CMU-CS-78-105

(Preliminary)
An Informal Definition of Alphard

Paul Hilfinger
Gary Feldman
Robert Fitzgerald
Izumi Kimura
Ralph L. London
KVS Prasad
VR Prasad
Jonathan Rosenberg
Mary Shaw
Wm. A. Wulf (editor)

February 12, 1978

Computer Science Department
Carnegie-Mellon University
Pittsburgh Pa., 15213

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contracts DAHC 15-72-C-0308 and F44620-73-C-0074 (which is monitored by the Air Force Office of Scientific Research), and in part by the National Science Foundation Grant DCR 74-04187.

An Informal Definition of Alphard

Preface

The authors and their colleagues have been experimenting with a collection of ideas about programming languages for several years. Our goals included determining the extent to which language could support contemporary programming methodology, could aid in the construction of verifiable programs, and, at the same time, could be a completely practical programming tool.

In the context of that exploratory spirit it seemed inappropriate to rigidly bind decisions about the details of the language. Hence, although our explorations were carried out in a relatively uniform notation and published under the name "Alphard", there really never was an Alphard language. The astute reader of our previous publications will have noted, and probably will have been frustrated by, the fact that we felt completely free to change the notation from paper to paper as the needs of our exploration seemed to warrant.

With this document we are breaking with our previous strategy. We are now defining a specific language which we expect to serve as the basis of our further research. In the future we do not intend to alter this language in the same free manner as we have in the past. There are two reasons for this shift in strategy: First, although we didn't admit it, much of the language was frozen in our heads, and the minor differences that appeared in published examples only served to confuse our readers. Second, and far more importantly, we believe that the premises on which all the "data abstraction" languages are based are untested in practice. We feel the need to gain experience before we can proceed with any confidence to tackle the next set of exploratory questions. To gain that experience we need to freeze, and to implement, at least some portion of the language -- and that is what we are now doing.

Since we expect to work in the context of the language defined here for some time to come, the language is extremely conservative. Our past experience has been that simultaneously achieving verifiability and efficiency is possible -- but delicate. Hence we have chosen to include *only* features whose implications we fully understand. For example, we have omitted features dealing with concurrency, exceptional-condition handling, and so on. We fully appreciate that these features will be needed in a "production" version of Alphard; they are omitted here because they are still the subject of our research.

The present version of this report carries the word "Preliminary" in its title; we hope to promptly circulate a second version of the report from which this word has been elided. Our purpose in circulating this first version is to solicit comment. We will deeply appreciate any and all critiques of both the language and its presentation. Such comments should be sent to Bill Wulf, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. 15213.

Chapter 1 *Introduction*

The Alphard language has been designed to meet several objectives simultaneously:

To support contemporary programming methodology, and to encourage the development of understandable and modifiable programs. Specifically, we wish to make the abstractions used during the construction of a program explicit in the resulting program text.

To permit formal specification of properties of a program, and to permit *practical* verification (proof) that the program satisfies these specifications.

To permit the programmer to control certain decisions that have traditionally been preempted by the language implementation (e.g., the representation of data structures and method of storage allocation).

To permit the Alphard compiler to generate compact, efficient code. With the aid of an optimizing compiler, we expect to produce better code than is typically produced by assembly language programmers.

In setting these objectives, our principal concern is with high quality, *real* programs -- those which are used extensively and are of significant size and complexity. Many of these programs arise in the area which has been called "systems": compilers, operating systems, and the like; such applications are representative of our concerns although they are not our exclusive focus. Our intended user community consists of relatively experienced professionals rather than casual or student programmers.

The designers of a programming language generally make a number of philosophical decisions that have manifold effects on the product of their effort. We, for example, believe that "power" or "expressiveness" is best achieved through mechanisms which permit the programmer to synthesize more complex facilities out of relatively simpler ones. Thus the *composition*, or *structuring* mechanisms play *the* central role in Alphard; by contrast, for example, the collection of primitive data types is small. The philosophical justifications for this decision are: (1) *all* of the familiar data types can be built from Alphard's primitives, (2) the basic language is much simpler without a large collection of data types, and (3) in making the composition mechanism strong enough to define the familiar data types, we have also made it strong enough to define many more problem-specific ones.

Perhaps nowhere are the language designers' philosophies more evident than in those decisions relating to the tradeoffs between expressiveness, safety, and efficiency. Alphard, like all languages, strives for a balance between these, but our notion of balance is colored by the intended application area(s) and user profiles. For these applications the long term costs of maintaining and running programs far outweigh their initial development costs. Thus we have tilted the balence in favor of those language attributes which contribute to efficiency and maintainability, possibly at the expense of those which facilitate rapid program construction.

We have, for example, not emphasized "expressiveness" in the sense of a large collection of constructs -- each of which is "just right" for a particular situation. We believe that we have, however, supported expressiveness in a larger sense by encouraging program organizations which convey important, abstract information about the way that the program works.

Similarly, we have emphasized safety and efficiency, sometimes at the expense of brevity or convenience. This has led us to restrict some traditional constructs (e.g., scope and parameter rules) and refrain from making some tempting generalizations. We are aware of many areas in which the present design could be generalized in rather obvious ways; we have chosen not to do so, however, when we might compromise the programmer's confidence in both the correctness and performance of his program[1]

1.1. Unusual Aspects of the Language

Many aspects of programming languages have become fairly standard in the past decade. Alphard constructs are intentionally similar in style and meaning to the analogous constructs in other languages. In particular, we have leaned heavily on the Algol-Pascal culture; the syntax of expressions, variable declarations, procedures, and so on, are all derived from this culture. The following, for example, is a fragment of a valid Alphard program and is obviously similar to Pascal:

```
begin
    var x,y,z:int;
    ...
    if x≥y then z:=z+1 fi;
    ...
end;
```

We expect that the similarity between the Alphard constructs and the analogous ones in other languages will aid both the reader of this report and the programmers who use the language.

There are, however, a number of aspects of the language which differ significantly from many traditional languages. This section provides brief notes on these aspects of Alphard. It is, in effect, a list of points at which the reader should be aware that things may not be as expected.

1. *Type Definitions*: The programmer may define a new type through a construct called a form. The form permits both the specification of the abstract properties of (objects of) the new type and the implementation of that type in terms of pre-existing types. Type definitions (forms) may be parameterized; in particular they may accept other form names as parameters. Such forms are called "generic" and define a class of types (e.g., array(T), where T is a type, defines array-of-integer, array-of-real, array-of-set-of-integer, etc.).

2. *Primitive types*: Integer, real, complex, etc. are *not* primitive types in Alphard; similarly, structures such as arrays, records, and references (pointers) are not primitive. All of these familiar notions are available, however. Either they are provided as "syntactic sugar" through some standard abbreviations, or else they are made available to the programmer as a part of a "standard prelude" -- a set of standard definitions which (conceptually) prefaces every program. Specifications for the standard prelude are included as appendix B to this report.

[1] We are convinced that deciding what *not* to include in a language design is *much* harder than inventing clever new things to include.

There are (only) two distinguished types in Alphard; they are distinguished in the sense that they must be considered as part of the language and not as part of the standard prelude. They are "rawstorage" and "boolean". Specifications of these types and their associated operations may be found in appendix B; informally, however:

a. Type "rawstorage": This type corresponds to a vector of contiguous, addressable, untyped memory "cells" of conventional computers (we shall refer informally to a rawstorage unit of length one as a "cell" or a "word" but we make no a committment to the number of bits in each such cell); bit-wise logical, shifting, and integer operations are defined on cells. All other types are (ultimately) represented in terms of objects of type rawstorage, and the definition of this type contains the basic mechanism for associating a "higher level" type with an area of storage. Type rawstorage is distinguished (only) because its implementation cannot be expressed in the language.

b. Type "boolean": Objects of type boolean are primitive (unstructured), and possess values from a set designated {true,false}. Type boolean is distinguished in the sense that, although it can be defined in terms of type rawstorage, it is needed for the definition of other language constructs -- e.g., the conditional statement. The customary operations are provided.

3. *Type Checking*: Most modern programming languages contain some notion of the "equivalence" of types and require that the types of actual parameters to procedures be equivalent to those specified by the corresponding formal parameter definitions. The presence of parameterized and generic type definitions in Alphard makes it advantageous to replace the notion of equivalence by a more liberal notion of "matching". Formal parameter definitions specify a collection of properties which the corresponding actual parameter must possess, specify a collection of properties which are irrelevant, and provide a limited facility for relating properties of distinct parameters. Together these define a class of valid actual parameter types, and provide what is generally called "strong typing"; in particular, the parameter specification and matching is sufficiently strong to ensure verifiability.

4. *Scope Rules*: Alphard's block structure is similar to that of Algol 60: declarations appear at the head of a block, the meaning of an identifier is determined from its nearest enclosing declaration, and so on. Unlike Algol, however, in Alphard the bodies of procedures and forms do not inherit the names of variables available in enclosing blocks. The intent of these scope restrictions[2] is to ensure that all effects of an action can be determined by examining the text immediately surrounding the action itself. An additional benefit is that Alphard can be implemented (very efficiently) using a stack, but without the need for a display.

5. *Operator Overloading*: The meaning of the usual infix operators (e.g., "+", "*", etc.) may be extended to programmer-defined data types. The symbols for, the associativity of, and the precedence of these operators are fixed by the language (see appendix C).

[2] A particular consequence of the scope restrictions -- together with companion restrictions on overlapping actual parameters and selectors -- is to prevent unintended "aliasing". That is, they ensure that within a given scope there is at most one name for a given storage cell.

6. ***Selectors***: The programmer may define the representation of a data structure by means of a selector. Intuitively, a selector defines an algorithm for naming data, just as a procedure or function defines an algorithm for computing values. A selector may be thought of as a procedure that returns a pointer (reference, address) to an element of a data structure; the syntax for defining selectors is therefore similar to that of procedures. "Pointer" is not, however, a type in Alphard; no variables of this type can be declared, and hence the "value" returned by a selector cannot be stored. The effect of this (coupled with some verification requirements) is that selectors are "safe"; most of the (useful) flexibility of general address arithmetic is retained without introducing its corresponding dangers. In particular, it is possible to define a restricted style of "reference" variable completely within the language and to ensure that this type is at least as safe as array indices in other languages.

7. ***Assertions***: Assertions are permitted almost everywhere and special syntax encourages their use in appropriate places. The language in which the assertions are written, however, is not defined by Alphard. The choice of that language is, we believe, a private matter between the programmer and verifier.

8. ***Iteration***: Four iteration statements are provided, three of which are somewhat different from what one might expect.
   a. The do statement repeats its body until the body invokes an explicit exit.
   b. The for statement serves a function similar to the for-step-until construct of Algol 60, but does so in a manner that permits the programmer to define the type of the control variable, the way it is initialized and incremented, and the nature of the test for completion of the loop. These aspects of loop control are all defined in a form (usually a specialized form called a generator).
   c. The first statement provides a special syntax for those common loops that search a data structure and perform one of two actions depending upon whether or not an element with a specified property is found.

   The fourth iteration construct is the familiar while statement.

9. ***Sugaring***: A number of familiar notions such as records and enumerated types are not primitive notions in Alphard. They are provided, however, as abbreviations for the more basic notions from which they are formed[3]

1.2. Style and Conventions of the Report

This report is a precise but informal definition of Alphard; it is neither a primer nor a completely rigorous formal definition. It is intended, however, to be *the* reference for users, implementors, and verifiers. To that end we have attempted to be as precise as our human limitations and the vagaries of English permit. We have consciously adopted the style and tone of the Algol 60 report, which we believe remains the exemplar language definition.

The syntactic definition of the language uses conventional BNF with the following additions and conventions:

---

[3] Pun intended!

1. Key words (reserved words) are denoted by underlining.

2. Metasymbols are denoted by lower-case letters enclosed in angular brackets, e.g., "<stmt>".

3. The symbols { and } are meta-brackets and are used to group constructs in the meta-notation.

4. Three superscript characters, possibly in combination with a subscript character, are used to denote the repetition of a construct (or a group of constructs enclosed in {}). In particular:
   "*" denotes "zero or more repetitions of"
   "+" denotes "one or more repetitions of"
   "#" denotes "precisely zero or one instance of".

   Since it is often convenient to denote lists of things that are separated by some single punctuation mark, we denote this by placing the punctuation mark directly below the repetition character. Thus,

   <vvv> ::= <a> { <b> | <c> }

   defines a <vvv> to be an <a> followed by either a <b> or a <c>.

   <xxx> ::= <a>*

   defines an <xxx> to be a sequence of zero or more a's.

   <yyy> ::= <a> <b>$^{*}_{,}$

   defines a <yyy> to be an <a> followed by zero or more <b>s separated by commas.

   <zzz> ::= {<a> | <b>}$^{+}_{;}$

   defines a <zzz> to be a sequence of one or more things separated by semicolons -- where the "things" may be either <a>s or <b>s.

   <uuu> ::= <a>$^{\#}$ <b>

   defines <uuu> to be either "<a><b>" or simply "<b>"

The semantics of the language are described in English. Proof rules for some constructs are provided in appendix E.

Certain portions of this report describe processes in terms of extra variable creations, text replacements (copying), or other actions. These are informal expositions and at all times the language (compiler) is required to only produce the same net semantic effect. Such expositions should be interpreted in their intended, helpful sense. Obscure consequences of the particular processes will not be supported.

Chapter 2
Fundamental Concepts

The following chapters define the syntax and semantics of Alphard; in this chapter we describe certain pervasive notions that are used in the definition.

A complete Alphard program consists of a collection of declarations and statements which are elaborated[4] to produce some desired effect. Declarations define forms (which, in turn, define classes of ***types***), routines[5] (which may be ***invoked*** to evoke further elaboration), variables, and a number of other entities of lesser immediate importance. Statements define actions to be performed; they may specify selective or iterative elaboration of component statements and expressions. Of particular immediate interest, because they cover the major ideas we wish to discuss, are the notions involved in the elaboration of the declaration of variables and in the elaboration of routine invocations.

The elaboration of a variable declaration, e.g.

```
    var x:vector(int,1,10)
```

begins with elaboration of the ***type description*** (vector(int,1,10)), followed by ***instantiation*** of an ***object*** of the ***type*** resulting from this elaboration (instantiation involves both ***allocation*** and ***initialization***); finally, a ***binding*** of the name to the instantiation is performed.

The elaboration of a routine invocation, e.g.,

```
    f(x,y)
```

begins with the elaboration of the actual parameters (x and y), followed by ***matching*** of the ***nominal type*** of the actual parameters with the ***type descriptions*** of the positionally corresponding formals; if this matching succeeds a set of ***bindings*** is performed[6] and the routine body is elaborated.

The words and phrases in bold-face above, ***type***, ***object***, ..., are representative of the notions we shall discuss in this chapter. Because of mutual dependencies between the notions, however, we shall not discuss them in precisely the order in which they are mentioned above. We have chosen instead an order which attempts to minimizes the forward references.

2.1. Objects, Addresses, and Values

Intuitively an ***object*** is a generalized (and typed) storage cell, or variable; it is used to hold the value of some abstract data type.

---

[4] We use the word "elaboration", in preference to "execution", to connote actions taken at "compile time" *as well as at* "run time". Elaboration may be thought of as an idealized, direct execution of the textual version of the Alphard program.

[5] The word "routine" is used systematically to cover the notions of proc, vproc, func, and sel.

[6] At this point a result object may also be instantiated, but this is not essential to the present discussion.

An object possesses a unique ***(generalized) address***, a ***type***, and a ***value*** (or ***state***). Objects may be dynamically created and destroyed. The address and type of an object are fixed throughout its lifetime, but the value it possesses may be altered.

An object may be primitive, in which case its values (i.e., the values it may possess) are members of an arbitrary set. Otherwise, the object is composed of a sequence of one or more (previously created) objects, called its ***concrete components***. The value of such an object may be taken to be the sequence of values of its concrete components. For the purpose of the following exposition, if x denotes an object, x_i denotes its i^{th} component object.

Two objects may ***overlap***; that is, their values need not be independent. A common case, though not the only one, is that one object wholly ***contains*** the other, as a vector contains its elements. Two objects that do not overlap are called ***independent***. Any logical dependency (i.e., overlap) between the values of two objects is fixed. A newly created object independent of all previously existing objects is called a ***new object***.

The creation of an object is generally associated with ***allocation*** of storage for the object and ***initialization*** of its value. The entire process is called ***instantiation*** and the resulting object is called an instantiation of its type. The first step of instantiation is the elaboration (evaluation) of a type description to yield a type (see section 5.6). Next the object is created. For primitive objects this is a direct operation; otherwise it is achieved by (recursive) instantiation of its concrete components. (Note that at the moment of creation the generalized address of the object is determined.) After allocation, the initialization procedure defined with the base type of the object is invoked as described in section 5.5.

Objects are destroyed by first invoking a finishing procedure defined with the base type of the object (as described in section 5.5), then de-allocating the object (for primitive objects) or destroying its concrete components (for non-primitive objects).

2.2. Type and Type Descriptions

Intuitively, ***type*** is that property of an object which defines its possible behaviors[7]. More formally, a type characterizes the possible values (states) of an object and the set of operations that may be applied to it.

There are two explicit syntactic manifestations of the notion of type in the language: form declarations (which define a class of types), and ***type descriptions*** (which describe a class of object types that may be bound to an identifier in declarations or formal parameter specifications).

Form declarations are defined in section 5.9. For our present purposes it is sufficient to note that: (1) every form has a name, (2) a form may be parameterized, and (3) the form declaration makes available various operations. A subset of these operations (the side-effect producing ones) is called the ***update set***.

[7] Note that objects, not values, are typed. Indeed naked values do not exist in Alphard -- values only exist in objects. Thus, for example, we may speak informally of the "value produced by a procedure", but in fact the procedure returns an object that contains the value.

Type descriptions are used in three contexts: (1) in variable declarations, where they define the type of an object to be instantiated, (2) in formal parameter specifications, where they define the class of legal actual parameters, and (3) in routine definitions, where they specify the type of the object returned. In addition, in both contexts type descriptions define the ***nominal type*** of any object bound to a particular identifier. Thus, the nominal type of an object is the information about its type that can be inferred by accessing the object through a particular identifier.

The distinction drawn in the last paragraph between "type" and "nominal type" is an important, though possibly subtle, one. A "type" is associated with an object and determines all possible behaviors of that object. A "nominal type" is associated with an identifier which, in turn, is bound to an object. The nominal type associated with an identifier determines the possible behaviors that can be caused through that identifier. In the general case a nominal type will be "less specific" than the type of the object to which the identifier is bound.

In the following sections we more formally define the notions of type, type descriptions, and nominal type.

2.2.1 Type

A ***type*** results from the elaboration of a type description (see section 5.6) and consists of a ***base type***, a (possibly null) sequence of ***actual type qualifiers***, and an ***update set***.

A ***base type*** is a form name; it uniquely identifies a class of types. For example, the base type of "vector(real,1,10)" is "vector". In the following, if T is a type, Base(T)[8] represents the base type of T.

An actual type qualifier is intuitively an actual parameter in a type description; hence it corresponds to a formal parameter in a form definition. It may be an object address, an object, a routine name, a type, or a marker denoted unk[9]. In "vector(real,1,10)", the actual type qualifiers are "real", "1", and "10".

A type none of whose qualifiers is unk is called a ***full type***; a type with at least one unk qualifier is called a ***partial type***.

The update set consists of a set of routine designators from among those defined with the base type; specifically, the update set consists of those routines which may have a (visible) effect on an object of the type. If T is a type, Update(T) denotes its update set.

If T is a type, then Qual(T) denotes the sequence of actual type qualifiers of T and $\text{Qual}_i(T)$ denotes the ith element of that sequence.

---

[8] Here and in the sequel we shall use functions such as Base(T), Qual(T), Update(k), etc. to explain semantic aspects of the language; these functions are only part of the semantic exposition, not constructs in the language itself.

[9] The marker unk, which is to be read "unknown", denotes situations in which the corresponding form formal is not considered a part of the nominal type.

2.2.2 Type Descriptions

A ***type description*** is a syntactic construct which describes a class of types and may designate restricted access to objects of those types. A type description consists of a base type, a sequence of ***formal type qualifiers*** and an ***update set***. The elaboration of a type description yields a type or a nominal type.

A base type is (as above) a form name.

A formal type qualifier is either the marker unk or else a description of an object address, an object, a routine, or a type. When used to specify a formal parameter, a formal type qualifier may be an identifier preceded by a "?" symbol; in such cases an "implicit binding" is implied (see section 2.4).

The update set consists of a set of routine names from among those defined with the base type. Update sets give restrictions on the effect-producing actions that may be applied to an object[10]. If D is a type description, Update(D) denotes its update set.

2.2.3 Nominal Type

A ***nominal type***, like a ***type***, consists of a ***base type***, a (possibly null) sequence of ***actual type qualifiers***, and an ***update set***. These notions are defined exactly as in the definition of ***type***.

A ***type*** is always associated with an object. A ***nominal type***, on the other hand, is always associated with an identifier. The nominal type of an identifier may, in the general case, be less specific than the type of the object to which that identifier is bound; however, the type of an object will always ***match*** the nominal type of the identifier.

2.3. Binding

During elaboration, some identifiers become associated with -- bound to -- entities; these entities may be objects, routines, or types. The binding of identifiers to objects is of particular interest and includes both the declaration of variables (and associated instantiation of an object) and parameter passing.

In all contexts in which an identifier may become bound to an object (i.e., in a variable declaration or formal parameter position) there is an associated type description. In the case of a variable declaration, this description determines the type of the object created. In the case of a formal parameter, the type description defines the types of allowed actual parameters. In both cases, however, the type description is elaborated to a ***nominal type*** which determines the permitted uses of the object identified through this identifier.

---

[10] In practice we allow more than just the names of effect-producing operations in the update set part of certain type descriptions, notably those which specify generic formal parameters. In such cases we allow non-effect-producing attribute names as well; this is merely a shorthand for an "assumes clause" (see section 5.12). This abbreviation is permitted because of its similarity of intent to the update set: it describes a set of attributes which the routine or form body requires for correct operation.

In both declarative and formal parameter positions the description of a binding may be preceded by either var or const. The only difference between the two is that in the latter case (const) the update set of the identifier is set to empty; in the former (var) case the update set is determined from the associated type description. A particular consequence of this mechanism is that parameters in const positions are, intuitively, passed "by reference" but cannot be modified by the called procedure.

If k is an identifier bound to an object, then we refer to this object as Obj(k) and to its associated nominal type as Type(k).

2.4. Type Matching

The process of parameter binding requires a notion of what it means for an actual parameter to ***match***, or ***satisfy***, a formal parameter specification. Intuitively this process involves determining that the nominal type associated with the formal parameter "includes", or "covers", the nominal type of the actual -- that is, ensuring that the behaviors permissible through the formal parameter name are among those permissible through the actual parameter name. For simplicity we break this process into three subprocesses: ***subsumption***, ***syntactic satisfaction***, and ***implicit binding***. Each of these is used for a different kind of actual/formal matching as specified below.

A list of actual parameters

$a_1, \ldots, a_n$

is said to ***match*** a list of formals

$f_1{:}t_1,\ f_2{:}t_2,\ \ldots,\ f_n{:}t_n$

where the a_i are objects, types, or routine names, if there exists a binding of objects, types, and routines to the implicit formals in the t_i such that if each f_i is bound to a_i, then for each i,

1. If t_i is a description of a routine (proc, vproc, func, or sel), then a_i is the name of a "proc", ..., or "sel" with formal parameters identical to those of t_i after possible renaming of formal parameters.

2. If t_i is "form" (or "pform), then a_i is a type (if a_i is a partial type t_i must have been pform) and the assumed definition of f_i (see section 5.12) is ***syntactically satisfied*** (see section 2.4) by a_i.

3. If t_i is a type description, then a_i must be an object such that when t_i is elaborated, t_i ***subsumes*** Type(a_i).

In addition, all implicit formals bound to types must be bound to types syntactically satisfying the assumed definition of the form (see section 5.12).

The notions of subsumption, syntactic satisfaction, and implicit binding are defined below; we begin with the notion of subsumption -- the kind of matching used when an object parameter is expected.

<u>*Definition:*</u> We say that a (nominal) type T_f ***subsumes*** a (nominal) type T_a (in symbols, $T_f >> T_a$) iff:

1. $\mathrm{Base}(T_f) = \mathrm{Base}(T_a)$.

2. $\mathrm{length}(\mathrm{Qual}(T_f)) \leq \mathrm{length}(\mathrm{Qual}(T_a))$. Note: if $\mathrm{length}(\mathrm{Qual}(T_f)) < \mathrm{length}(\mathrm{Qual}(T_a))$, the formal qualifier sequence of T_f is extended on the right with a sufficient number of <u>unk</u>'s.

3. For each qualifier of T_f, i.e., $\mathrm{Qual}_i(T_f)$:

   a. If $\mathrm{Qual}_i(T_f)$ is <u>unk</u>, $\mathrm{Qual}_i(T_a)$ may be anything.

   b. If $\mathrm{Qual}_i(T_f)$ is a type, $\mathrm{Qual}_i(T_a)$ is also a type and $\mathrm{Qual}_i(T_f) >> \mathrm{Qual}_i(T_a)$

   c. If $\mathrm{Qual}_i(T_f)$ describes a routine, then $\mathrm{Qual}_i(T_a)$ is also a routine and $\mathrm{Qual}_i(T_a)$ matches $\mathrm{Qual}_i(T_f)$.

   d. If $\mathrm{Qual}_i(T_f)$ is an object of base type U, $\mathrm{Qual}_i(T_a)$ is also an object of base type U and the value of the result of applying &= for U to $\mathrm{Qual}_i(T_f)$ and $\mathrm{Qual}_i(T_a)$ would be true.

4. $\mathrm{Update}(T_f) \subseteq \mathrm{Update}(T_a)$.

In some cases condition 3d cannot be checked at compile time. At the discretion of the implementors, the compiler may provide the options of generating warnings, generating checking code, or refusing to compile such cases.

<u>*Definition:*</u> Two types are ***identical*** if each subsumes the other.

In Alphard, both routines and <u>forms</u> may be "generic". That is, they may require types as parameters -- or, equivalently, they may have parameters whose type is not specified in the routine (<u>form</u>) header. In such cases there will be an "assumes clause" which specifies the properties that the routine (<u>form</u>) assumes about the generic parameter; this clause gives sufficient information to check all uses of the parameter locally. In order for a given use of the <u>form</u> or routine to make sense, the actual parameter must at least meet the syntactic assumptions made about it. Thus the notion of matching formal and actual parameters in such cases involves of determining whether the actual parameter ***syntactically satisfies*** the formal parameter assumptions [11].

[11] More generally, of course, a proof will be required to demonstrate that the actual parameter makes semantic sense as well.

> ***Definition:*** Given an assumed declaration of a generic parameter, T:
>
> ```
> form T
>     specs
>     <definitions of f1...fn>
> ```
>
> and a candidate actual type
>
> ```
> Q(a1,a2,...)
> ```
>
> whose base type is declared
>
> ```
> form Q(p1,...)
>     specs ...
> ```
>
> we say that the type $Q(a_1,...)$ ***syntactically satisfies*** T if textual substitution of "$Q(a_1,...)$" for "T" uniformly throughout the specifications of T results in T's specifications containing declarations of $f_1,...f_n$ identical to those in Q's specifications (ignoring assertions and implementations), though possibly only after suitable renaming of formal parameters.

In the process of determining whether an actual parameter of type T matches a formal parameter specification we may discover that $Qual_i$ of the formal is an identifier preceded by "?". Such identifiers are called "implicit formal parameters", and are "implicitly bound" to corresponding (qualifiers of the) actual parameters. Such bindings are performed before other matching.

> ***Definition:*** Let T_f be the formal type and T_a be the actual type. If, in determining whether $T_f >> T_a$, $Base(T_f)$ or $Qual_i(T_f)$ is an identifier preceded by a "?", the identifier is ***implicitly bound*** to the corresponding $Base(T_a)$ or $Qual_i(T_a)$ and becomes an "implicit formal parameter". The nominal type of an implicit formal is made identical to the nominal type of the corresponding form formal. Note that only one binding is established for such identifiers, so multiple occurrences must be consistent.

In the procedure declaration

```
proc P(x:vector(int,?lb,?ub)) is ...
```

for example, "lb" and "ub" are such implicit formals. They are, respectively, the lower and upper bounds of an actual parameter vector. Thus, if some program fragment contains

```
... var y:vector(int, 1, 10); ... P(y) ...
```

then 1 and 10, respectively, will be implicitly bound to the formals lb and ub.

The following table attempts to recap the essential aspects of the notion of actual/formal parameter matching:

Formal	Actual	Matching Rule
x:form	full type	syntactic satisfaction
x:pform	type	syntactic satisfaction
x:<routine description>	routine name	point 3 of match rule
x:?T	object	Type(object) must syntactically satisfy T
x:<type description>	object	subsumption
object	object	equality under &= for the type of the formal
unk	anything	always matches
?T	match after implicit binding	
routine name	routine name	same routine

Chapter 3
Basic Lexical Structure

3.1. Symbols

```
<letter>              ::=   A | B |...| Z | a |...| z
<digit>               ::=   0 | 1 |...| 9
<alphanumeric>        ::=   <letter> | <digit> | `
<special symbol>      ::=   <basic symbol> | <operator>
<basic symbol>        ::=   begin | end | endof | ; | : | ( | ) | $ | , | :: | & |
                            if | then | else | fi | case |
                            of | esac | fo | with | in | ni | first | suchthat | from | do | od |
                            for | exitloop | leave | skip | assert |
                            var | const | aux | as specified | = | init |
                            final | unk | ? | proc | vproc | func | sel | label |
                            note | eton | ! | elif | elof | pform | while | > | < | . | * |
                            copy | alias | form | inline | pre | post | rule |
                            is | forward | external | specs | impl | shared |
                            invariant | initially | axiom | repmap |
                            record | enumerated | assumes | value | generator
<operator>            ::=   <binary operator> | <unary operator>
<binary operator>     ::=   ↑ | * | / | div | rem | + | - | < | ≤ | = | <> | ≥ | > | and | or | cand | cor | imp |
                            <assign op>
<unary operator>      ::=   + | - | not
```

Typographical features such as blanks (spaces), ends of lines, etc., are generally not significant (but see section 3.4.3); an implementation may use them to delimit identifiers, numbers, etc. Outside strings, no such features may appear immediately after the symbol "&" or "?", or around the symbols "." and "$" when they are used as described in sections 3.4.2 and 4.3.

Upper and lower case letters are distinct. Also, note that the grave symbol is considered a (significant) alphanumeric and thus may be used in constructing identifiers; it is intended that this be used to improve program readability by separating mnemonically signifcant portions of such identifiers.

Basic symbols such as begin are conceptually single characters and are underlined in this report to emphasize that fact. An implementation, however, must reserve (all upper/lower case spellings of) the corresponding identifiers to denote these symbols. Thus "BEGIN", "begin", "Begin", etc. are all interpreted as the basic symbol begin; we strongly encourage, however, consistent use of one spelling in a given program.

3.2. Comments

The following two commenting construots are lexically equivalent to a space (blank) character when they appear outside of strings.

note <any sequence not containing the lexeme "eton"> eton
!<any sequence up to end of line>

The first commenting construct encountered in a line takes precedence over any contained within it.

3.3. Identifiers

3.3.1 Syntax

```
<identifier>           ::=    <letter> {<alphanumeric>}*
<special identifier>   ::=    &start | &finish | &next | &done | &value | &subscript | &<operator>
<identifier list>      ::=    <identifier>,+
```

3.3.2 Examples

```
A
a3
TheDogTheCatChased
The'Dog'The'Cat'Chased
the'dog'the'cat'chased
start
&start
&=
```

All the above identifiers are distinct.

3.3.3 Semantics

Identifiers have no inherent meanings. They identify objects, forms, types, procedures, selectors, statements, and parameters. Declarations establish the meanings of identifiers within particular scopes.

Two identifiers are defined to be ***similar*** if they differ at most in the typographical case used to spell them; thus "ABC", "Abc", "aBc", etc. are all similar. Except when used as routine names, similar identifiers may not be declared in the same scope[12].

Special identifiers denote entities of special significance in the language. They may be defined but never directly referenced; they are invoked as the consequence of using some other construct defined by the language. A simple example of the use of such symbols appears in section 3.4.1, where the language-defined notion of "+" invokes the user-definable function named "&+"; more interesting examples may be found in sections 4.7 and 4.8. (Requirements on the definitions of such routines appear in appendix C.)

3.4. Special Rewrite Rules

In order to simplify the language definition, a number of familiar and convenient notations are provided indirectly rather than as a part of the syntax. To accomodate these, we define several "rewrite rules" that transform programs from the more familiar notation to that described by the report. These transformations convert infix operators to function invocations, provide "qualified names", and introduce semicolons. We shall use the notation C_1 --> C_2 to describe some of these transformations; the notation means that constructs of the form C_1 are transformed into constructs of the form C_2.

[12]This restriction is imposed in order to prevent subtle errors arising from the use of similar identifiers in the same scope. Routines are exempted from the restriction in order to permit operator overloading.

3.4.1 *Operators*

Neither the syntax nor semantics of Alphard includes the traditional notion of arithmetic or boolean expressions with infix operators. Rather, the language is defined as though all operations were expressed as function invocations. In order to permit the user to write programs in the more familiar infix-expression format, however, two transformations are performed. First, the input text is fully parenthesized in order to observe the following precedences and associativities:

1. Associativities: The operators of highest and lowest precedence are right associative; the remainder are left associative.

2. Precedence:

   ↑ (highest precedence)
   * / div rem
   + -
   ≤ < = ≠ > ≥
   not
   and cand
   or cor
   imp
   := +:= -:= *:= etc. (lowest precedence)

After being parenthesized, expressions are converted to functional form. If α and β denote arbitrary unary (monadic) and binary (diadic) operators, respectively, then the following transformations are performed:

$\langle term\rangle_1\ \beta\ \langle term\rangle_2 \dashrightarrow \&\beta(\langle term\rangle_1, \langle term\rangle_2)$

$\alpha\langle term\rangle \dashrightarrow \&\alpha(\langle term\rangle)$

$\langle term\rangle_1\ \beta{:=}\ \langle term\rangle_2 \dashrightarrow \&{:=}(\langle term\rangle_1, \&\beta(\langle term\rangle_1, \langle term\rangle_2))$

where the <term>s denote any phrases balanced in parentheses.

After being placed in functional form, three of the relational operators are rewritten as the boolean negation of one of the remaining three:

$\&{\neq}(t_1,t_2) \dashrightarrow \&\underline{not}(\&{=}(t_1,t_2))$

$\&{\le}(t_1,t_2) \dashrightarrow \&\underline{not}(\&{>}(t_1,t_2))$

$\&{\ge}(t_1,t_2) \dashrightarrow \&\underline{not}(\&{<}(t_1,t_2))$

In addition, two of the boolean operators are rewritten as conditional expressions:

$t_1\ \underline{cand}\ t_2 \dashrightarrow \underline{if}\ t_1\ \underline{then}\ t_2\ \underline{else}\ \text{false}\ \underline{fi}$

$t_1\ \underline{cor}\ t_2 \dashrightarrow \underline{if}\ t_1\ \underline{then}\ \text{true}\ \underline{else}\ t_2\ \underline{fi}$

This rewrite is required to avoid the possibility of undefined argument values in invocations.

Note that, as stated earlier, symbols of the form &<operator> may be defined by the user. Thus, by giving a definition to "&+", the programmer gives a definition to the operator "+"; this does not allow redefinition of existing operators, but does allow these operators to be extended to new types.

3.4.2 Name Qualification and Subscripting

It is often convenient to refer to the (visible) components of an object by symbolic names; for example, the components of a record have traditionally been named in this way. The conventional syntax allows "X.y" to denote the y component of X.

The syntax of Alphard does not support such "dotted name" qualification directly, but instead uses the functional form, y(X). To permit the dotted-name notation and user-defined subscripting, qualified names are transformed in two steps. First, dotted names are eliminated in favor of a functional form:

```
        <qualname>.<identifier>  -->  <identifier>(<qualname>)
```

where <qualname> is any sequence of identifiers (including special identifiers), '.'s, '$'s, and sequences of lexemes balanced and enclosed in parentheses or square brackets. The rule is applied right-to-left; thus, for example

```
        A.y  -->  y(A)
        Q.g[s.t].f  -->  f(g(Q)[t(s)])
```

After all dots have been removed, square brackets are removed:

```
        <term>[<expression list>]  -->  &subscript(<term>,<expression list>)
```

Thus, the example above becomes

```
        Q.g[s.t].f --> f(g(Q)[t(s)]) --> f(&subscript(g(Q),t(s)))
```

Note that the user may define the selector &subscript and hence may specify the access algorithm for a type.

3.4.3 Automatic Introduction of Semicolons

The effect of the following transformation is to eliminate the need for explicit semicolons to separate declarations or statements when those semicolons would fall at the end of a text line. According to the syntax in this report, certain phrases are separated from each other by semicolons. In those cases where the final lexeme on a line could end such a phrase, e.g.,

end,), fi, esac, fo, ni, od, exitloop

and the next lexeme (i.e., excluding comments) could begin such a phrase, e.g.,

begin, (, if, case, with, first, do, for

the compiler automatically inserts a semicolon between the two unless, on the basis of preceding symbols it is possible to determine that doing so would render an otherwise syntactically valid program into an invalid one.

3.5. Special literals

Certain well-established literal denotations exist for some types (e.g., integer, real, boolean).

3.5.1 Syntax

```
<special literal>      ::=   <unsigned integer> | <unsigned real> | <string> | <boolean> | <radix>
<unsigned integer>     ::=   {<digit>}+
<unsigned real>        ::=   <unsigned rational>{E<scale-factor>}# | <unsigned integer>E<scale-factor>
<unsigned rational>    ::=   <unsigned integer>.<unsigned integer>
<scale-factor>         ::=   {+|-}#<unsigned integer>
<string>               ::=   "<any sequence of characters with all quotes doubled>"
<boolean>              ::=   true | false
<radix>                ::=   {<alphanumeric>}+#<alphanumeric>
```

3.5.2 Examples

```
        3
        147.5E-3
        32#8
        true
```

```
    "ABcdEF"
    "He said,""Ha!"""
```

3.5.3 Semantics

<radix> literals are of type rawstorage. The <alphanumeric> following the "#" character specifies the representation base. The values of the alphanumerics are interpreted as follows: 0-9 denote 0-9, A-Z denote 10-35, a-z denote 36-61. Note that 0, 1 and ' (zero, one and grave) are not legal base denotations.

Chapter 4
Program Structure, Expressions and Statements

4.1. Program Structure, Blocks

A compilation unit may be either a block or a set of declarations. If it is a block, it is a "program" in the traditional sense -- a stand-alone computation. If it is a set of declarations, the scope of the declarations is system-dependent.

4.1.1 Syntax

```
<compilation unit>    ::=    begin <block> end  |  <exec decl list>
<block>               ::=    {<exec decl list>;}* {<stmt>};+ {;}*
<exec decl list>      ::=    { <exec decl> };*
<exec decl>           ::=    <var decl>  |  <const decl>  |  <proc decl>  |  <form decl>  |  <label decl>
```

4.1.2 Semantics

A block specifies a computation whose effect is as though the following order of execution were observed:

1. Elaborate the declarations in the order given (see 2.2 and 5.5).

2. Elaborate the statements (<stmt>s) in sequence (aside from exits; see 4.9).

3. Destroy the objects created in 1 in the reverse order of declaration.

The scope of all declarations in a block is the text of the block, where not superseded by nested declarations.

4.2. Expressions and Statements

Expressions and statements designate actions to be performed. Their elaboration results in changes in the execution state of the program. Expressions differ from statements only in that their elaboration may "produce values" as well as performing other actions; statements only perform actions. For definitional brevity and convenience, every expression is considered to be a statement, but not conversely. When an expression is used in a context requiring a statement, its "value" is discarded.

Somewhat more precisely, the "value produced by an expression" is an object resulting from its elaboration; the type of this object is uniquely determined by the rules stated in the remainder of this chapter. This resulting (unnamed) object exists until any immediately enclosing expression or statement that uses it has finished execution.

4.2.1 Syntax

```
<expression>          ::=    <invocation> | <conditional expression> | <value expression> |
                             <with expression> | <first expression>
<stmt>                ::=    <expression> | <loop stmt> | <exit stmt> | <null stmt> |
                             <inner block> | <labeled stmt> | <assert stmt>
<labeled stmt>        ::=    <identifier> : <stmt>
```

4.2.2 Semantics

Statement labels are used by exit statements (section 4.9). The effect of an exit statement is to force control to the point immediately following the labeled statement whose label is used in the exit. Labels must be declared (see section 5.4) and may be used to label only one statement within the scope of their declaration.

4.3. Invocations

4.3.1 Syntax

```
<invocation>          ::=    <special literal> | <simple invocation>{<actuals>}* | (<invocation>)
<actuals>             ::=    ({<actual>}*,)
<actual>              ::=    <expression> | <type description>
<simple invocation>   ::=    {<identifier>$ }* <identifier> | <special identifier>
```

Infix and prefix operators fall under this syntax by the rewrites of section 3.4.1. Subscripting, denoted by "[...]", falls under this syntax by the rewrites of section 3.4.2. Note: <special identifier>s may not appear in source programs; they result only from these rewrites.

4.3.2 Examples

The most obvious <invocation>s are those denoting routine "calls", e.g.:

```
sin(x)
integrate(F,a,b,eps)
```

In addition, however, <invocations> result from the rewrite rules for infix operators and subscripting, e.g.:

```
&:=(a, &+(b,c))
&:=(x, &subscript(V,i))
```

Finally, <invocation>s occur as part of type descriptions:

```
vector(int,1,unk)
```

4.3.3 Semantics

A simple invocation may designate a type, an object, or a routine (a procedure or selector) as indicated in chapter 5. Identifiers may designate multiple entities in any given context (operator overloading), so some means of resolving the conflict is necessary. An identifier may be qualified on the left by the name of the form containing its definition; such qualifiers are separated by "$". Alternatively, the proper definition may be determined by examining the types of the actuals -- that is, by choosing that definition for which type checking (section 2.4) succeeds. It is an error if the compiler cannot disambiguate statically (i.e., at compile-time).

Assuming that the entity designated by G is uniquely determined, an invocation such as:

$G(e_1,e_2,\ldots,e_n)$

denotes an elaboration and possibly a resulting value as follows:

1. The actual parameters, e_i, are elaborated in an undefined order. An e_i designating a routine without an argument list designates that routine, rather than the value resulting from its execution. The results of this elaboration are objects, types, and routines (see section 5.6 for the evaluation of partial types).

2. The number of formal parameters of G must be *n* (n≥0). The actuals must ***match*** the formals of G (see section 2.4); it is an error if they do not.

3. Each formal parameter of G which designates a routine (as in "f:proc(...)"), a type (as in "T:form"), or a reference parameter (as in "var x:..." or "const x:...") is bound to the corresponding e_i (also see section 5.7).

4. Each object formal (see 5.7) with an empty <binding> is treated as if it were specified const (see below).

5. For each formal identifier, k, designated to be a const parameter (see section 5.7), Update(k) is made empty. For all other k bound to objects, Update(k) is derived from the type description of the formal parameter.

6. var and const parameters are checked for possible overlap (see section 2.1). In order for the <invocation> to be legal it is necessary that either

   there is no overlap between an actual parameter that stands in a var position and any other actual in either a var or const position.

   or

   all overlapping positions are designated alias in the formal parameter specifications.

7. For each formal identifier, k, designated to be a copied parameter, a new object of the same type is instantiated. The values of the actual parameters (see sections 5.6 and 5.7) are copied into these variables from the corresponding e_i using the "&:=" procedure defined for that type; it is an error if the "&:=" procedure is not defined for the type.

8. If G is a routine and returns a value, or if it is a selector (see section 5.8), its definition contains a type description which specifies the type of its result. This description is elaborated. If G is a vproc or func, an object of the type is instantiated to receive the "value" that will be returned. If G is a selector no object is instantiated; the type description defines the type of the object whose (generalized) address is returned. Note that a vproc or func must specify a full type for the result; a sel need only specify a partial type.

9. If G is a routine its body is elaborated with the established bindings. If G is a type description either an instantiation or a matching is performed, depending on the context.

10. Any auxiliary objects (i.e., copied parameters or actuals which are themselves result objects of procedures) are deallocated.

It should be noted that, by the rules above, the invocation of a parameterless procedure, P, is necessarily written "P()".

4.4. Conditional Expressions

4.4.1 Syntax

```
<conditional expression> ::= <if expression> | <case expression>
<if expression>          ::= if <expression> then <block> { elif <expression> then <block> }* {else <block>}* fi
<case expression>        ::= case <expression> of <case> { elof <case> }* { else <block> }* esac
<case>                   ::= {<expression>},+ :: <block>
```

4.4.2 Examples

```
if a[i]>max then maxp := i; max := a[i] fi
y := if x>z then x else z fi
if a<b then t := 1 elif a<c then t := 2 else t := 3 fi
case IC of
    ADD:: MB := C[EA]; R := R+MB elof
    SUB:: MB := C[EA]; R := R-MB elof
    MUL:: MB := C[EA]; R := R*MB else
    ERROR
esac
T := case n of 1::MALE elof 2::FEMALE else NEUTER esac
```

4.4.3 Semantics

Conditional expressions denote expressions and statements to be evaluated conditionally. Such an expression has a value if

a. All <block>s in it are single expressions,

b. All these expressions are of identical type (this type becomes the type of the expression), and

c. An else clause is present.

The expression

if B_1 then S_1 elif B_2 then S_2 ... elif B_n then S_n else S_{n+1} fi

is equivalent to

```
if B1 then S1 else
    if B2 then S2 else
        ...
            if Bn then Sn else Sn+1 fi
        ...
    fi
fi
```

In the expression

if B then S_1 else S_2 fi

B must have a result of type boolean, and the elaboration of B must not have observable effects. If the value of B is true, S_1 is evaluated, otherwise S_2 is evaluated. If the if expression occurs in a context requiring a value, the value is that of the expression chosen (by the rules above, S_1 and S_2 must be simple expressions of the same type). In the absence of an else clause, S_2 is taken to be skip (see section 4.10).

In the expression

```
case E0 of
    E11, ... E1n1 :: S1 elof
    E21, ... E2n2 :: S2 elof

    Em1, ..., Emnm :: Sm else
    Sm+1
esac
```

E_0, E_{11}, ..., E_{mn_m} must all be expressions of the same type. E_0 is evaluated. The E_{ij} are evaluated (in unspecified order) and the results are compared with E_0 using the &= operator for Type (E_0). It is an error if there is no such operator. The evaluation of E_0 and the E_{ij}'s must not have observable effects. As soon as a match yields true (say with E_{ij}), S_i is evaluated. If all matches fail, S_{m+1} is evaluated. Exactly one block S_i is evaluated for each correct evaluation of the case expression. The value of the case expression is that of the S_i evaluated (again, each S_i must be a single expression).

4.5. Value Expression

4.5.1 Syntax

```
<value expression>   ::=     value <identifier>{:<obj type>}* of <block> fo
```

4.5.2 Examples

```
S := value y:int of  y:=0; for x:invec(A) do y:=y+x od fo
value A of Munge(A,43) fo
```

4.5.3 Semantics

A value expression is used to convert a <block>, and hence a sequence of declarations and statements, into a (value-yielding) expression. In the expression

```
value x:T of S fo
```

the variable x (whose scope is S) is instantiated and S is executed. If ":<obj type>" is omitted, as in

```
value x of S fo
```

the existing instantiation of x is used. In both cases, the result of the value expression is the object Obj(x).

4.6. With Expressions

4.6.1 Syntax

```
<with expression>    ::=     with <with list> in <block> ni
<with list>          ::=     { <identifier>:<invocation> }+,
```

4.6.2 Examples

```
with  Z:A[i].son[k]  in Z.age := 0; Z.number := k ni
with  R:x.y.z,  Q:x.y.w  in var s:T; s := Q; Q := R; R := s ni
```

4.6.3 Semantics

A with expression provides a local shorthand for complicated invocations. The phrase

```
with x:R in S ni
```

causes elaboration of R, binding of x to R, and elaboration of S with that binding. If the <block> is a single expression, the with expression yields a value (the value of the <block>).

4.7. First Expression

4.7.1 Syntax

```
<first expression>   ::=   first<template> suchthat <expression> {then<block>}* {else <block>}* fi
<template>           ::=   <identifier> from {<identifier>:}*<type description> | <identifier> from <invocation>
```

4.7.2 Examples

```
first i from upto(1,n) suchthat A[i]>max then max := A[i]; jmax := i fi
y := first x from invec(A) suchthat x>max then x else 0 fi
```

4.7.3 Semantics

The first expression invokes the generator specified in its template (see section 5.11) to produce a sequence of values. These values are tested in turn by the suchthat clause, which must be a boolean expression and which may not have observable side effects. If the first expression

```
first x from g:Q suchthat B then S1 else S2 fi
```

occurs in a context where a value is not required, its semantics are precisely those of the statement

```
L1: begin
    L2: begin
        var g:Q;  &start(g);
        do
            if &done(g) then &finish(g); leave L2 fi
            with x:&value(g) in if B then S1;&finish(g); leave L1 fi ni
            &next(g)
        od
        end L2
    S2
    end L1
```

where &start, &done, etc. are provided by the form (or generator) Q (see also sections 4.8 and 4.9). If either the then or the else clause is absent, it defaults to skip (see section 4.10). If "g:" is absent, an elsewhere unused identifier is substituted by the compiler. If the full type in the template is absent, the declaration (var g:Q) is omitted; in such cases an existing instantiation (namely "g") is used.

Note that the statement S_2 in the expansion above is outside the scope of the declarations of g and x; neither of these may be referenced in S_2.

If the first expression occurs in a context requiring a value, S_1 and S_2 must both be present and be single expressions of identical type (say T). In these contexts, the semantics are precisely those of

```
value t:T of first x from g:Q suchthat B then t:= S1 else t:=S2 fi fo
```

4.8. Loop Statements

4.8.1 Syntax

```
<loop stmt>      ::=   <simple loop> | <while stmt> | <for stmt>
<simple loop>    ::=   do <block> od
<while stmt>     ::=   while <expression> <simple loop>
<for stmt>       ::=   for <template> <simple loop>
```

4.8.2 Examples

```
do
    if x=y then exitloop fi
    if x>y then x := x-y  else y := y-x fi
od

while x≠nil do P(car(x)); x := cdr(x) od

for x from invec(a) do x := 0 od
```

4.8.3 Semantics

The simple loop; "do S od", executes S repeatedly; it will terminate only if an exit command (see section 4.9) is executed. The while loop, "while B do S od", is semantically equivalent to

```
    do if not(B) then exitloop fi; S od
```

The for loop, "for x from g:Q do S od", is semantically equivalent to

```
    begin
    var g:Q;   &start(g)
    do
        if &done(g) then exitloop fi
        with x: &value(g) in S ni
        &next(g)
    od
    &finish(g)
    end
```

As in the first expression (section 4.7), if "g:" is absent, an elsewhere-unused identifier is substituted by the compiler. If the full type in the template is absent, the declaration (var g:Q) is omitted and an existing instantiation is used.

4.9. Exit Statements

4.9.1 Syntax

```
<exit stmt>          ::=     exitloop | leave <identifier>
```

4.9.2 Examples

```
    leave L
    exitloop
```

See also sections 4.8 and 4.7.

4.9.3 Semantics

The statement "leave L" occuring within a statement labeled "L" (or a routine named "L") causes evaluation of the innermost such statement (routine) to terminate; execution resumes at the point it would have if the statement (routine) had terminated normally. (Note: if the relative nesting of the labeled statement is such that, had the leave not been executed, objects would have been deallocated and final clauses executed, these same deallocation actions and finalizations are performed in the same order as part of the leave.)

An exitloop causes termination of the innermost loop statement (do, while, or for, section 4.8) containing the exitloop.

4.10. Null Statement

4.10.1 Syntax

```
<null stmt>          ::=     skip
```

4.10.2 Semantics

The null statement does nothing.

4.11. Inner block

4.11.1 Syntax

```
<inner block>        ::=     begin <block> end  |  begin <block> endof {<identifier>}*
```

4.11.2 Semantics

The declarations and statements of the block are executed as given in section 4.1. If the optional identifier is present, it must match the label of the inner block or the name of the routine whose body is the inner block.

4.12. Assert Statement

4.12.1 Syntax

```
<assert stmt>        ::=     assert <assertion>
```

4.12.2 Examples

```
do assert {GCD(x,y) = GCD(s0,y0) };
   if x=y then exitloop fi
   if x>y then x -:= y else y -:= x fi
od
```

4.12.3 Semantics

An assert statement indicates a condition that must be true when control passes through the statement. It has no semantic effect. The syntax of <assertion> is not specified by the language (other than that the assertion text must be enclosed in, and balanced in brackets, "{...}"). It is the province of a verifier or verifying compiler only.

Chapter 5
Declarations

Declarations define routines (proc, vproc, func, and sel declarations) and classes of types (form declarations), specify the instantiation of objects (var, const), and bind identifiers to these entities.

5.1. Scope of Declarations

The *scope* of a declaration -- the program text in which the binding it establishes is valid -- depends on the kind of declaration and the place it appears. In the sequel, ***normal Algol scope*** means the innermost block containing the declaration, including all blocks it encloses that do not redefine the identifier. ***Restricted Algol scope*** is the same as normal Algol scope, but excludes the text of routine and form declarations.

Generally identifiers naming routines and forms obey Algol scope rules; identifiers naming objects (i.e., variable names) obey restricted Algol scope. Thus, no free variable names appear in routine or form bodies; all variables are either locally declared or passed in through the parameter list. A few additional scope restrictions are discussed later.

5.2. Auxiliary Declarations

Any declaration may be preceded by the keyword aux. Identifiers defined in such declarations may not be used (except within the assertion language). Auxiliary declarations serve as modeling tools in the specifications; the entities described by such declarations may or may not exist in the implementation. If such entities do exist, they may be implemented in a manner different from that described in the aux declaration. (Note, however, that the clause as specified may be used in an implementation to force precisely the representation appearing in an aux definition of an entity).

An auxiliary declaration of a boolean-valued function, for example, might be conveniently used to express a condition that is useful for verification or specification purposes. No obligation to actually implement this function (whose implementation might be undesirable or impractical in some environments) is implied by the aux declaration. Consider, for example, the form:

```
form DirectedGraph(size:int) is
   specs
      ...
      aux func IsTree(g:DirectedGraph):boolean
          post { returns true iff g is a tree };
      ...
   end DirectedGraph;
```

This form provides the predicate "IsTree" which tests an arbitrary directed graph for treeness; since the predicate is specified aux it can only be used in specifications, not in code.

5.3. Remote Definitions

Occasionally the detailed implementation of an entity (variable, routine, or form) may appear at a point remote from its declaration. The following important cases arise:

1. forward: It may be necessary to mention an entity before is defined; this is logically necessary in mutually recursive routine and form definitions. A forward indicates that the required definition appears later in the current program text.

2. as specified: In the implementation of a form it may be desirable to define an implementation of an entity to be identical to its specification; such definitions are denoted as specified. Somewhat similarly, a form implementation may mention an (object) parameter of the form, describing it as specified, to denote that a run-time representation of the parameter is to exist.

3. external(<system specs>): The definition of some entities may be defined external to the present program text, e.g., on a "file" or a "library" supported by the host system. In such cases the entity may be defined as external. The <system specs> is a system-dependent notion (and syntax) that describes the place where the definition is to be found (e.g., in a particular "file").

5.4. Label Declarations

5.4.1 Syntax

```
<label decl>          ::=     label <identifier list>
```

5.4.2 Examples

```
label L1, EXIT, Rethink;
```

5.4.3 Semantics

Labels, like all other identifiers, must be declared before use. A label may be "placed" (used to label a statement) only once in the scope of its declaration.

5.5. Object Declarations

5.5.1 Syntax

```
<var decl>            ::=     <aux> var {<obj decl group>{<init fin clause>}*},+
<const decl>          ::=     <aux> const {<obj decl group> {<init fin clause>}* | <const assign>},+
<aux>                 ::=     {aux}*
<obj decl group>      ::=     <identifier list> : <obj type>
<obj type>            ::=     <type description> | as specified
<init fin clause>     ::=     = <expression> | {init<stmt>}* {final <stmt>}*
<const assign>        ::=     <identifier list> = <expression>
```

5.5.2 Examples

```
var a,b,c:int, g:real
aux const a:int = 5
var INF:file(vector(mumble,1,unk)) init open(INF)
var q,x,r:as specified
var Q:queue(int) init new(Q) final destroy(Q)
const ADD=0, SUB=1, MULT=2
```

5.5.3 Semantics

Object declarations may occur inside form declarations or in blocks; their meanings in the two contexts differ. See section 5.9 for a discussion of their meanings in form declarations.

In blocks, object declarations have restricted Algol scope (see section 5.1). Semantically, constant declarations (<const decl>s) differ from variable declarations (<var decl>s) only in that, outside the initialization and finalization clauses, Update(c) is empty for c a constant; for variables the update set is determined from the <type description>.

The <obj decl group>s within an object declaration are processed in unspecified order (note, however, that declarations are processed in left-to-right order; thus the programmer may impose an ordering if that is appropriate). For each group, the full type is evaluated, and for each identifier, a new object of that type is instantiated and bound to the identifier. If an init clause is present, it is executed. The declaration

```
. . .x,y,z:T=E
```

is equivalent to

```
. . .x,y,z:T init x:=y:=z:=E
```

If ":T" is absent (from a constant declaration), as in "const a=5", the type is that of the expression to the right of the equal sign.

A final clause, if present, is executed just before deallocation of the variables or constants with which it is associated. Deallocation is always in reverse of the (possibly unspecified) order of creation.

Objects may be designated "as specified" only in form implementations (section 5.9). This indicates that the <type description> is to be copied from the form specifications.

An <init fin clause> in a constant declaration can be omitted only in the specifications part of a form (see section 5.9).

5.6. Evaluation of Type Descriptions

<type description>s are syntactic entities which appear in object declarations and formal parameter specifications.

5.6.1 Syntax

<type description>	::=	<simple invocation> { ({<formal qual>}$^{+}_{,}$) }* {<update set>}* | ?<identifier>{<update set>}*
<formal qual>	::=	<expression> | ?<identifier>{<update set>}* | {<identifier>:}* <type description> | unk
<update set>	::=	< { <identifier> | <special identifier> }$^{+}_{,}$ >

Note that the outer <>'s in the definition of <update set> are part of the language, not metabrackets (see examples below).

5.6.2 Examples

```
integer
vector(real,1,10)
stack(T:form<&:=>,22)
collection(unk)
queue(process,?length)
```

5.6.3 Semantics

The <simple invocation> (see 4.3) must designate a unique form ("$" qualification allows duplicate nested form definitions). Disambiguation on the basis of argument types is not performed.

Elaboration of a <type description>, $T(e_1,\ldots,e_n)\langle p_1,\ldots,p_m\rangle$, proceeds as follows[13] :

1. The e_i are elaborated in an undefined order. The results of this elaboration are objects, routines, and types. The following special cases should be noted:

   a. The elaboration of unk is unk.

   b. The elaboration of ?identifiers implies an implicit binding; it is illegal if this is not in the context of a formal/actual parameter matching.

2. The number of formal parameters of T must be n ($n \geq 0$). The actuals must *match* the formals of T (see section 2.4). It is an error if they do not.

3. Each object actual is handled as in section 4.3.3.

4. The sequence of routines, types, objects, and unk markers produced by the preceding becomes the Qual property of the result type. T is the Base type (see section 2.2). The p_i become the update set.

The <update set> defines the update set of the type description (see 2.2). The listed identifiers must be names of routines declared with the base type. Only these routines and routines with no visible effects may be applied to the object within the scope covered by the declaration in which the type description appears. In the case that the update set is attached to a ?identifier or a <type formal>, the listed identifiers must be further specified in an "assumes clause" (see 5.12) *unless* they

[13]Note that an implementation may require compile-time elaboration of those type descriptions used as formal parameter specifications.

are <special identifier>s such as "&=". The assumptions about <special identifier>s are uniform and are included in appendix C. If no <update set> is given it is assumed to contain the full update set of the type; the <update set> "<>" denote the empty set.

5.7. Formal Parameters

Certain entities -- forms and routines -- can be parameterized. ***Formal parameters*** specify these parameterizations. The process of determining whether a given sequence of actuals conforms to the sequence of formal parameters is known as ***matching*** or ***type checking***. This process binds actual parameters to the corresponding formal parameters. It may also cause certain ***implicit bindings*** of identifiers marked with a "?" lexeme in the formals; these identifiers are ***implicit parameters*** with the same scope as ordinary formals. The implicit bindings are in effect whenever the explicit bindings of ordinary formals are.

The scope of a formal parameter to a routine is the text of the parameterized declaration, in the "restricted Algol scope" sense. That is, the scope of a formal does not include routine or form declarations within the parameterized routine text.

The scope of a formal parameter to a form is ***at most*** the text of the parameterized form declaration: it is also subject to "restricted Algol scope". In addition, unless explicitly redeclared in the impl (specifically, redeclared as specified), the scope of form formals is limited to the specifications and object declarations (including init clauses) of the form other than shared objects.

5.7.1 Syntax

```
<formals>           ::=    ( {<routine formal> | <binding><obj formal> | <type formal>},+ )
<routine formal>    ::=    <formal id list> proc <parms>  |  <formal id list> {vproc | func | sel}<v parms>
<binding>           ::=    { copy  |  { alias}*{ const | var} }*
<formal id list>    ::=    <identifier list> :
<obj formal>        ::=    <formal id list> <type description>
<type formal>       ::=    <formal id list> { form | pform }{<update set>}*
```

5.7.2 Examples

```
(const x:T1, var q:?T2, copy r:vector(?T2,1,n))
(T:form, h:proc(x:real):int)
```

5.7.3 Semantics

Formal parameters give the specifications of allowable actual parameters and provide local names for these parameters. A <routine formal> indicates a parameter position to be filled with a procedure or selector. A <type formal> indicates a position to be filled by a type description. An <obj formal> indicates a position to be filled by an object. The association of actual parameters to formals is determined positionally.

The specification of an object parameter may be preceded by a qualifier that controls the binding of actual to formal; the possible qualifiers are copy, const, var, and alias. The qualifiers const and var denote "by reference" parameters; in both cases the parameter name is bound to the actual parameter object. const parameters (like names declared in const object declarations), have an empty update set. As specified in section 4.3, the qualifier copy indicates that a local object is to be instantiated and initialized from the actual by copying using the &:= operation defined for the type. The update set of copied parameters is set to empty, hence they may be used only for input. The qualifier alias has no semantic effect. It indicates that a reference parameter may overlap other reference parameters; unless the alias qualifier is present, overlapping reference parameters are prohibited.

For type formals the actual type description must be a full type if the formal is specified as a form. It may be a full or partial type if the formal is specified as a pform.

If no ⟨binding⟩ is specified, const is assumed. It should be noted that for routine parameters not qualified by alias, the compiler assumes that the semantics of const and copy are identical[14] -- hence the compiler is free to copy const parameters if it seems desirable to do so. This statement is *not* true for forms or in the presence of alias, and the optimization may not be performed in those cases.

The notation

```
        a,b,c:T
```

is short for

```
        a:T, b:T, c:T
```

Identifiers preceded by "?" are implicit formals. One binding is established for the identifier during matching, no matter how often it appears. It is an error if it is not possible to establish such a binding. All instances of the identifier inside a given ⟨formals⟩ list must be preceded by "?".

5.8. Routine Declarations

Routines encapsulate computations. A routine (proc, vproc, func) may or may not return a result object. If it does, the update set of the compiler-generated name bound to the returned object is always null; there can be no effects on a procedure result. A selector always has a result and must not have effects; unless explicitly restricted, the ⟨update set⟩ of a selector result is the full update set of the base type. Except within form specifications (section 5.9), routine declarations have normal Algol scope.

5.8.1 Syntax

```
<routine decl>     ::=   <vproc decl> | <proc decl> | <sel decl>
<vproc decl>       ::=   <aux> {inline}* {vproc | func}<routine id> <v parms> <pre post> <assumes>
                         {<routine body>}*
<proc decl>        ::=   <aux> {inline}* proc <routine id> <parms> <pre post> <assumes> {<routine body>}*
<sel decl>         ::=   <aux> {inline}* sel<routine id><v parms> <pre post> <assumes> {<routine body>}*
<routine id>       ::=   <identifier> | <special identifier>
<parms>            ::=   {<formals>}*
<v parms>          ::=   {<formals>}*:<type description>
<pre>              ::=   {pre <assertion>;}*
<pre post>         ::=   <pre> {post <assertion>;}*
<routine body>     ::=   is <stmt> | is as specified | is forward | is external (<system specs>)
```

14 This assumption is valid only so long as user-defined assignment operators, "&:=", preserve the intended meaning. The compiler cannot enforce the correctness of *any* user-defined operation, and specifically not that of "&:=". Thus there is an addition [illegible]

5.8.2 *Examples*

```
vproc f(x:int):real
    pre {abs(x)<maxintreal};
    is float(x);
inline sel triang(var A:vector(?T,1,?n), i,j:int):T
    is A[i*(i-1) div 2+j]
proc empty is as specified
```

5.8.3 Semantics

Selectors (declared sel) name objects. Procedures (declared proc) produce effects but do not return values. Value-returning procedures (declared vproc), may produce effects and also return values (actually objects). Functions (declared func) are semantically equivalent to vproc's except that they are deterministic[15] and do not have observable effects on their parameters. The <stmt> portion of the <routine body> of a vproc or func must be a single expression of the type returned by the routine.

The qualifier inline has no semantic effect. It indicates that the compiler should make the declared routine "open" -- i.e., produce a copy of the object code at each invocation site.

The pre and post clauses have no semantic effect, but are specifications of the routine's behavior. The pre clause gives conditions which will be true at entry; post gives conditions at routine exit (the keyword result is conventionally used in post conditions to specify the value returned).

The routine body may be absent only in form specifications (see 5.9). It may be given "as specified" only in form implementations; this indicates that the body is to be carried down from the specifications. The routine body may be specified as forward if its declaration appears later in the same <block>. The body may be specified as external if the text of the definition is to be found in the system-dependent entity specified by <system specs>.

The formal parameter lists and result types must be omitted in form implementations if the same routine (name) is declared in the specifications of the form. They are copied from the declaration of the routine in the specifications.

The assumes clause (see section 5.12) declares generic parameters (implicit and explicit). Throughout the text of the routine declaration, only those properties declared in the assumes clause are used in testing syntactic and semantic validity.

As stated previously, identifiers that name objects observe restricted Algol scope. Thus, the body of a routine cannot access objects declared outside itself unless they are passed through the parameter list. Also, as stated previously, the objects named by distinct formal parameter names cannot overlap unless they are explicitly qualified with alias.

[15] That is, invocations with equal inputs yield equal outputs. More precisely, for F to be a func, &=(A,B) must imply &=(F(A),F(B)). If &= is not defined, invocations with identical inputs must have identical outputs.

Forms define classes of types. There is one form declaration for each base type. Identifiers declared in form declarations have normal Algol scope.

5.9.1 Syntax

```
<form decl>          ::=  <aux> form <identifier>{<formals>}* <pre><assumes> is <form body>
<form body>          ::=  {<specs>}*{<impl>}* end {<identifier>}*  |  <abbrev body>  |  forward  |  external
                          (<system specs>)  |  as specified
<specs>              ::=  specs { <var decl>  |  <other form decls> };+
<impl>               ::=  impl { <shared> <var decl> | <other form decls> };+
<other form decls>   ::=  <routine decl> | <form decl> | <axiom>| <shared> <const decl>
<shared>             ::=  shared*
<axiom>              ::=  invariant <assertion>  |  initially <assertion>  |  axiom <assertion>  |  repmap
                          <assertion>  |  rule <identifier> <assertion>
```

5.9.2 Examples

```
form F(T:form, x:int) is
    specs
        var m:int;
        vproc p(f:F):T pre {m<x} post {m>x}; ...
    impl
        const x:as specified
        var m:as specified;
        vproc p is F.m:=F.x+1; ...
    end
```

5.9.3 Semantics

The names defined in the specifications (<specs>) of a form are available outside the form declaration. The scope of these names is the same as if they had been declared immediately outside the form, except that var and const declarations become routine declarations as described below. The scope of the names in the specifications does not include the implementation (<impl>). Note, however, that all these names must be redeclared in the implementation. The implementation may be omitted if the form declaration appears as part of the specifications of another form. The specification may be omitted if the declaration appears in the implementation of another form, in which case it is copied from the specifications of the containing form.

The scope of object names appearing in the formal parameter list of the form is restricted to the specs and <var decls> of the <form body> unless they are explicitly redeclared (as specified) in the impl. In the latter case a run-time representation of these objects becomes part of the implementation of objects instantiated from the form. In such cases it is also possible for these names to be mentioned (again as specified) in the specs, and hence to be externally available. These redeclarations in the impl and specs must be compatible with the <binding> of the formal parameter; that is, an identifier redeclared var must also have a var <binding> (an identifier redeclared const may have a var, const, or copy <binding>).

Objects declared in a form usually become the concrete components of the objects that result from instantiating the form; there are thus distinct instantiations of these objects for different instantiations of the form. An exception occurs when a declaration is prefixed by by the modifier shared: A single instantiation of a shared object is common to all instantiations of the form. In particular, it makes perfect sense to define a shared const of a given type within the definition of that type. Such a constant functions as a named literal of the type.

<Axiom>s have no semantic effect, but provide further specifications.

Non-shared constant and variable declarations within form specifications are shorthand for certain procedure and selector declarations. That is,

```
    form T...
        specs...
        var P:Q
        const A:R
```

is short for

```
    form T...
        specs...
        sel P(var t:T):Q
        func A(t:T):R
```

In the implementation, object declarations again become selector and procedure declarations as follows. When an object of base type T is instantiated, the object declarations in its implementation are elaborated as usual (and the init clauses are performed). This results in a set of newly created objects which become the concrete components of the object being created. These components may be accessed within the bodies of the routines in the implementation by using implicit selectors (procedures) with the same names as those given in the object declarations. Thus, we can write

```
    form T...
        impl
            var x:int;
            proc f(Q:T) is ... Q.x ...
```

That is, inside f, "x" is treated as a selector on objects of type T and is applied to the formal parameter, Q, to access the x-component of the particular actual. Note in particular that "x", like all object names, obeys restricted Algol scope and hence is not inherited by the body of the proc, f.

5.10. Abbreviations

Abbreviated form body definitions are provided for two commonly occurring kinds of abstractions, "records" and ordered "enumerated" types.

5.10.1 Syntax

```
<abbrev body>       ::=   <record type> | <enumerated type>
<record type>       ::=   record ({<obj decl group>},⁺)
<enumerated type>   ::=   enumerated ( <identifier list> )
```

5.10.2 Examples

```
    record (re,im:real)
    record (x,y:int, load:real, theta:radians)
    enumerated ( red, blue, green, purple, bardot )
```

5.10.3 Semantics

The declaration

```
    form F(...) is record (d_1,d_2,...,d_k)
```

is semantically equivalent to

```
    form F(...) is
        specs
            var d_1, ..., d_k;
            func cons(d_1, ..., d_k): F(...);
            func &=(lhs,rhs:F): boolean;
            vproc &:=(var lhs,rhs:F): F(...);
        impl
            var d_1, ..., d_k;
            func cons is
                value v:F(...)
                    of note assign to components of v eton fo
            func &= is note compare components for equality eton
            vproc &:= is note assign rhs to lhs component-by-component eton
        end F
```

In addition, all parameters of F are converted to "?identifiers" when they appear in formal parameter lists of "cons", "&=", or "&:=".

The declaration

```
    form C is enumerated (i_1,...,i_n)
```

is semantically equivalent to

```
    form C is
        specs
            shared const i_1, ..., i_n:C;   ! distinct constants;
            func &= ...;                    ! equality test
            vproc &:= ...;                  ! assignment
            func min ...;                   ! minimum element (=i_1)
            func max ...;                   ! maximum element (=i_n)
            func succ ...;                  ! successor (not defined on i_n)
            func pred ...;                  ! predecessor (not defined on i_1)
            func card ...;                  ! cardinality of enumeration (=n)
            func decode ...;                ! converts element to its ordinal (e.g., decode(i_3)=3)
            func code ...;                  ! converts ordinal to element (e.g., code(2)=i_2)
            func spell ...;                 ! convert element to stringlet (its printname)
            func unspell ...;               ! convert stringlet (printname) to element
            generator gen ...;              ! generates elements in order (i_1 ... i_n)
        end C
```

5.11. Generators

Generators are specialized forms. They are useful for defining objects that will be bound to the control variables of the for and first constructs (see 4.7, 4.8). (Any form may provide such objects, but their use is sufficiently constrained by the language that special abbreviated syntax is provided for defining forms intended specifically for this purpose.)

5.11.1 Syntax

```
<form decl>          ::=     <aux> generator <identifier> {<formals>}* : <type description> <pre> <assumes>
                             is <form body>
```

5.11.2 Example

```
generator upto(lb,ub:int): int is
    specs
        pre {ub≤maxint-1 ∧ lb≥minint}
        aux var k:int=ub+1;
        rule for
            {premise l≤k≤u ∧ I([l..k-1]) {ST(k)} I([l..k])
             concl I([]) {for k from g: upto(l,u) do ST(k) od} I([l..u])}
        rule first
            {premise P ∧ l≤k≤u ∧ (∀w)(l≤w<k ⊃ ¬β(w)) ∧ β(k) {S1(k)} Q
             premise P ∧ (∀w)(l≤w≤u ⊃ ¬β(w) {S2} Q
             concl P {first k from g: upto(l,u) suchthat β(k) then S1(k) else S2 fi} Q}
    impl
        var k: as specified;
        const lb,up: as specified;
        func &done is g.k>g.ub;
        sel &value is g.k;
        proc &start is g.k:=g.lb;
        proc &next is g.k+:=1;
        proc &finish is g.k:=g.ub+1;
    endof upto
```

Note that in this example we have chosen to ignore statement and predicate parameters other than k.

5.11.3 Semantics

In order to generate loop control variables, a form must provide definitions for the routines &start, &next, &done and &value. A definition of the routine &finish is optional. If no definition is provided for &finish, the compiler will provide (1) an appropriate header, (2) the body skip in the specifications, and (3) the body as specified in the implementation. Restrictions on the definitions of &done, &start, &next, &finish, and &value are provided in appendix C.

A distinguished class of forms defining objects for controlling loops is designated by the reserved word generator and the syntax indicated above. This class is significant because the behavior of these objects is sufficiently constrained to be specified by a proof rule. Necessary properties of the specifications of the generator routines can be derived from the proof rules and the constraints[16]. As a result, these specifications are not written explicitly. Instead, proof rules for first and for loops that use the generator are written as shown above. An instantiation of a generator may be used only to control loop constructs for which it provides proof rules.

---

16 See "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators" by Mary Shaw, Wm. A. Wulf, and Ralph L. London, in *Communications of the ACM*, August 1977, pp. 553-564.

5.12. Assumptions

Assumptions provide "skeleton" declarations for generic parameters and provide sufficient information to verify all uses of these parameters locall.y

5.12.1 Syntax

```
<assumes>          ::=     {assumes <form decl>};*
```

5.12.2 Examples

```
    assumes form T is specs func &=(a,b:T):boolean ...
```

5.12.3 Semantics

Assumed forms may not be parameterized and may not have implementations. Identifiers appearing in the update sets of generic parameters must be specified in an assumes clause (see 5.6). Actual parameters corresponding to generic formals must syntactically satisfy the assumptions about the formal (see 2.4).

The following routine illustrates the use of assumptions and generic parameters:

```
    func equal'vectors(x,y:vector(?T,?lb,?ub)):boolean
        assumes form T is specs func &=(a,b:T):boolean end
        is   first i from upto(lb,ub) suchthat x[i]≠y[i] then false else true fi
```

This routine will determine whether two vectors, x and y, are equal so long as (1) the type of the elements of the vectors, T, provides an equality operator and (2) the two vectors have the same upper and lower bounds. An assumes declaration behaves as though it had normal Algol scope; in particular, assumptions about generic parameters of a form need not be repeated within routines (or other forms) declared within the form.

Appendix A
Collected Syntax

```
<letter>               ::=   A | B |...| Z | a |...| z
<digit>                ::=   0 | 1 |...| 9
<alphanumeric>         ::=   <letter> | <digit> | '
<special symbol>       ::=   <basic symbol> | <operator>
<basic symbol>         ::=   begin | end | endof | ; | : | ( | ) | $ | , | :: | & |
                             if | then | else | fi | case |
                             of | esac | fo | with | in | ni | first | suchthat | from | do | od |
                             for | exitloop | leave | skip | assert |
                             var | const | aux | as specified | = | init |
                             final | unk | ? | proc | vproc | func | sel | label |
                             note | eton | ! | elif | elof | pform | while | > | < | . | # |
                             copy | alias | form | inline | pre | post | rule |
                             is | forward | external | specs | impl | shared |
                             invariant | initially | axiom | repmap |
                             record | enumerated | assumes | value | generator
<operator>             ::=   <binary operator> | <unary operator>
<binary operator>      ::=   ↑ | * | / | div | rem | + | - | < | ≤ | = | <> | ≥ | > | and | or | cand | cor | imp |
                             <assign op>
<unary operator>       ::=   + | - | not

<identifier>           ::=   <letter> {<alphanumeric>}*
<special identifier>   ::=   &start | &finish | &next | &done | &value | &subscript | &<operator>
<identifier list>      ::=   <identifier>,+

<special literal>      ::=   <unsigned integer> | <unsigned real> | <string> | <boolean> | <radix>
<unsigned integer>     ::=   {<digit>}+
<unsigned real>        ::=   <unsigned rational>{E<scale-factor>}* | <unsigned integer>E<scale-factor>
<unsigned rational>    ::=   <unsigned integer>.<unsigned integer>
<scale-factor>         ::=   {+|-}*<unsigned integer>
<string>               ::=   "<any sequence of characters with all quotes doubled>"
<boolean>              ::=   true | false
<radix>                ::=   {<alphanumeric>}+#<alphanumeric>

<compilation unit>     ::=   begin <block> end | <exec decl list>
<block>                ::=   {<exec decl list>;}* {<stmt>};+ {;}*
<exec decl list>       ::=   { <exec decl> };*
<exec decl>            ::=   <var decl> | <const decl> | <proc decl> | <form decl> | <label decl>

<expression>           ::=   <invocation> | <conditional expression> | <value expression> |
                             <with expression> | <first expression>
<stmt>                 ::=   <expression> | <loop stmt> | <exit stmt> | <null stmt> |
                             <inner block> | <labeled stmt> | <assert stmt>
<labeled stmt>         ::=   <identifier> : <stmt>

<invocation>           ::=   <special literal> | <simple invocation>{<actuals>}* | (<invocation>)
<actuals>              ::=   ({<actual>},*)
<actual>               ::=   <expression> | <type description>
<simple invocation>    ::=   {<identifier>$ }* <identifier> | <special identifier>
```

```
<conditional expression>  ::= <if expression> | <case expression>
<if expression>      ::=   if <expression> then <block> { elif <expression> then <block> }* {else <block>}* fi
<case expression>    ::=   case <expression> of <case> { elof <case> }* { else <block> }* esac
<case>               ::=   {<expression>},+ :: <block>

<value expression>   ::=   value <identifier>{:<obj type>}* of <block> fo

<with expression>    ::=   with <with list> in <block> ni
<with list>          ::=   { <identifier>:<invocation> },+

<first expression>   ::=   first<template> suchthat <expression> {then<block>}* {else <block>}* fi
<template>           ::=   <identifier> from {<identifier>:}*<type description> | <identifier> from <invocation>

<loop stmt>          ::=   <simple loop> | <while stmt> | <for stmt>
<simple loop>        ::=   do <block> od
<while stmt>         ::=   while <expression> <simple loop>
<for stmt>           ::=   for <template> <simple loop>

<exit stmt>          ::=   exitloop | leave <identifier>

<null stmt>          ::=   skip

<inner block>        ::=   begin <block> end | begin <block> endof {<identifier>}*

<assert stmt>        ::=   assert <assertion>

<label decl>         ::=   label <identifier list>

<var decl>           ::=   <aux> var {<obj decl group>{<init fin clause>}*},+
<const decl>         ::=   <aux> const {<obj decl group> {<init fin clause>}* | <const assign>},+
<aux>                ::=   {aux}*
<obj decl group>     ::=   <identifier list> : <obj type>
<obj type>           ::=   <type description> | as specified
<init fin clause>    ::=   = <expression> | {init<stmt>}* {final <stmt>}*
<const assign>       ::=   <identifier list> = <expression>

<type description>   ::=   <simple invocation> { ( {<formal qual>},+ ) }* {<update set>}* |
                           ?<identifier>{<update set>}*
<formal qual>        ::=   <expression> | ?<identifier>{<update set>}* | {<identifier>:}* <type description> |
                           unk
<update set>         ::=   < { <identifier> | <special identifier> },+ >

<formals>            ::=   ( {<routine formal> | <binding><obj formal> | <type formal>},+ )
<routine formal>     ::=   <formal id list> proc <parms> | <formal id list> {vproc | func | sel}<v parms>
<binding>            ::=   { copy | { alias}*{ const | var} }*
<formal id list>     ::=   <identifier list> :
<obj formal>         ::=   <formal id list> <type description>
<type formal>        ::=   <formal id list> { form | pform }{<update set>}*
```

```
<routine decl>        ::=   <vproc decl> | <proc decl> | <sel decl>
<vproc decl>          ::=   <aux> {inline}* {vproc | func}<routine id> <v parms> <pre post> <assumes>
                            {<routine body>}*
<proc decl>           ::=   <aux> {inline}* proc <routine id> <parms> <pre post> <assumes> {<routine body>}*
<sel decl>            ::=   <aux> {inline}* sel<routine id><v parms> <pre post> <assumes> {<routine body>}*
<routine id>          ::=   <identifier> | <special identifier>
<parms>               ::=   {<formals>}*
<v parms>             ::=   {<formals>}*:<type description>
<pre>                 ::=   {pre <assertion>;}*
<pre post>            ::=   <pre> {post <assertion>;}*
<routine body>        ::=   is <stmt> | is as specified | is forward | is external (<system specs>)

<form decl>           ::=   <aux> form <identifier>{<formals>}* <pre><assumes> is <form body>
<form body>           ::=   {<specs>}*{<impl>}* end {<identifier>}* | <abbrev body> | forward | external
                            (<system specs>) | as specified
<specs>               ::=   specs { <var decl> | <other form decls> };+
<impl>                ::=   impl { <shared> <var decl> | <other form decls> };+
<other form decls>    ::=   <routine decl> | <form decl> | <axiom>| <shared> <const decl>
<shared>              ::=   shared*
<axiom>               ::=   invariant <assertion> | initially <assertion> | axiom <assertion> | repmap
                            <assertion> | rule <identifier> <assertion>

<abbrev body>         ::=   <record type> | <enumerated type>
<record type>         ::=    record ({<obj decl group>},+)
<enumerated type>     ::=    enumerated ( <identifier list> )

<form decl>           ::=   <aux> generator <identifier> {<formals>}* : <type description> <pre> <assumes>
                            is <form body>

<assumes>             ::=   {assumes <form decl>},*
```

Appendix B
Standard Prelude

The complete specifications of the types in the standard prelude will appear in a future technical report. This appendix previews that report by listing the types to be included and summarizing their main properties. This, together with the reader's experience and goodwill, should suffice to understand the language definition.

The type definitions loosely referred to as the "standard prelude" actually comprise three classes of definitions. The ***primitive prelude*** defines operationally the basic notion of linear contiguous storage and all of the common machine operations. The ***standard prelude*** proper includes types that we expect to be implemented for all versions of the language, usually as a part of the compiler. The ***implementation prelude*** includes types that provide facilities of the underlying hardware, operating system, or support environment of a particular implementation. In addition, types (such as real) which should be defined for a large class of systems (but not all) are defined here. It is our intention that whenever a particular implementation chooses to define a type given here, it follows our specifications to the greatest extent possible.

B.1. Primitive Prelude

The ***RawStorage*** form supplies the fundamental abstraction of linear contiguous storage as described in Chapter 1. Fetching and storing are defined for ***RawStorage*** objects of equal length. The integer and bitwise boolean operations are provided for ***RawStorage*** objects of length one. In addition we provide support here for the subsequent definition of forms such as collections and references, as well as the ability to do storage management.

B.2. Standard Prelude

These types were chosen for their simplicity and common utility. They are intended to provide only primitive facilities that may reasonably be expected to appear (and be efficiently implementable) on all systems. The reader must bear in mind that these specifications were selected with the understanding that they become essentially ***required*** of all implementations. When in doubt, therefore, we have tended to exclude features rather than to include them.

There are three major classes of types in the standard prelude. The sections below sketch the major properties of each.

B.2.1 Scalars

The scalar types of the standard prelude are ***Boolean*** and ***Integer***. Literals of these types are defined in section 3.5.

Boolean is the other form required by the language definition (Chapter 1). Objects of type ***Boolean*** are unstructured; they possess values from a set designated {true, false}. The customary unary and binary functions are provided, along with assignment.

Integers are restricted to machine precision. The standard prelude supports the customary unary and binary arithmetic operators on integers, the arithmetic relations, named constants to describe the finite range of a particular implementation, assignment, and transfer functions to and from ***Stringlets***.

B.2.2 Linear Structures

Two linear, homogeneous, fixed-length data structures are provided in the standard prelude. ***Vectors*** may have elements of many types; ***Stringlets*** are minimally sufficient to support I/O operations.

Vectors may have elements of almost any type (the type must be allocatable without special restrictions). Any particular ***Vector***, of course, contains elements of only one type. The length of a ***Vector*** is fixed at instantiation time. A subscript selector for integer indices and a generator that produces the elements in order are provided. If the element type supports either assignment or equality test, that operation is extended to the entire ***Vector***. There is a slice routine which returns a subvector of the original vector (with the new origin forced to zero).

Stringlets closely resemble vectors of characters. The same operations are provided for ***Stringlets*** as for ***Vectors***. In addition, literals are supported by the syntax of section 3.5 and an assortment of special predicates on ***Stringlets*** of length 1 is provided (IsCharacter, IsLetter, IsDigit, etc.). Transfer functions to and from ***Integers*** are also provided. Note that type "character" is not included in the standard prelude; use ***Stringlets*** of length 1 instead.

B.3. Implementation Prelude

Certain other types appear in some fashion in almost every language. These types do not have uniform specifications across implementations, but rather depend on the host machine. Since we don't want to require these for all systems, we cannot include them in the standard prelude. Instead, such types are provided in an ***Implementation prelude*** -- a segment of the definition of each implementation that is frankly machine-dependent. In addition, certain types which may not exist under all implementations, but which must have uniform specifications for those systems on which they do exist, are included here.

The types defined here may provide direct access to features of the underlying hardware; they may support special facilities of the environment or the operating system; they may simply be data types that are best supported directly by the compiler.

B.3.1 Scalers

Reals have the properties of hardware floating-point values. Unary and binary arithmetic operators and relations are supported to the extent that the underlying machine allows. Constants describing floating point accuracy, assignment, and transfer functions to and from ***Integers*** and ***Stringlets*** are also provided. There is ***NO*** mixed-mode arithmetic.

B.3.2 Input and Output

The implementation prelude requires a minimal set of operations on files of characters. This, together with transfer functions to and from ***Stringlets***, is intended to guarantee the availability of at least primitive input/output facilities. It is intended that each implementation prelude provide such richer support as is appropriate, provided such support is an upward compatible extension of the facilities described here.

Form ***IOFile*** supports sequential files of characters. ***Files*** are sequences of ***stringlets*** of length one. They are constructed by appending ***Stringlets*** and decomposed into ***Stringlets***. The available operations are commands to open or close an ***File*** for reading or writing, to test the status of an ***File***, and to read or write a ***Stringlet*** from or to an ***File***. Note that ***File*** is an auxilliary form definition--such objects cannot be declared. ***IOFile***, on the other hand, has a null representation. Hence it is not meaningful to declare objects of this type.

B.3.3 Machine Dependent Types

These types may provide direct access to features of the underlying hardware or to special facilities of the environment or the operating system.

Appendix C
Special Identifier Assumptions

A certain number of Alphard routines are used for special purposes. Their definitions are thus more constrained than has been indicated elsewhere in this report. These routines are distinguished in that their names begin with ampersand ("&"); syntactically, they are <special identifier>s. The routines are invoked by specific language constructs rather than by ordinary routine invocation.

These routines are grouped into two classes. ***Generator routines*** are invoked by the for and first iteration constructs in the manner described in sections 4.7 and 4.8. ***Extensible operators*** are affected by the infix operator rewrite rules as well as by the subscript rewrite rule (see section 3.4). Extensible operators include all operators named by <special identifier>s other than generator routines; they can be overloaded by user-defined forms.

C.1. Generator routines

The generator routines are &done, &start, &next, &finish and &value.

C.1.1 Use restrictions

The restrictions on the invocation of generator routines as an ordinary routines are intended to support two conflicting requirements. First, it is mandatory that a for or first loop body not be able to use these functions on the object generating the loop control variable. It is, in general, desirable that they not be invoked in other arbitrary places outside of loops either. It is, however, useful to be able to use these routines in the definitions of other generators, specifically those which simultaneously generate elements of two or more data structures.

The invocation restriction is thus that: &start, &next and &finish can only be invoked in the definition of an &start, &next or &finish routine of another generator; &done can only be invoked in the definition of an &start, &next, &finish or &done routine in another generator; &value can only be invoked within another generator. This would seem to permit use of the same generator object to create control variables in two nested loops. Any problems which this might cause for the generator object would be detected as violations of sufficient independence of the object from the body of the outer loop.

C.1.2 Definition restrictions

Suppose a generator named gen defines a type of object that provides a control variable of type t for the "<identifier> from" construct. The generator routines must have header specifications subsuming:

```
func &done(g:gen):bool;
sel &value(g:gen):t;
proc &start(g:gen);
proc &next(g:gen);
proc &finish(g:gen);
```

C.2. Extensible routines

The extensible routines are &↑, &*, &/, &div, &rem, &+(unary and binary), &-(unary and binary), &<, &=, &>, &and, &or, &imp, &not, &:= and &subscript.

C.2.1 Use restrictions

Extensible routines can be invoked either by using the infix operator rewrite rule or by using the subscript rewrite rule (see 3.4). There is no loss of generality in prohibiting their invocation as ordinary routines.

C.2.2 Definition restrictions

Within form f, specifications of these operators must subsume:

```
func &↑(leftparm:f,rightparm:t):t1;
func &*(leftparm,rightparm:f):f;
func &/(leftparm,rightparm:f):f;
func &div(leftparm,rightparm:f):t;
func &rem(leftparm,rightparm:f):t;
func &+(leftparm,rightparm:f):f;
func &+(parm:f):f;
func &-(leftparm,rightparm:f):f;
func &-(parm:f):f;
func &<(leftparm,rightparm:f):bool;
func &=(leftparm,rightparm:f):bool;
func &>(leftparm,rightparm:f):bool;
vproc &:=(alias var leftparm:f, alias rightparm:f):f;
sel &subscript(afterdot:f,p1:t1, .. ,pn:tn):t;
```

The compiler is able to enforce only the precedence and syntax of these operators. However, their traditional use in mathematics raises other expectations about them; most people, for example, presume "+" is at least associative and possibly commutative. We strongly urge that the programmer overload these operators only with operations that preserve those expectations; failure to observe this convention may badly mislead the reader.

Appendix D
A Complete Example

We now present a complete Alphard program. This program defines ***finite sets*** with a fixed maximum size, then uses them in a small program. Several aspects of the program deserve special note.

Form FinSet defines one variety of sets. These sets must be homogeneous (i.e., all elements must be of the same type), but the elements may be of any type that provides assignment and equality operators. Thus FinSet is a ***generic*** type definition. The specifications of FinSet are stated in terms of ordinary mathematical sets; the restrictions that apply to sets of type FinSet are explicit.

We assume the types of the standard prelude, Appendix B. In particular, we use vectors, integers, and the generator upto for integers. The main program defines an ordered enumerated type and uses the generator that is automatically defined for such a type.

The names V and m used in the implementation of FinSet are available to the bodies of the routines defined in that form. V and m may be used as qualifiers on any objects of type FinSet that those routines receive as parameters. The names V and m are ***not*** available outside the form.

FinSet defines routines &+(union), &*(intersect), &:=(assign), and &=(equality). These extend the definitions of the binary infix operators +, *, :=, and = to pairs of FinSets. The FinSet implementation also takes advantage of the rewrite rule for +:= and -:=.

Assertions are included in the specifications of the FinSet operators. (We use "prime" notation in post conditions: S' is the value that S had on entry to the procedure.) Some have also been included in the main program to explain the operation of the program.

Three kinds of loops are used. Routines Insert, Remove, and Has all use first loops to search for elements. Insert defaults the then part and Remove defaults the else part. Note that the equality test in the suchthat clauses of these loops uses the equality defined for EltType. As a result, the code for these routines may depend on the definition of the type passed as an instantiation parameter to FinSet. The main program declares three sets -- one is a set of colors and the other two are sets of sets (of colors). The program uses for loops with the generator color$gen on colors. It also uses a conventional while.

The '$' name qualification is used for names color$card and color$gen. These are constants defined for the ordered enumerated type color (number of elements and a standard generator, respectively). They are associated with the type rather than with any variable of the type. The qualification is used to distinguish ord and gen from the corresponding definitions associated with other enumerated types.

```
begin

form FinSet(EltType:form<&:=,&=>,MaxSize:integer)
pre { MaxSize ≥ 0 }
is specs
     aux var FS:MathematicalSet(EltType)
     invariant { cardinality(FS) ≤ MaxSize }
     initially { FS={ } }
     proc  Insert(var S:FinSet,x:EltType)      pre  { cardinality({x} ∪ S.FS) ≤ S.MaxSize }
                                               post { S.FS=S.FS' ∪ {x} }
     proc  Remove(var S:FinSet,x:EltType)      post { S.FS = S.FS' - {x} }
     vproc  Choose(S:Finset):EltType           pre  { S.FS = { } }
                                               post { result ∊ S.FS }
     func  Has(S:FinSet,x:EltType):boolean     post { result = x∊S.FS }
     func  &+(R,S:FinSet):Finset               pre  { cardinality(R.FS ∪ S.FS) ≤ R.MaxSize }
                                               post { result = R.FS' ∪ S.FS' }
     func  &*(R,S:FinSet):FinSet               post { result = R.FS' ∩ S.FS' }
     vproc &:=(var R,S:FinSet):FinSet          post { result = R.FS = S.FS' }
     func  &=(R,S:FinSet):boolean              post { result = (R.FS' = S.FS') }
     func EmptySet(Eltype:form,MaxSize:integer):FinSet(Eltype,MaxSize)
          post { result = { } }

impl
     var V:vector(EltType,1,MaxSize), m:integer init m:=0
     const MaxSize: as specified;
     repmap { FS = {V[i] | 1≤i≤m} }
     invariant {(0≤m≤MaxSize) ∧ (∀i,j∊[1..m](V[i]=V[j]⊃i=j)) }
     proc Insert
          is first p from upto(1,S.m) suchthat S.V[p]=x
               else S.m +:= 1; S.V[S.m]:=x fi
     proc Remove
          is first p from upto(1,S.m) suchthat S.V[p]=x
               then S.V[p]:=S.V[S.m]; S.m -:= 1 fi
     vproc Choose is S.V[1]
     func Has
          is first p from upto(1,S.m) suchthat S.V[p]=x
               then true else false fi
     func &+
          is value T:Finset(R.EltType,R.MaxSize) of
               T.m := R.m
               for j from upto(1,R.m) do T.V[j] := R.V[j] od
               for j from upto(1,S.m)do Insert(T,S.V[j])od
               fo
     func &*
          is value T:Finset(R.EltType,R.MaxSize) of
               for j from upto(1,R.m) do if Has(S,R.V[j])
               then Insert(T,R.V[j]) .fi od
               fo
     vproc &:=
          is value R of R.m := S.m; for i from upto(1,S.m) do R.V[i] := S.V[i] od fo
```

```
    func &=
         is R.m=S.m and first i from upto(1,R.m) suchthat not Has(S,R.V[i])
              then false else true fi
    func EmptySet
         is value x of skip fo
end

!   Compute powerset of enumerated type
form color is enumerated (red,orange,yellow,green,blue,violet) end
var ColorSet: FinSet(FinSet(color,color$card), 2↑color$card)
assert { ColorSet = { } }
Insert(ColorSet,EmptySet(FinSet(color,color$card)))
assert { ColorSet = { { } } }
for c from color$gen do
    var Temp: FinSet(FinSet(color,color$card), 2↑color$card)
    Temp := ColorSet
    while Temp ≠ EmptySet(FinSet(color,color$card)) do
         var Current: FinSet(color,color$card)
         Remove(Temp,Current:=Choose(Temp))
         Insert(Current,c); Insert(ColorSet,Current)
         od
    od
assert { ColorSet = { S | S ⊆ {x|x is a value of type ColorSet} } }
end
```

Appendix E
Proof Rules

Proof Rules Omitted From Preliminary Version