements of a set. Alphard, Euclid, and CLU are all developing mechanisms by which the author of an abstract data type may specify, and conceal, a set of procedures which will generate the elements of the type one by one [51, 31]. All of the constructs define a closed range which takes an object of the type as a parameter, creates a concealed object to maintain the state of the iteration, exports a variable containing the current element of the generated sequence, and provides a means to "pulse" the state to produce the next item in the sequence.

The unsolved problem in this line of research is the question of how to describe the ways in which the object which is the parameter to the iteration module may be modified, both within the module and in the loop body. Notice that this is the same problem that Algol and Fortran had with their step-control expressions. One early solution proposed for Alphard was to prohibit the loop body from modifying the parameter objects. This was finally rejected because it excluded the common operation of examining the elements of a set and removing some of them. Another proposal provides syntactic means for specifying precisely which operations on the parameter objects are permissible within the loop body. None of the solutions proposed so far makes it possible in general to specify an iteration module which will terminate regardless of what the loop body does.

4.5.2. Aliases Revisited

Concern over aliases and side-effects has been a recurrent theme throughout this paper. It has been a primary criterion for judging binding mechanisms, for designing pointer mechanisms, and for analysing relationships among objects, types, and ranges. Indeed, one language in particular, Euclid, has set the removal of aliases as one of its most important goals [48]. It has done extremely well.

Euclid's rule regarding aliases is the following:

> "The language guarantees that two identifiers in the same scope can never refer to the same or overlapping variables."[26]

To do this, Euclid introduced a number of innovations. First, any range which can be entered other than via the textually preceding statement is a CLOSED range. This eliminates the possibility of aliases or side-effects through free names. In the place of this, Euclid provides the *imports* clause, which permits a range to name and use identifiers declared outside it, provided they are available in both the static and dynamic contexts of the range. Thus every object used in such a range must be available in the dynamically enclosing range, or created locally. If a given range contains no aliases, and all of the parameters and imports to each range it invokes are distinct, then it has not introduced any aliases into the ranges it calls. By induction, all Euclid programs are alias-free. (The variables listed in the imports clause are considered to be parameters, and thus eligible for modification.)

The other major cause of aliases is pointers. We have already mentioned that Euclid's collection variables were designed to alleviate pointer alias problems. This requires further explanation. A collection variable is considered to be an unbounded vector of objects of the type for which it is a collection. Then a pointer variable is considered to be an index into the collection vector. If two pointer variables point into the same collection, one cannot determine statically that they do not contain the same index. However, this is no worse than proving that two indices into an array are not equal, and Euclid relegates that task to the verifier, or inserts runtime checks if so instructed.

The two innovations above indeed make alias free programs achievable. One might wonder what flexibility Euclid sacrificed to do this. Reviewing all of the comparisons given so far in this paper, the only major expressive techniques unavailable in Euclid are generic types, VAR parameters to types, and simultaneous access to the representations of related capsules. Of these three, we have shown that the first can be partially simulated by variant records. The second mechanism can be partially simulated by *imports*, and those uses which cannot be

simulated seem perilous. The third technique cannot be imitated in Euclid, but the issues involved don't seem to include aliases. Ultimately, only experience and further research will tell whether Euclid has sacrificed too much to avoid aliases.

The designers of Alphard, while skeptical of aliases, are less militant than Euclid. They permit aliasing when it is carefully documented.

4.5.3. Exception Handling

One of the chief complaints voiced about exception handling mechanisms in standard programming languages is that they either don't permit the handling routine to access the objects it needs, or don't preserve the integrity of the data structures which were being modified when the exception occurred.

Levin [27] has used the concepts of objects, ranges, and capsules to clarify the issues, survey existing facilities, and present a new mechanism. He describes exception handling in terms of the signalling environment, the entity to which the exception applies, and the environments which may process the exception. A condition may be associated either with an instance of a control construct, as when a procedure call receives unusual parameters, or with an instance of an object, such as when a file is found to contain parity errors. A handler is always associated with a "user" of the instance on which the condition is defined, whether that be the caller of a function, or a range in which an object is accessible. The signaller of a condition is the program segment which detects the condition.

An exception handling mechanism, then, may be characterized by the ways in which handlers may be associated with instances, and by the ways in which control and data may flow between the range signalling a condition on an instance, and the ranges with handlers attached to that instance. Prior to data abstraction, most mechanisms associated conditions only with control instances. Handlers for conditions were generally either statically defined, or provided by the callers of the procedure raising the condition. Little provision was made for passing data between the signaller and the handling range. Algol 68 introduced, with its file exception mechanism, the idea of associating conditions and handlers with objects. In data abstraction languages, a handler could be associated with a variable at its declaration site, or for the duration of a particular control construct, or could even be defined in a type definition to hold for all instances of that type. Refer to Levin's thesis for more detail.

4.5.4. Type Breaching

Most systems programmers sooner or later find themselves faced with a programming problem for which the most direct solution is to treat a single object in the task domain as having two different types. In most cases it is impossible to prove anything about objects treated that way, because the interpretation must include information about the exact bit-level representation of both types; this information is often not available. Consequently modern programming theory frowns upon such practices. However, modern theory has not provided an adequate set of alternative techniques, so for the time being most languages, at least those intended for systems programming, provide some mechanism for it. Pascal didn't intend to provide such a mechanism, but programmers quickly discovered that the variant record construct permitted it by not forcing a variant field to be treated as having the type implied by the tag field. Indeed, several Pascal compilers, written in Pascal, make heavy use of this feature. Euclid attempts to provide a carefully controlled feature of this nature, namely an explicit conversion operator. To support that operator, Euclid insists that it occur in a "machine dependent" module, and that it only map between types whose standard representations are defined in the language. Euclid provides one other mechanism, for conversion in cases where one of the types has no meaningful values, like machine words. In all other cases, uninterpreted type conversion is illegal.

4.5.5. Scope Aspects of Multiprogramming

Modern scope mechanisms also make programming of cooperating processes a little easier. The underlying scope problem in this area is very much like the Alias Problem: a program cannot be verified if the values of variables it relies upon may change unexpectedly.

Hoare[20] has produced a language construct, called a monitor, for controlling data sharing among processes. In its simplest form, it is a general module, accessible by any process, but with the restriction that only one process may be executing in it at a time. If a process attempts to enter a monitor while another is executing in it, the entering process is suspended until the other is done with the monitor. Modula interface modules and device modules are extensions of that construct, providing certain ways that a process may suspend itself in the middle of the module, permitting other processes to execute in the module while it's suspended.

The monitor is useful for verification because when verifying the code of a module, one can assume that no other process will change the variables declared in the module, except possibly during *wait* and *signal* operations, thus making verification nearly as simple as with serial programs. Unfortunately, the monitor is now being pushed beyond the limits of its usefulness [47]. Many are now trying to use it to implement elaborate synchronization protocols, not just simple mutual exclusion.

Owicki [38] has taken the technique a step farther and shown that pre- and post-conditions of the abstract specifications for monitor procedures must be phrased in terms only of variables private to the calling process, and not in terms of the shared variables. Only the invariant properties of the module specification may mention its parallel nature. The reason for this is that any non-private, non-invariant property occurring in the post-condition of a procedure might immediately be made false by another process.

5. Programming Examples

In this chapter we shall sample the scope philosophies of several of the languages used in this study, by studying how a particular programming exercise would be written in each of them. The languages we have studied cover an extremely broad range of expressive power and intended usage, so it would be inappropriate to try to solve exactly the same problem in each language. Instead, for each language I shall state and solve a slightly different version of the problem, designed to show the strengths and weaknesses of that language.

5.1. The Problem

Each of the programming examples in this chapter shall implement a a queue. For our purposes we define a queue to be a sequence of objects with the following restrictions:

- Objects may only be added to the sequence by appending them to the left-hand end. For this purpose each program will include the operation *insert*.
- Objects may only be removed from the sequence by deleting them from the left-hand end, via the operation *remove*, which also returns the value of the object removed.
- The sequence is initially empty
- The length of the sequence may never exceed a specified maximum. Inserting an object in a queue of maximum length is not permitted. (Similarly, removing an object from an empty queue is not allowed).
- The current length of the queue must be available to the user.

5.2. Fortran

Fortran was invented before scope was considered an issue; nonetheless it is possible to write a collection of programs to implement a queue in a reasonably straightforward way. The principal issue here is the sharing of the representation of the queue among several subprograms, without forcing the user to be aware of too much detail. If the items to be queued are integers then a solution could encode the "front" and "back" pointers into the array holding the data, and pass it as a parameter. However, I prefer a version which would apply to real numbers as well. Therefore the queue is stored in a named COMMON area. Examples for other languages will build queues of characters; in Fortran characters may be

conveniently represented as integers.

```
      Subroutine Qinit
      Integer Q(100)
      Integer front, back
      Common /Queue/ front, back, Q
      front = 1
      back = 1
      return
      end


      Integer Function Qsize
      Integer Q(100)
      Integer front, back
      Common /Queue/ front, back, Q
      Qsize = Mod ((back-front),100)
      return
      end


      Subroutine Qinsrt ( I )
      Integer Q(100)
      Integer front, back
      Common /Queue/ front, back, Q
      If Qsize .eq. 99 STOP
      Q ( back ) = I
      back = Mod ( back, 100 ) + 1
      return
      end


      Integer Function Qremov
      Integer Q(100)
      Integer front, back
      Common /Queue/ front, back, Q
      If Qsize .eq. 0 STOP
      Qremov = Q ( front )
      front = Mod ( front, 100 ) + 1
      return
      end


      Integer Function Qfirst
      Integer Q(100)
      Integer front, back
      Common /Queue/ front, back, Q
      If Qsize .eq. 0 STOP
      Qfirst = Q ( front )
      return
      end
```

The principal shortcomings of this technique are the weaknesses of the named COMMON construct: discrepancies in the variable lists for a given area between different subprograms are not checked, and any subprogram declaring a common area with the same name, has access to the data.

5.3. Algol 60

Algol 60 permits one to implement a queue whose size is a parameter (of sorts), by the trick of declaring and initializing a variable holding the queue size, in an outer block. The user of the stack need not be aware of the representation of it, except that he must avoid using the names of variables used to implement the queue. (This may be enforced by the clever use of blocks, too.)

```
begin
integer qlimit;
qlimit = 100;
   begin
   string array q[1:qlimit];
   integer front,back;

   integer procedure qsize;
      qsize := (back - front) rem qlimit;

   procedure modincr ( i );
      integer i;
      i := (i rem qlimit) + 1;

   procedure qinsert ( s );
      string s; value s;
      if (qsize = qlimit - 1) then fail
      else begin
         q[back] := s;
         modincr ( back )
      end;

   string procedure qremove;
      if qsize = 0 then fail
      else begin
         qremove := q[front];
         modincr ( front )
      end;
```

```
    string procedure qfirst;
        if qsize = 0 then fail
        else qfirst := q[first];

    Comment queue initialization;
    first := 1;
    last := 1;

     ...

    end
end
```

Rather than define a separate procedure to initialize the queue, I have written out its initialization as the first executable statements of the block. This is reasonable when the scope of the queue and the scope of its implementation are the same. In the next section we will examine a different technique.

Observe that Algol 60 has no provision for constructing complex objects which are not arrays, so that the representation of a queue must span several variables.

5.4. Pascal

One of the chief contributions of Pascal was its generalization of data structuring mechanisms, within the framework of a language committed to minimal runtime overhead. In the following program we define the *type* queue, enabling the user to declare as many queues as he needs.

```
qlimit = 100;
qmax = 99;
queue = record
    front: 1.. qlimit;
    back:  1.. qlimit;
    data: array [1..qlimit] of char
    end

procedure qinit ( var q:queue);
    begin
    q.front := 1;
    q.back := 1
    end;
```

```
function qsize ( q:queue ):integer;
   qsize := (q.back - q.front) mod qlimit;

procedure modincr ( var i:integer );
   i := (i mod qlimit) + 1;

procedure qinsert (var q:queue, c:char);
   if qsize ( q ) = qmax then fail
   else begin
      q.data[q.back] := c;
      modincr[q.back]
   end;

function qremove (var q:queue):char;
   if qsize ( q ) = 0 then fail
   else begin
      qremove := q.data[q.front];
      modincr ( q.front)
   end;

function qfirst ( q:queue ):char;
   qfirst := q.data[q.front];
```

This implementation once again includes an initialization routine, but this time it is because there may be many queues declared in different places. Observe that this implementation does not use *any* global variables, and only two global constants, qlimit and qmax. Thus the user of queues may be sure that his operations are not affecting any variables except the queues on which they operate.

Pascal is notorious [14] for its decision to include array bounds as part of the type of a variable, leading to the requirement that array bounds be compile time constants. If a program had to have queues of two different sizes, it would have to have two complete sets of definitions of the type queue and its routines, with the only difference between the two being the value of qlimit.

5.5. Algol 68

Algol 68 does not include the dimensions of arrays in the type of a structured object. Therefore, in the following program the size of the queue is stored as a field of its

representation. The initialization routine is written so that it may be used as part of the statement in which the queue is declared, i.e. so that the declaration may be followed by a *collateral assignment* to its fields.

```
begin
int qlimit := 100;
mode queue = struct
       (int front,back,limit,mx,
       [qlimit] char data);
proc qinit = queue:(1,1,qlimit,qlimit-1, skip);

proc qsize = (queue q)int:
       ((back of q - front of q) mod limit of q);

comodincr would need too many parameters to be worthwhile co

proc qinsert = (ref queue q, char c)void: (
       if qsize ( q ) = mx of q then fail
       else begin
           (data of q)[back of q] := c;
           back of queue := (back of q mod limit of q) + 1
       end);

proc qremove = (ref queue q)char:(
       if qsize ( q ) = 0 then fail
       else begin
           char c := (data of q)[front of q];
           front of q := (front of q mod limit of q) + 1
       end);

proc qfirst = (queue q)char:(data of q)[front of q];

 ...

end
```

Unfortunately, both the array dimensions and the initial values for the limit and mx fields had to be computed from global variables, since Algol 68 has no means for parameterizing a type definition. Furthermore, neither Algol 68 nor Pascal provide a means to restrict the scope of the representation of a type, so that any part of the program text which has access to the name of a queue variable may also modify its fields individually.

5.6. Euclid

The following Euclid module both implements and protects the *type* queue. Furthermore, the length of the queue is a parameter to the type definition, so that each queue declaration may specify concisely what its size will be. Even better, all of the initialization is taken care of by the module at declaration time, based on the same parameter, so that the declarer need not be concerned about initial values.

```
type queue( pervasive limit:integer ) = module
   exports ( insert, remove, size, front )
   var front,back:1..limit
   pervasive mx = limit - 1
   var data = array 1..limit of char

inline function size returns s:integer =
   imports (front, back )
   s := (back - front) mod limit

inline procedure modincr (var i:integer) =
   i := ( i mod limit ) + 1

procedure insert ( c:char ) =
   imports ( var data,  var back, front, size )
   pre size < mx
   post ...
   begin
      data[back] := c
      modincr ( back )
   end

procedure remove returns c:char =
   imports ( var front, data, back, size )
   pre size > 0
   post ...
   begin
      c := data[front]
      modincr ( front )
   end
```

```
inline function first returns c:char =
   imports (front, back, data, size)
   c := data[front]

initially
   begin
      front := 1
      back := 1
   end
end
```

The imports clauses in this example seem rather long. This is primarily due to the Euclid rule that imports must come from both the static and dynamic contexts of a range. Notice how the pervasive designation permitted the use of constants without importing them.

Both Pascal and Algol 68 specified the fields of a queue by using the field names as *selectors* on the queue variable. Euclid's syntax is such that the module variable to be used in a module operation is a *prefix parameter* of the call. Then any field name imported into the operation implicitly refers to that field of the prefix parameter, rather than having to explicitly attach it to a module variable name. However, this makes it impossible to refer to fields of two module variables in the same procedure, because the field variable would be ambiguous.

5.7. Alphard

In this example, even the type of object being queued is a parameter to the form definition. Alphard uses the name qualification syntax of Pascal and Algol 68 to refer to the fields of form instances, making possible the function "transfer" which moves a specified number of objects from the head of one queue to the back of another.

Due to the index computations in the transfer function, the data array in this program is based at zero rather than one.

```
Form QUEUE(T:form <:=>, limit:INT) =
pre ( limit > 0 )
begin
spec
   func size (q:QUEUE):INT
   proc insert (q:QUEUE, x:T)
   vproc remove (q:QUEUE):T
   func first (q:QUEUE):T
   proc transfer (q,r:QUEUE, n:INT)

impl
   var data:VECTOR(T,0,limit-1), front, back:INT = 0
   invariant ( 0 ≤ front ≤ limit-1 ∧ 0 ≤ back ≤ limit-1 )

   func size is (q.back - q.front) mod q.limit

   proc insert is if size (q) ≥ q.limit - 2 then fail else
          q.data[q.back] := x;
          q.back := (q.back + 1) mod q.limit fi

   vproc remove = if size (q) = 0 then fail else
          var x:INT := q.data[q.front]
          q.front := (q.front + 1) mod q.limit
          x  fi

   func first is if size (q) = 0 then fail else
          q.data[q.front] fi

   proc transfer is
      if (size(q) < n) or (size(r) > (r.limit-n-1) then fail else
          for i:upto(1,n) do
             r.data[(r.back + i) mod r.limit] :=
                q.data[(q.front + i) mod q.limit] od
          r.back := (r.back + n) mod r.limit;
          q.front := (q.front + n) mod q.limit fi

end QUEUE
```

5.8. Modula

Queues are the basis for a set of synchronization problems called producer/consumer problems. The simplest such problem is composed of two processes. One process is producing objects and placing them in a queue; the other is removing them from the queue and consuming them. The problem is to synchronize the insertions and deletions so that simultaneous activity doesn't destroy the consistency of the data, and so that instead of halting the program when the queue overflows or runs out, the code for the queue will suspend the process attempting an insertion or deletion until the size of the queue is

appropriate for the operation.

The following *interface module* solves the simple producer/consumer problem. An interface module may only be entered by one process at a time, except for processes suspended in the middle by executing wait or signal statements. Thus the consistency of the queue data may be assured by simply making sure that it is correct at procedure entry, procedure exit, and at each wait and signal statement.

```
interface module queue;
   define qsize, qinsert, qremove, qfirst;
   use [qlimit];
   var front, back:integer;
   var full, empty:signal;
   var data:array 1:qlimit of char;


   procedure modincr ( i:integer );
      use [qlimit];
      begin
      i := (i mod qlimit) + 1;
      end;


   procedure qinsert (c:char ) ;
      use [ data, back, size, limit, modincr ];
      begin
      if qsize = qlimit-1
      then wait (full)
      else begin
         data[back] := c;
         modincr ( back );
         if size = 1 then signal (empty)
         end;
      end;


   procedure qremove:char;
      use [ data, front, size, modincr, qlimit]
      begin
      if qsize = 0
      then wait ( empty )
      else begin
         qremove := data[front];
         modincr (front);
         if qsize = qlimit - 2 then signal (full);
         end
      end;
```

```
procedure qfirst:char;
        use [ data, front, size ];
        if size = 0 then fail
        else qfirst := data[front]
        end;

end queue;
```

Unfortunately, interface modules do not adequately support queues of more complex objects. If the objects being queued were buffers of, say, 256 characters each, it would be important for efficiency reasons not to copy the buffers into and out of the queue. Furthermore, one would like to permit the producer and the consumer to fill and empty their respective buffers simultaneously. Therefore each buffer would have to reside in a different interface module, or in no interface module at all. The former solution is ridiculous; the latter gives up the very protection the interface module was supposed to provide. In either case, due to the fact that Modula has no pointers, the interface module could only be used to protect the computation of a buffer index, and not to protect the buffer itself. Even so there would be no protection against either the producer or consumer using buffers for which it had not received indices.

5.9. Simula 67

I have saved Simula 67 for last because, though in many respects it is not an elegant language, it permits a degree of flexibility in queues not available in other languages. By careful use of the *subclass* facility it is possible, indeed reasonable, to implement queues which contain objects of any type, indeed of types not known at the declaration site of the queue.

In the following program, the Queue class knows nothing about the buffer class. The buffer class is actually a dummy class; it takes no parameters, and has no attributes. However, users of the Queue class can construct subclasses of the buffer class, without changing either the buffer or queue classes. Since a queue contains references to buffers, and those buffers may be any subclass of the buffer class, a queue may contain any arbitrary mixture of elements. Indeed, the INSPECT statement is sufficiently flexible that a program removing elements from the queue can pick out exactly those elements it knows how to process, and skip those whose class it does not recognize.

```
class buffer;
begin
end;

class queue (limit); integer limit;
hidden protected limit, front, back, data, modincr;
begin
integer front, back;
ref (buffer) array data [1:limit];

procedure modincr ( i );
   integer i;
   i := (i mod limit) + 1;

procedure insert (b);
   ref (buffer) b;
   if size = limit - 1 then fail
   else begin
      data [ back ] := b;
      modincr ( back )
      end;

ref (buffer) procedure remove;
   if size = 0 then fail
   else begin
      remove := data [ front ];
      modincr (front)
      end;

ref (buffer) procedure qfirst;
   if size = 0 then fail
      else qfirst := data [ front ];

front := 1;
back := 1
end *** queue *** ;

buffer class a ( b );
begin
 ...
 ...
 ...
end
```

```
buffer class c ( d );
begin
 ...
 ...
 ...
end


ref (queue) q;
ref (buffer) x;
 ...
 ...
x := remove ( q );
inspect x when a do ...
          when c do ...
 ...
```

6. Summary

We have seen that scope is a strong element of program structure. The set of range definition facilities provided by a language determine the class of permissible program structures. Languages which have a sufficiently rich set of range facilities have been able to simplify their parameter and binding mechanisms. Many of the problems formerly associated with the extent of variables are also simplified in a richer range environment. Modern understanding and implementation of data abstraction concepts has contributed to the solution of a number of important program and system structuring problems.

The first major innovation in scope structure was Algol 60's nested block concept. It provided a tool for building programs in a hierarchical scope structure, where both the control structure of the algorithm and the scope and extent structure of the data had to fit into the same hierarchy. The resulting structure turned out to be better suited for expressing algorithmic structure than for expressing the structure of data, as evidenced by the Alias and Hole-in-Scope problems.

Data encapsulation mechanisms, the single most important development in modern language design, provide the means to express the structure of data in a way which distinguishes it from the structure of algorithms. A data abstraction is often a natural tool for describing a design decision, or group of decisions, making it possible to concentrate all of the parts of a program which are affected by such a decision into a concise package. This improves the modularity of programs, thus enhancing understandability and verifiability.

Languages supporting data abstraction have been able to simplify their parameter mechanisms and reduce their overhead due to free variables. The NAME, PROC, and LABEL mechanisms are disappearing from modern languages because they embody both data and control information, which can now be better expressed with range definition mechanisms. Many of the control abstractions necessitating EXPR or PROC parameters can be implemented as well or better using data abstraction facilities. Data abstraction languages also provide a firmer foundation for exception handling mechanisms, reducing the need for PROC and LABEL parameters in this capacity.

Free variable binding, highly desirable in an Algol-like scope environment, has receded in importance in modern languages because of the distinction between the structure of algorithms and of data. Such binding is no longer necessary to provide common data to

multiple procedures, nor is there a tendency to need a large number of global variables in any one environment. Furthermore, free name binding was a major source of alias and side-effect problems. Consequently, most data abstraction languages do not permit free variable binding across procedure or capsule boundaries. Note, however, that constants, field selectors, and procedures are often inherited in the conventional Algol way.

The OWN concept, considered innovative and promising when introduced in Algol 60, has been displaced somewhat by data abstraction. The OWN concept separated the extent of a variable from the duration of the control construct delimiting its scope; data abstraction has done that and more by separating the scope of the name of a variable from the scope of its object.

Data abstraction, we have seen, has shed light on a number of long-standing systems programming problems. It has provided the basis for a sound exception handling proposal which generalizes to multiprogramming. It has shown the way to reasonably controlled type-breaching. It has suggested useful methodologies for parallel programming. Most importantly, however, it has provided the means for decomposing a system into coherent modules while retaining meaningful structure and disciplined scope relations between modules.

By studying implementations of queues in various languages, we find that indeed the modern languages provide a great deal more structure to programs. We find the data structuring tools of Pascal and Algol 68 to fit naturally into good programming style. We find the modules and of Euclid and Modula to be convenient encapsulation tools. We find Alphard forms to be excellent tools for constructing data abstractions. Surprisingly, though, we find that Simula 67, which was the first language to permit grouping of procedures around a data type, still excels in the flexibility of its generic types, ten years after its creation.

References

[1] F.T. Baker, Structured Programming in a Production Programming Environment. Proceedings of the International Conference on Reliable Software, SIGPLAN Notices 10,6 (1975).

Presents management tools for implementing the Chief Programmer Team concept, including the notion of structured coding as distinct from structured programming.

[2] Frederick P. Brooks, The Mythical Man-Month, Addison Wesley, 1975.

An entertaining set of essays on the management of large software projects.

[3] Ole-Johan Dahl et al, Simula 67 Common Base Language, Norwegian Computing Center, Oslo.

Reference manual.

[4] Ole-Johan Dahl and C.A.R. Hoare, Hierarchical Data Structures. In Structured Programming, Dahl, Dijkstra and Hoare, Academic Press, London, 1972.

Presents the class and subclass mechanisms of Simula 67.

[5] Edsger W. Dijkstra, GOTO Statement Considered Harmful. **Communications of the ACM** (March 1968).

An early example of the influence of language on the quality of software.

[6] Edsger W. Dijkstra, Notes on Structured Programming. In Structured Programming, Dahl, Dijkstra and Hoare, Academic Press, London, 1972.

A landmark work on structured programming.

[7] Edsger W. Dijkstra, 1972 ACM Turing Award Lecture: The Humble Programmer. **Communications of the ACM** (October 1972).

Describes the human limitations which make structured programming imperative.

[8] Edsger W. Dijkstra, Correctness Concerns and, Among Other Things, Why They Are Resented. Proceedings of the International Conference on Reliable Software, **SIGPLAN Notices** 10,6 (June 1975), 546-550.

Motivates program verification.

[9] Edsger W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

Presents a programming and verification methodology which places heavy emphasis on proving termination and controlling the scope of names. Also introduces a novel approach to initialization.

[10] Mark Elson, Concepts of Programming Languages, SRA, 1973, 67-84.

Uses an elaborate formal basis for classifying binding mechanisms in programming languages.

[11] A.C. Fleck, On The Impossibility of Content Exchange Through The By-Name Parameter Transmission Mechanism. **SIGPLAN Notices** (1976), November.

Illustrates the problem of repeated evaluation in the NAME parameter mechanism.

[12] Lawrence Flon, On the Design and Verification of Operating Systems, Computer Science Department, Carnegie-Mellon University.

[13] Jack Goldberg (ed.), Proceedings of a Symposium on the High Cost of Software, SRI.

An example of the literature of the "software crisis", including analysis of the components of the cost of software.

[14] A. Nico Habermann, Critical Comments on the Programming Language Pascal, Computer Science Department, Carnegie-Mellon University.

Criticizes the concept of type underlying Pascal.

[15] Paul N. Hilfinger et al, An Informal Definition of Alphard, Computer Science Department, Carnegie-Mellon University (in preparation).

Reference manual.

[16] C.A.R. Hoare, Notes on Data Structuring. In Structured Programming, Dahl, Dijkstra and Hoare, Academic Press, London, 1972.

[17] C.A.R. Hoare, Proof of Correctness of Data Representations. **Acta Informatica** 1 (1972).

Presents the verification methodology eventually adopted by Alphard, Clu, and Euclid.

[18] C.A.R. Hoare, Data Reliability. Proceedings of the International Conference on Reliable Software, **SIGPLAN Notices** 10,6 (1975), 528-533.

Presents the mathematical notion of type.

[19] C.A.R. Hoare and Niklaus Wirth, An Axiomatic Definition of the Programming Language Pascal. **Acta Informatica**2,4 (April 1973).

The axioms for variant records are internally inconsistent, precisely where variant records in Pascal are type-unsafe.

[20] C.A.R. Hoare, Monitors: An Operating System Structuring Concept. **Communications of the ACM** 17,10 (October 1974), 549-557.

The synchronization concept behind interface modules in Modula.

[21] John B. Johnston, The Contour Model Of Block Structured Processes. **SIGPLAN Notices** February 1971, 55-82.

[22] Anita K. Jones and Barbara Liskov, An Access Control Facility For Programming Languages, Computer Science Department, Carnegie-Mellon University.

A refinement of binding mechanisms in data abstraction languages, permitting the programmer to specify precisely which of the operations defined on an object are permissible.

[23] Donald E. Knuth, Remaining Trouble Spots in Algol 60. **Communications of the ACM** (October 1967).

One of the last papers analyzing Algol 60, describing among other things the binding issues surrounding the loop construct.

[24] Donald E. Knuth, The Art of Computer Programming: Fundamental Algorithms,Vol. 1, 2nd Edition, Addison-Wesley, 1973.

[25] Donald E. Knuth, Structured Programming With GOTO Statements. Computing Surveys (December 1974).

Shows proper and improper uses for the GOTO statement, eventually arguing that the GOTO is necessary for certain cases of multiple exit points from a compound statement. Points out that improper use of the GOTO is harmful, but the GOTO itself is not.

[26] Butler W. Lampson et al, Report On The Programming Language Euclid. SIGPLAN Notices (February 1977).

Reference manual.

[27] Roy Levin, Program Structures For Exceptional Condition Handling, Ph. D. Thesis, Computer Science Department, Carnegie-Mellon University.

Representative of the state of the art in exception handling, presenting the scope issues and a promising solution.

[28] Barbara Liskov and S. Zilles, Programming With Abstract Data Types. SIGPLAN Notices (April 1974), 50-59.

A reasonable tutorial on the concept of an abstract data type, and a basic introduction to CLU.

[29] Barbara Liskov, An Introduction To CLU. In New Directions in Algorithmic Languages 1975, S. Schuman, ed., IRIA, Paris, 1975.

[30] Barbara Liskov and S. Zilles, Specification Techniques for Data Abstractions. Proceedings of the International Conference on Reliable Software, SIGPLAN Notices 10,6 (June 1976), 72-87.

A survey of specification techniques, highlighting those properties of an abstract data type which are visible outside it, and must therefore be precisely specified.

[31] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, Abstraction Mechanisms in CLU. Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12,3 (March 1977).

An introduction to abstraction in CLU, including its iteration construct.

[32] Ralph L. London, A View of Program Verification. Proceedings of the International Conference on Reliable Software, SIGPLAN Notices 10,6 (1975), 534-545.

A gentle introduction to verification.

[33] Ralph L. London, Mary Shaw, and William A. Wulf, Abstraction and Verification in Alphard: A Symbol Table Example, Computer Science Department, Carnegie-Mellon University.

A non-trivial example of a generic type.

[34] Joel Moses, The Function of FUNCTION in LISP, or, Why the FUNARG Problem Should be Called the Environment Problem, Project MAC, Massachusetts Institute of Technology

MAC-M-428 AI-199.

Describes how certain deceptively simple binding mechanisms can cause enormous implementation and conceptualization difficulties.

[35] Peter Naur (ed.) , Revised Report on the Algorithmic Language Algol 60. Communications of the ACM (January 1963), 1-17.

Reference manual.

[36] Eliot I. Organick and Loren P. Meissner, Fortran IV, Addison Wesley, 1974.

Reference manual.

[37] Susan Owicki and D. Gries, Verifying Properties of Parallel Programs: An Axiomatic Approach. Communications of the ACM 19,5 (May 1976), 279-285.

[38] Susan Owicki, Specifications and Proofs for Abstract Data Types in Concurrent Programs, Digital Systems Laboratory, Stanford University TR No. 133.

Data abstraction combined with monitors permits natural extension of Hoare's methodology to parallel programs.

[39] Frank G. Pagan, A Practical Guide to Algol 68, John Wiley & Sons, 1976.

Provides plenty of examples, and enough conventional prose to explain the Algol 68 terminology.

[40] Jacob Palme, New Feature for Module Protection in Simula. SIGPLAN Notices (May 1976).

Turns classes into protected capsules, and permits fine control over sharing with subclasses. These features are now part of standard Simula.

[41] David L. Parnas, Information Distribution Aspects of Design Methodology. Proceedings of the IFIPS Congress 71, Vol. 1 (1972).

The effect of design information changes on system construction.

[42] David L. Parnas, Some Conclusions From an Experiment in Software Engineering Techniques. Proc. AFIPS FJCC vol. 41, AFIPS Press, Montvale, N. J. (1972), 325-329.

How a methodology based on modules facilitated construction of a toy system.

[43] David L. Parnas, A Technique for Software Module Specification With Examples. Communications of the ACM 15,5 (May 1972), 330-336.

Specifying a module as a black box with lights and buttons.

[44] David L. Parnas and D.P. Siewiorek, Use of the.Concept of Transparency in the Design of Hierarchically Structured Systems, Computer Science Department, Carnegie-Mellon University.

More methodology based on modules.

[45] David L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15,12 (December 1972), 1053-1058.

Parnas's concept of what a module should comprise.

[46] David L. Parnas, On a 'Buzzword': Hierarchical Structure. Proceedings of the IFIPS Congress 74 (1974).

It is necessary to specify exactly which relation among modules is hierarchical.

[47] David L. Parnas, The Non-problem of Nested Monitor Calls. Operating Systems Review 12,1 (January 1978).

Points out the difference between the monitor as a synchronization construct, and the monitor as a resource manager.

[48] G.J. Popek et al, Notes on the Design of Euclid. Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12,3, SIGPLAN Notices 12,3 (March 1977), 11-18.

Discussion of, among other things, the scope issues in the design of Euclid.

[49] Craig Schaffert, Alan Snyder, and Russell Atkinson, The CLU Reference Manual, Project MAC, Massachusetts Institute of Technology.

[50] Mary Shaw, Abstraction and Verification in Alphard: Design and Verification of a Tree Handler, Computer Science Department, Carnegie-Mellon University.

An example of a complex relationship between two abstract data types.

[51] Mary Shaw,William A. Wulf and Ralph L. London, Abstraction and Verification in Alphard: Iteration and Generators, Computer Science Department, Carnegie-Mellon University.

Uses data capsules to solve a control abstraction problem.

[52] Richard Sites, Algol W Reference Manual, Computer Science Department, Stanford University STAN-CS-71-230.

[53] Lawrence Snyder, An Analysis of Parameter Evaluation For Recursive Procedures, Computer Science Department, Carnegie-Mellon University.

Comparing the power of parameter mechanisms using program schemata.

[54] R.D. Tennent, PASQUAL: A Proposed Generalization of PASCAL, Department of Computing and Information Science, Queens University.

Advocates using a uniform binding mechanism for declarations and parameters.

[55] Clark Weissman, Lisp 1.5 Primer, Dickenson, 1967.

Introduction to LISP

[56] A. van Wijngaarden (ed.), Revised Report on the Algorithmic Language ALGOL 68. SIGPLAN Notices (May 1977), 1-70.

Reference manual.

[57] Niklaus Wirth, The Programming Language PASCAL (Revised Report), Berichte der Fachgruppe Computer-Wissenschaften, Eidgenossische Technische Hochschule, Zurich.

Reference manual

[58] Niklaus Wirth, Modula: A language for modular multiprogramming, Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich.

Reference manual.

[59] Niklaus Wirth, Toward a Discipline of Real-Time Programming. Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12,3 (May 1977), Communications of the ACM.

The methodology behind Modula.

[60] William A. Wulf, Ralph L. London, and Mary Shaw, Abstraction and Verification in ALPHARD: Introduction to Language and Methodology, Computer Science Department, Carnegie-Mellon University.