Modeling and Implementation of Visibility in Programming Languages

Phillip Edward Garrison

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

December 1987

Dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of
the University of California, Berkeley

Report No. UCB/CSD 88/400

This research was supported in part by a National Science Foundation Fellowship, by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871 (monitored by Naval Electronics System Command under Contract No. N00039-84-C-0089), and by a State of California Micro Fellowship.

Modeling and Implementation of Visibility in Programming Languages

Phillip Edward Garrison

Abstract

The term *visibility control* refers to the use of names in programming languages. A *binding* of a name and an entity results from a declaration. A binding is *visible* where the name can be used to reference the entity. The *visibility rules* of a language define how names may be used in that language, how to determine which declaration is denoted by a reference anywhere in a program, and the meaning of multiple declarations of the same name.

Existing approaches to modeling visibility rules are not powerful enough to model the wide variety of visibility features present in modern programming languages. The *Inheritance Graph Model*, presented in this dissertation, is a natural and general model of visibility that embodies the fundamental concepts of visibility. Because the Inheritance Graph Model is based on fundamental concepts, it is easy to represent the visibility structure of a wide variety of languages and visibility features in a straightforward manner. These fundamental concepts were developed on the basis of a comprehensive survey and analysis of the visibility features used in programming languages.

The visibility structure of a program is represented by an *inheritance graph*. A vertex in the graph represents a uniform referencing environment (a contour). Multiple *visibility classes* can be used to explicitly represent different kinds of visibility in the graph. An edge in the graph represents inheritance of visibility of bindings with a specific visibility class from one vertex to another. Inheritance of visibility of bindings via an edge can be restricted depending on attributes of each binding or on the program construct represented by the edge. A general mechanism for detecting errors and specifying shadowing of bindings declared in enclosing scopes is also provided. An implementation based on data flow analysis techniques can be generated automatically from a specification of the inheritance graph for a language.

The study of visibility features and their precise meanings exposed several subtle complexities in the visibility rules of popular languages. Among these is the requirement in Pascal that a declaration precede all references to the declaration. This requirement provides few benefits, and results in visibility rules that are difficult to understand, specify, and implement correctly.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like, first and foremost, to thank my wife, Vickie Blakeslee, for her patience, encouragement, and support throughout the development of the ideas presented in this dissertation. I would also like to thank my parents for their financial support, for always giving me the freedom to choose my own directions and goals, and for having confidence in my abilities to reach those goals.

Susan Graham was my research advisor for this dissertation. Her suggestions and insights have significantly improved the quality of this research.

Thanks are also due to Paul Hilfinger, who served on my dissertation committee and read my dissertation, and provided many useful comments on both the technical content and presentation of the dissertation. Lucien LeCam also served on my dissertation committee, for which I am grateful.

The following people provided assistance in the preparation of this dissertation, either by reading and commenting on parts or all of the dissertation or by participating in discussions on the topics discussed in the paper: Peter Kessler, Robert Henry, Kirk McKusick, and the members of the PIPER group at Berkeley – Eduardo Pelegrí-Llopart, Bob Ballance, Michael VanDeVanter, Chris Black, Dain Samples, Charles Farnum, Bill Maddox, and Jacob Butcher. Their assistance is gratefully acknowledged. Eduardo Pelegrí-Llopart and Bob Ballance deserve special thanks for their participation in many discussions with me.

This research was supported in part by a National Science Foundation Fellowship, by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871 (monitored by Naval Electronics System Command under Contract No. N00039-84-C-0089), and by a State of California Micro Fellowship.

CHAPTER 1

Introduction

The term *visibility control* refers to the use of names in programming languages. The noun "name" is used to denote many different things, but is used in this dissertation with a specific meaning. A *name* is a unique representation of an identifier, label, or other character string used in a program to denote some entity[1], such that all identifiers, etc., considered equivalent in a language map to the same name. The actual representation of names is unimportant.

A declaration is *visible* where the name introduced by the declaration can be used to denote the entity created by the declaration. The rules for visibility control in each language define:

- how names may be used in that language, including what constitutes a legal name.
- what any name means in any possible context.
- the significance of a declaration of an entity and its effect on the use of names.
- how entities may be named (*referenced*).

These rules are sometimes called *scoping rules*. To avoid any possible confusion of terminology later in the dissertation, we refer to these rules as *visibility control rules*, or simply *visibility rules*.

The visibility control rules are a significant part of a language. They affect the style in which a programmer writes a program, particularly the modularity of a program. They determine to some extent how difficult it will be to express a concept or problem solution in a particular programming language. Poorly-designed visibility rules can make programming clumsy and error-prone, while well-designed visibility rules make building a correct program easier.

A language designer must define the visibility control rules of his or her language, either formally or informally. A language implementor must have a good grasp of the language's visibility rules in order to implement them correctly. Overly complex visibility rules can make the task of both language designers and implementors much more difficult.

A good understanding of the fundamental concepts of visibility control would be of value to language designers, implementors, and users. However, very little work has been done with the goal of improving this understanding. Most efforts at description of visibility rules have been *ad hoc*, varying from one language to another and from one specification to another. Several different models of visibility control have been used, but most have been implicit, with little understanding of the ramifications of the use of each model. The primary goal of this dissertation is to improve this situation, presenting a better understanding of the fundamental concepts of visibility control.

Visibility rules in programming languages have perhaps not been studied much for several reasons:

- The visibility control issues in a language usually do not appear very complex, although closer examination would reveal otherwise.
- Programming language users can usually get by quite well with a simplified mental model of the visibility rules of a language, avoiding needing to know all of the ramifications and

[1] Precisely what is meant by "entity" will be defined later.

subtleties of the rules, or even that such subtleties exist. So, they often can ignore these issues (until a problem arises).

- Language designers are often more concerned with how various visibility control features can be used than with the underlying concepts or with ease of understanding. Language designers have also been known to place more emphasis on implementation details than clean language design, negatively affecting the visibility rules of their languages.

Although the visibility control rules of a language may seem simple, they often contain complexities and subtleties that only become apparent through close examination. Language features for visibility control that appear the same are often subtly different. Such differences may be easy to overlook, but significant enough to result in an illegal program, or worse, a legal program that produces unintended results. For each such language feature, there are probably as many different versions as there are languages that include the feature. Many of these subtleties were not discovered until this research was well underway.

Differences in a specific visibility control feature from language to language also make visibility rules much more difficult to specify, because it is difficult to define a set of primitives that will suffice for all languages. This usually results in high-level specification methods that are incomplete, or very low-level specification methods that provide little help for the specific problem of specifying visibility rules.

1.1. The Inheritance Graph Model

The *Inheritance Graph Model* was developed by the author to specify visibility control rules and improve the understanding of visibility control in programming languages. The Inheritance Graph Model is a natural and general model of visibility control that is useful for a wide variety of purposes. It permits the direct representation of the actual structure of visibility in a program, free from implementation or other details that might detract from the expression and understanding of the fundamental concepts of the visibility control features of a language. The greatest asset of the Inheritance Graph Model is its construction from a few very general basic concepts:

- name-entity *bindings*.
- *visibility regions*: a visibility region is an area of a program over which visibility is constant; it may or may not coincide with a "scope." A visibility region is represented by a vertex in an *inheritance graph*
- multiple *visibility classes*: used to distinguish different visibility of a binding for different kinds of references. The most common kind of reference is an ordinary reference in an expression. Another kind of reference occurs when processing a declaration in Pascal: one must look for another declaration of the same name in the same scope to determine if the declaration is legal. This reference is restricted to finding only declarations in the same scope.
- *inheritance* of visibility of a binding, with specified visibility classes, from one visibility region to another via *inheritance edges* in the inheritance graph.
- *clash resolution*: a general mechanism for specifying the result when two clashing bindings are inherited by the same visibility region. In most languages, two bindings clash if they have the same name, but the test may be more complex. Clash resolution is a general mechanism for specifying shadowing and error checking.

The Inheritance Graph Model is very general: All visibility features considered can be described using the basic Inheritance Graph Model. The Inheritance Graph Model can be used to represent both lexical and dynamic inheritance of visibility from enclosing scopes.

The Inheritance Graph Model can also be used as a basis for implementation. A reasonably efficient implementation of the visibility rules for a language can be generated from a

specification of the inheritance graph for the language. Data flow analysis is used to compute the set of bindings visible at each vertex of the inheritance graph for a program. The Inheritance Graph Model can also be used as a guide to hand-written implementations of visibility rules.

1.2. Previous Work

This discussion briefly discusses previous related work. More comprehensive discussions appear at appropriate places in later chapters, and particularly in Chapter 9.

Formal Language Specification Methods

A number of specification methods applicable to programming languages have been developed, including denotational semantics [Stoy 1977], various kinds of operational semantics [Plotkin 1981; Wegner 1972], and attribute grammars [Knuth 1968].

The standard technique for defining visibility in all of these specification methods is to pass around a large "environment" attribute in the grammar or rules used to describe the language. The environment is manipulated in a procedural manner, although the manipulation may be specified in a functional notation, as in denotational semantics. The semantics of the visibility control rules of a language are embedded in these auxiliary environment manipulation procedures, which are not an embedded part of the specification. It is also often difficult to localize the meaning of a visibility control construct in a single place: the effects of the construct may affect many parts of the language specification.

Reiss's Models of Visibility Control

Reiss developed a formal model of visibility control, and a model of visibility control embodied in a specification language used as a basis for the automatic generation of symbol tables [Reiss 1983]. The specification language contains features that allow the description of a fairly wide variety of visibility features, but the specification is declarative, with specific keywords corresponding to specific choices in the visibility rules. If there is no combination of keywords that corresponds exactly to the desired visibility rules, then those visibility rules will be difficult or impossible to define using the specification language.

Reiss's models will be described in §3.5, and discussed more completely in §9.2.1.

Wolf's Model of Visibility Control

Wolf's work [Wolf 1985; Wolf *et al.* 1986a; Wolf *et al.* 1987] is more concerned with language features for precise control of visibility of bindings than a general model of visibility control suitable for easily describing a wide range of language features. However, his *visibility graph* model does handle the constructs he was most concerned with (the provide and request operations in particular).

Only a very simple example was given, but it appears that the descriptions of many visibility control features using Wolf's model will be very complex. The visibility graph model requires that the *global* effect of each visibility construct be described, by creating edges between declarations of bindings and scopes where those bindings may be used. In contrast, the Inheritance Graph Model allows the effects of most visibility constructs to be described locally, with respect to the immediate context of the visibility construct.

Wolf's models will be described in §3.6, and discussed more completely in §9.2.2.

1.3. Plan of the Dissertation

This dissertation is organized in rough correspondence to the historical development of the ideas discussed in the chapters. This has been done in order to help the reader understand the reasoning behind the important ideas, instead of presenting them as if they had miraculously sprung

from some fount of knowledge. New ideas seldom develop this way; they usually result from a long process of exploration that hopefully matures into a proper understanding of the concepts of the problem at hand. Only then is a good solution usually possible. I have attempted to show a little of this process, although of course the presentation is of necessity much idealized. This goal is most apparent in Chapters 3, 4, and 5, which concern the factors affecting the design of a model of visibility control. Some earlier versions of models of visibility control and some blind alleys followed are omitted due to the need for some coherence, and a desire to keep the dissertation to a reasonable length.

An extensive and comprehensive survey of the visibility control features available in programming languages is presented in Chapter 2, with the goal of understanding what is required of a model of visibility control, and what the fundamental concepts of visibility control are.

Chapter 3 presents several models of visibility control, including models which have been implicitly used by programmers and language designers without ever having been formally stated, and formal models of visibility control that have been developed by others.

Chapter 4 presents the requirements for a general and natural model of visibility control, based on the fundamental concepts of visibility control derived from an analysis of the language features surveyed in Chapter 2. The available choices in the design of a model of visibility control are evaluated in Chapter 5, with the goal of determining what choices are important, what choices should be abstracted out, and what the effects of the various choices are.

The Inheritance Graph Model is presented in Chapter 6. This chapter also includes examples of descriptions of most of the visibility control concepts described in Chapters 2 and 4. Chapter 7 consists of an extensive example of the application of the Inheritance Graph Model to the description of the visibility control rules of Modula-2.

Chapter 8 presents a method of automatically computing an assignment to an inheritance graph, allowing resolution of references, based on data flow analysis. It also discusses other methods of implementation of visibility control rules, both automatic implementations from inheritance graph descriptions, and hand-written implementations using the Inheritance Graph Model as a guide.

The final chapter pulls together all of the main ideas discussed elsewhere in the dissertation, and discusses some other important issues that do not merit an entire chapter. It describes several potential uses of the Inheritance Graph Model. Chapter 9 also contains a more extensive comparison of work related to the research presented in this dissertation. Several language design issues are described, along with some suggestions for making the visibility control rules of languages easier to specify, implement, and understand. Avenues for further research in this area are also described.

CHAPTER 2

Survey of Visibility Control in Programming Languages

Programming language designs exhibit a wide variety of choices of visibility control rules. These choices are important not only because they greatly affect how a programmer uses a particular language, but also because they affect the implementation of translators (*i.e.*, compilers and interpreters) for the language, in particular the implementation of the name-resolution mechanism in the translator (often called the symbol table).

The purpose of this chapter is to describe the various visibility control features found in a variety of programming languages, especially those that concern the declaration and referencing of entities. Two different models of visibility control are introduced, and one of them is used throughout the chapter to help explain each visibility control feature as it is presented.

This is intended to be a comprehensive survey, covering some fairly complex visibility control features. However, in order to ensure a common base of knowledge with our readers, we[2] will take a somewhat historical approach, first presenting a few simple examples illustrating the progression from very basic visibility control capabilities to more complex and powerful ones. We will then carefully define our terminology to avoid potential confusion. Only then will we be able to explore some of the interesting but complicated visibility features found in more recent languages.

2.1. Introduction to the Use of Names in Programming

A *name* is a unique representation of an identifier, label, or other character string used in a program to denote some entity[3], such that all identifiers, etc., considered equivalent by a language map to the same name. The actual representation of names is unimportant.

A string of the form "A.B.C" does not normally correspond to a single name: rather, "A", "B", and "C" correspond to separate names that together form a qualified reference, which will be discussed in §2.6.2.

For the sake of exposition, identifiers will be used in examples in place of the corresponding names, with the translation from identifier to name explicit. We use double quotes when we are referring to an identifier (*e.g.* "ColorArray"). When referring to the entity denoted by that identifier (or the name corresponding to the identifier), we use italics (*e.g. ColorArray*).

2.1.1. FORTRAN

We begin our discussion using fairly common, though imprecise terms. Once we have presented a basic introduction of visibility control rules, we will move toward more precise, though less familiar terms. A sequence of examples is used to illustrate the fundamental concepts of visibility control.

---

[2] A slightly different version of this chapter was co-authored by me and my research advisor, Susan Graham. Thus, "we" in this chapter refers to the two authors, or is used in the all-inclusive sense. In other chapters, "we" is used only in the all-inclusive sense.

[3] Precisely what is meant by "entity" will be defined later.

Figure 2.1 contains a simple FORTRAN [ANSI 1966] program. The example reads a number, computes its square and cube, and prints the result.

Names are used primarily in two distinct ways in programs, illustrated by the uses of the identifier "answer" in the main program in this example. The first use is in the declaration of the real variable *answer*. This is called a *defining occurrence* of a name, or simply a *definition*, because it is defining the meaning of the identifier "answer". The other kind of use is in the write statement, where the value of *answer* is printed. This is called a *reference occurrence* of a name, or simply a *reference*, because the identifier "answer" is being used to refer to the variable defined earlier.

Names can sometimes be treated as objects themselves. In LISP [Clark and Weisman 1967], names can be manipulated just as any other value; for instance, they can be compared to one another. However, this kind of usage is the exception rather than the rule in programming languages, and we are most concerned with defining and reference occurrences of names.

FORTRAN has very simple visibility control rules. There are only two kinds of declarations: global declarations, visible or meaningful in the entire program (for instance *square* in the example above), and local declarations, such as *arg*, visible only in the function or subroutine where they are declared. Functions *square* and *cube* have distinct local variables, both named "arg".

For historical reasons, identifiers in FORTRAN consist of at most six alphabetic or numeric characters, the first of which must be alphabetic. Although the syntax of identifiers is not the central issue in this paper, it is a factor in how the naming and visibility control features of a

```
      real x, answer
      read (5, 10) x
10    format (f10)
      answer = square (x)
      write (6, 20) answer
      answer = cube (x)
      write (6, 20) answer
20    format (1x, f10)
      stop
      end

      function square (arg)
      real arg
      square = arg * arg
      return
      end

      function cube (arg)
      real arg
      cube = arg * arg * arg
      return
      end
```

Figure 2.1: A Simple FORTRAN Program

programming language affect the user. In particular, the restricted identifier construction rule visibility control rules for FORTRAN may limit the ability of the user to write understandable and modular programs. All names in FORTRAN designate variables or subroutines. Variable declarations are optional. The defining occurrence of a name may be the first reference to that name. In that case, the type of the named variable is determined by the spelling of the name's identifier.

2.1.2. ALGOL 60 and Block Structure

ALGOL 60 [Backus *et al.* 1960] was the first widely-known language to provide mechanisms for more precise control over the visibility of names. The concept of *block structure* was introduced in ALGOL 60 to permit more modularity in the use of names. Block structure allows finer control over the visibility of named objects than is possible in (say) FORTRAN. The ALGOL 60 example in Figure 2.2 illustrates block structure.

This example illustrates again that names can be used to designate more than one kind of object. In this case, "BlockStructureExample" and "print" denote operations (both procedures), and "var1", "var2", and "var3" denote variables. "B1" and "B2" name the blocks that follow them.

Each block in this ALGOL 60 program is introduced by the keyword **begin**, and ended by the keyword **end**. The defining occurrence of each variable is a declaration and the scope of the variable name is the block in which it is declared. As this program illustrates, blocks may occur inside one another in ALGOL 60. This is called *nesting* of blocks. A block is just another form of statement, and may occur anywhere a statement may occur, so in principle blocks can be

```
procedure BlockStructureExample
begin
    integer var1, var2; comment declarations of var1 and var2;
    var1 := 0; var2 := 0;

  B1:begin
        Boolean var1, var3;

        var1 := false; comment reference to inner var1;
        var2 := 1;
        var3 := false;
    end B1;

    print (var1); comment prints "0", not false;
    print (var2); comment prints "1";
    print (var3); comment illegal reference to var3 here;
  B2:begin
        integer var4, var5;
        ...
    end B2
end
```

Figure 2.2: Block Structure in ALGOL 60

nested to an unlimited depth[4]. Nesting is the fundamental concept of block structure, because it is the syntactic structure that makes possible the name control discussed below.

The example in Figure 2.2 illustrates how block structure makes it possible for a name to denote different things in different parts of a program. There are two separate variables with identifier "var1" in the procedure *BlockStructureExample*, an integer variable in the outer **begin** ... **end** block, and a boolean variable in the inner block *B1*. Within the inner block, a reference to "var1" denotes the boolean variable, while outside the *B1* (but still within the outer block), "var1" denotes the integer *var1*. We say that the inner declaration of "var1" *shadows* or *hides* the outer declaration of "var1", and we call this process *shadowing* or *hiding*.

An inner block *inherits* visibility of all declarations visible in the block enclosing the inner block, except those declarations shadowed by a declaration in the inner block. So, the assignment "var2 := 1" in the inner block *B1* modifies the variable *var2* declared in the outer block, because there is no (re)declaration of "var2" in *B1*.

Block structure provides several advantages. When writing a block or a procedure, a programmer is free to create local variables without concern that the new names are identical to names used elsewhere in the program – they may be identical, but that is not relevant to the correct formation of the program. The other variables with the same name either will not be visible, or will be hidden by the new local declaration. Because of the automatic inheritance of non-shadowed variables, the programmer can still reference non-local variables (this has disadvantages also, to be discussed later). The use of local declarations also helps programmers to understand a program – they can find the declarations of local variables more easily than those of global variables, saving time and page-turning. If the procedure references no global variables – only local variables and parameters – then the entire behavior of the procedure can be understood by looking at the procedure alone.

Block structure also controls access to names. The variables declared within a block cannot be referenced outside that block, so the programmer can be confident that changes to the values of local variables of a block can occur only due to references within that block or by explicit passing of variables as parameters to procedures or functions outside the block.

One of the early motivations for block structure in ALGOL 60 was the storage sharing it provided [Baumann *et al.* 1964]. Storage for the local variables of a block can be allocated on a run-time stack when the block is entered (activated), and deallocated when the block is exited. Storage is allocated only for those blocks that are currently active. The stack storage area is shared among the different blocks in a program. In Figure 2.2, the variables declared in block *B1* can share storage with the variables declared in block *B2*.

Despite its advantages over a "flat" name space with no locality, block structure has a number of shortcomings. The most important one is a result of its unrestricted visibility – a name declared local to a particular scope is visible at any point within all blocks nested within that scope, no matter how deeply nested, unless a redeclaration of that name intervenes. Thus, a global variable is visible in the entire program, even though its original intended use might be to share data among only a limited set of procedures. The programmer has no way of preventing other procedures from using or modifying the variable, and therefore must search the entire program to understand completely how the variable is used. This and other disadvantages are discussed more thoroughly in [Clarke *et al.* 1980; Hanson 1981; Wolf *et al.* 1987].

Block structure also affects the efficiency of program execution. Block entry and exit may require extra operations to maintain the state of the stack on which local variables are stored. On

[4] Implementations of block-structured languages often limit the maximum nesting depth for ease or efficiency of implementation.

the other hand, since local variables are usually referenced heavily [Shimasaki *et al.* 1980], an implementation can attempt to take advantage of this locality of reference by treating local variables specially.

2.1.3. An Ada Example

Our final introductory example (Figure 2.3) is part of an Ada[5] program. Ada has block structure also. Unlike ALGOL 60 blocks, however, the beginning of the (optional) declarations section of an Ada inner block is marked by the **declare** keyword, before the **begin** keyword that marks the beginning of the *body* of the block. In the example, the identifiers "A", "B", "C", and "D" are used in similar fashion to the way they might be used in ALGOL 60.

Two new uses of names are illustrated in Figure 2.3. The identifiers "green", "amber", "red", "blue", etc. are called *enumeration constants*; they designate values rather than variables or subroutines. They are members of the two enumeration types *stoplight* and *color*. In addition the identifiers "stoplight" and "color" designate *user-defined* types.

```
block_1_1:
  declare
    A      : integer;
    type stoplight is (green, amber, red);
    type color is (red, green, blue, yellow, orange);
    ColorArray : array [1..10] of color;
  begin
    A := 1;
    ColorArray [A] := green;
    block_2_1:
      declare
        B      : integer;
      begin
        block_3_1:
          declare
            C      : integer;
          begin
            A := 1;
            C := 2;
          end block_3_1;
      end block_2_1;
    block_2_2:
      declare
        D      : integer;
      begin
        D := A;
      end block_2_2;
  end block_1_1;
```

Figure 2.3: Use of Names in Ada

[5] Ada is a trademark of the United States Department of Defense.

Another facet of this example is the multiple use of a single name in the same block. For example, the identifier "red" is declared both in the type *stoplight* and in the type *color*. In many languages (ALGOL 60, for one), this would be an error. It is not an error in Ada, however. Ada permits *overloading*; a name can denote more than one object at a single place in a program (as distinguished from the ALGOL 60 example, where a single name could denote different objects, but only in different parts of the program). The context of the name's use determines which object is being referenced by the name. For example, since the elements of the array are of type *color*, the occurrence of *green* in the assignment to *ColorArray*[*A*] designates a *color* value rather than a *stoplight* value.

The final item of note in this example is the declaration of the array variable *ColorArray*. In this declaration, an array type is created without an explicit declaration of that type, and without giving that type a name. That is the only way that arrays can be created in FORTRAN or in ALGOL 60. However, this is called an *anonymous type* in Ada, since the possibility also exists that types can be named. The example illustrates that not all objects of a language need have names.

The examples given so far illustrate only a small selection of the variety of naming and visibility control rules introduced by language designers. In fact, we have not even summarized completely the visibility control rules of FORTRAN, Ada, or ALGOL 60. However, in order to continue, we need to introduce more precision and detail.

§2.2 describes bindings: the association of names with entities upon declaration. §2.3 describes scopes, and how they are used. In §2.4, we discuss declarations and how they affect visibility. §2.5 discusses visibility, and how the entity referred to by a name is determined. Some of the various forms of explicit scope control are discussed in §2.6. §2.7 discusses miscellaneous language features which are concerned with visibility control.

Examples are presented in Ada or Pascal when possible. Examples not expressible in Ada or Pascal are presented in extensions to these languages or in a pseudo-language whose semantics either are obvious or are explained in the text.

This paper does not attempt to discuss language features other than those directly related to visibility control rules. Sources of collected, more general information about other aspects of programming languages include [DeRemer *et al.* 1979], [Wexelblat 1981], [Pratt 1984], [Tucker 1977], [Higman 1977], [Barron 1977], and [Feuer and Gehani 1984].

Languages considered in compiling this discussion of features include those listed in Table 2.1 (in alphabetical order), though the discussion is not explicitly limited to these languages. When we mention LISP without mentioning a specific dialect, we are referring to a traditional, interpreted LISP such as LISP 1.5 [Clark and Weisman 1967].

2.2. Binding Names to Entities

At any instant of time, an executing program is in a certain *state*, consisting of the current values of its variables and other objects in its executing *environment*. An executing program changes and accesses its state explicitly by assigning values to and reading values from variables, and implicitly by evaluating control statements and expressions[6].

A name may be used in a program in order to denote the value of a corresponding variable, providing access to the program state. It may be used by the programmer to change the program state by assigning values to variables. As discussed earlier, names are also used to reference

[6] Some languages, for instance FP, have no notion of programmer defined variables [Backus 1978]. Nevertheless, executing programs in these languages do have execution state, but this state is only changed implicitly by actions such as expression evaluation, and can neither be accessed nor changed by the programmer.

Language	Reference(s)
Ada	[USDoD 1983]
ALGOL 60	[Backus *et al.* 1960] [Baumann *et al.* 1964]
ALGOL 68	[Van Wijngaarden 1976]
Alphard	[Wulf *et al.* 1976]
APL	[Iverson 1962]
Basic	[Kemeny and Kurtz 1964]
Bliss	[Wulf *et al.* 1971]
C	[Kernighan and Ritchie 1978]
CLU	[Liskov *et al.* 1981]
COBOL	[USDoD 1961]
Common Lisp	[Steele Jr. 1984]
EL/1	[Wegbreit 1971]
Euclid	[Lampson *et al.* 1981]
FORTRAN	[ANSI 1966]
Icon	[Griswold and Griswold 1983]
L	[Cormack 1983]
LISP	[Clark and Weisman 1967; Fox 1960]
LISP Machine LISP	[Weinreb and Moon 1981] [Symbolics, Inc. 1986]
Mesa	[Mitchell *et al.* 1979]
Modula	[Wirth 1977]
Modula-2	[Wirth 1982]
Pascal	[ANSI 1983; Jensen and Wirth 1974]
Plain	[Wasserman *et al.* 1981]
PL/I	[ANSI 1967]
PROLOG	[Clocksin and Mellish 1981; Roussel 1975]
Rigel	[Rowe *et al.* 1981]
Scheme	[Steele Jr. and Sussman 1978a]
SETL	[Dewar 1979; Schwartz 1971; Schwartz 1973] [Schwartz *et al.* 1986]
Simula 67	[Dahl and Nygaard 1967; Dahl and Hoare 1972] [Birtwistle *et al.* 1973]
Smalltalk	[Goldberg and Robson 1983]
SNOBOL	[Maurer 1976]
T	[Rees *et al.* 1984] [Rees and Adams IV 1982]
TEMPO	[Jones and Muchnick 1978]

Table 2.1: Languages Included in This Survey

entities that may have no run-time representation, such as types, to denote operators or other computational objects such as procedures and functions, or to denote values, as in defined constants.

The use of symbolic names to denote variables is not absolutely necessary; the programmer could use memory addresses on the underlying machine. Symbolic names are for the programmer's convenience, and date back to the first interpreters and symbolic assembly languages. Their use frees the programmer from many low level programming details. If well chosen, the names also provide a modicum of abstraction and self-documentation.

An *entity* is any conceptually separate program object, or a descriptor thereof, that may possess attributes that affect the semantic correctness of a program – for example, a type, a constant, a variable, a field in a record, or a routine. Most importantly, an entity can be bound to a name, which can then be used to reference the entity. Whether an entity is better considered as an object or a descriptor depends on whether the process of resolving a name to an entity is dynamic, where a name is usually resolved to a specific storage location, or static, where a name is resolved to a descriptor, which may represent a compile-time concept such as a type, or an object such as a variable declared within a recursive routine which may have many instances active in any particular program state. In the latter case, the entity represents a class of run-time objects. There must be a separate mechanism to map correctly from the entity to the appropriate instance of that entity. In the case of the local variable of a recursive routine, each new instance of the variable is allocated on a stack, and a run-time access method must find the correct instance on the stack.

In a compiled implementation of a language, each entity is usually described by a separate entry in the symbol table, which records its attributes. For example, if a record type is declared, the corresponding symbol table entry includes the names and types of its fields (or pointers to the separate entries for the fields), information about variant parts, if present, and default values, if any. The definition of an entity is of necessity somewhat vague, because slightly different definitions of an entity may be desirable in different contexts, for instance for use in describing different languages.

The specification of a programming language must define how names are bound (mapped) to the corresponding entities and their values. This binding process may be conceptually factored into one or more steps. What these steps are vary according to personal taste, the language under consideration, and even the implementation of that language.

We use the Ada example in Figure 2.3 to illustrate the meanings of the different steps in mapping from a name to a value: Consider the assignment of the value 1 to the variable *A* in *block_1_1* in Figure 2.3. The first step binds the name corresponding to "A" to the entity for the variable *A*. That entity has the attribute that its values are integers.

Another attribute associated with the entity is an access method that defines how to find the correct storage location at run-time. The access method may simply return a storage address, for statically allocated variables, or a stack offset for local variables in a simple block-structured language. The access method may be a more complicated search procedure. The second step uses the access method to find the storage location. The third and final step binds the storage location to a value (*i.e.* a memory access).

For the reference to the type *color* in the declaration of *ColorArray*, the first step is the same: the name "color" is bound to the entity for the type *color*. However, the similarity ends there – the type *color* has no access method, run-time storage location, or run-time value associated with it. For a function, its code segment could be regarded as its value.

In general, this binding process varies for different classes of entities, languages, and implementations. In a compiled implementation of a language, the binding steps described above for the variable *A* would usually be appropriate for variables. The first step of variable binding would occur at compile time, the access method to storage location binding at load time or run-time, and the final step – producing a value – at run-time. For constants, all steps can often be done at compile time. In some languages and implementations, it is more convenient to identify an entity with a single storage location, so the first two binding steps described above are collapsed into a single run-time step.

This dissertation is primarily concerned with the first step of the binding process: that from names to entities. This is the step with which visibility control rules of programming languages are concerned. Programming language users must understand this step in order to understand the meaning of their programs. This is the step handled by symbol tables (also known as declaration

tables or name tables) in many conventional implementations of translators.

We will not discuss the issues of run-time organization and entity-object binding further. In their example language TEMPO [Jones and Muchnick 1978], Jones and Muchnick give a good presentation of the different stages of binding that occur, when they can occur, and how they affect a language implementation. Schwanke [Schwanke 1978] also discusses entity-object binding.

2.2.1. Declarations

In many languages, the association between a name and an entity is established by a *declaration*. The declaration of a named entity within a program can itself be divided into two major steps:

(1) Entity Definition: an entity is created, with attributes describing its characteristics.

(2) Binding: a name is associated with the entity. For instance, in a type declaration the name of the type is bound to the entity describing the type. A *binding* is a pairing of a name with an associated entity. This step is omitted if the entity is *anonymous* (unnamed). For example, in Ada or Pascal, an array type may be described as an attribute of a variable or another type and never associated with a name. Note that the word "binding" is used both as a verb and as a noun. From this point on, the noun "binding" will only be used to mean a name-entity binding, unless otherwise noted.

When these steps occur depends on the precise definition of an entity in a language. If an entity is a descriptor for a compile-time object or for a class of run-time objects, and the effects of a declaration on visibility can be determined statically, then the two steps of a declaration can be done statically at compile time. If an entity is a run-time object, then the two steps must be done dynamically at run time.

We use the Ada example in Figure 2.3 to illustrate the two steps of a declaration: For the first declaration in Figure 2.3, the new entity (call it e_1) has attributes stating that it is a variable with values of the predefined type *integer*. The binding step associates the identifier "A" with the new entity e_1. For the declaration of *stoplight*, several entities are involved. First, entities are established for each of the enumeration constants (*green*, *amber*, and *red*), and the appropriate name (for example "green") is bound to each entity. After all of the bindings for the constants are established, an entity that refers to the bindings of *green*, *amber*, and *red* as its member constants is created for the enumeration type and bound to the name "stoplight".

The declaration of the enumeration type *color* is handled the same way. Remember, however, that this declaration causes the names corresponding to "red" and "green" to be overloaded. This fact does not affect the entity creation step – *new* entities are created for the constants *red* and *green*, and the names "red" and "green" are bound to these new entities.

For the more complex array declaration, more than one entity is involved. The first entity (call it e_2) is the subtype of *integer* with a range from 1 to 10. This is an anonymous type (remember that an anonymous entity is one having no name), so there is no binding step for e_2. The second entity (call it e_3) associated with the array type declaration defines the array type itself. Its attributes state that it is an array with elements of type *integer*, and with index type described by entity e_2. Finally an entity (call it e_4) defining the variable whose type is described by e_3 is created, and the name corresponding to "ColorArray" is bound to e_4.

An entity definition usually results from a programmer-written declaration, although that is not always the case, as with implicit declarations in FORTRAN. The two stages in entity declaration may occur at different times, depending on the language or the language translator, and such concerns as ease of implementation and the efficiency of the resulting implementation.

Most languages predefine some entities, which are predeclared by the translator before beginning compilation of a program. An example is the predefined type *integer* in Pascal and in many other languages. Language-defined mathematical function libraries are another example. In languages such as ALGOL 60, ALGOL 68, Pascal, and Ada, all other programmer-referable entities must be explicitly declared by the programmer. In FORTRAN and PL/I, an attempt to reference an unbound name as a variable will result in an implicit declaration of an entity with default characteristics, and the binding of the name to that entity.

Programming language implementations usually define entities at the earliest possible stage, for efficiency. For ALGOL-family languages, entities are usually statically defined, because the rules of the languages are defined so that all entities depend only on information that can be computed at compile time. Thus, all processing of declarations and resolving of references can be done at compile-time. The processing of declarations and references could be done at run-time, but there is little reason to do so, and the programs would run much more slowly. In languages that are traditionally interpreted, such as SNOBOL and LISP, run-time creation of identifiers (and thus names) is permitted, so at least the binding step must be done at run-time in some cases.

2.3. Scopes

The word "scope" has multiple meanings in the computing literature. The "scope" of a binding usually refers to the range of program text over which the bound name has a certain meaning. In contrast, the word "scope" is also used to refer to a syntactic unit, such as a block, a procedure, a record, or a module. As discussed in the introduction, the term "scoping rules" is often used instead of visibility control rules. Fortunately, the meaning is usually clear from the context. To avoid confusion, we will use the term *range* for the former meaning of scope: the range of a binding encompasses all points in a source program where the name component of the binding can refer to the entity component of the binding. We will use the term *scope* to denote syntactic units as described above. Our choice of terminology is different from that in [Schwanke 1978], another paper what discusses some of the same issues. ALGOL 68's usage of the terms "scope" and "range" is the reverse of our usage. We feel that our definitions of scope and range are more intuitive.

In this section, we define precisely the meaning of scope, along with the associated terminology.

Scopes provide environments for declaring and referencing bindings. Programs in some languages (for example Basic) consist of only a single scope, and thus have only a single name space. FORTRAN programs have two levels of scopes: local and global. All variables are declared in local scopes, and all common-block names and subprogram names are in the global scope. Entities declared in a local scope can only be referenced within the same subprogram, while entities declared in the global scope can be referenced within any subprogram. No nesting of scopes is allowed. Most modern programming languages permit the programmer to define any number of scopes, using the above-mentioned syntactic structures, as well as others. As illustrated in the ALGOL 60 and Ada examples, a scope may be nested within other scopes.

We illustrate some of the terminology of scopes with the Ada example in Figure 2.3:

- Given a scope *S*, an *inner* scope is any scope *R* textually contained entirely between the delimiting brackets ("**begin** ... **end**", "**procedure** ... **end**", etc.) of *S*. *R* is said to be contained within *S* or *nested* within S. In the example, *block_2_1*, *block_2_2*, and *block_3_1* are inner scopes of *block_1_1*, and *block_3_1* is also an inner scope of *block_2_1*. An inner scope is sometimes called a *subordinate* or *inferior* scope.
- An *outer* (*enclosing*) scope of *S* is any scope *R* whose delimiting brackets surround *S*. *R* is said to contain *S*. In the example, *block_1_1* is an outer scope of *block_2_1*, *block_2_2*, and

block_3_1.

- A *directly enclosing* scope of *S* is the scope which encloses *S*, but which does not enclose any other scope which encloses *S*. *block_1_1* directly encloses *block_2_1* and *block_2_2*, but not *block_3_1*. An outer scope is sometimes called a *dominant* or *superior* scope.
- A scope *R* is *directly contained within S* if *S* contains *R*, and *S* does not contain any scope which also contains *R*. *block_3_1* is directly contained within *block_2_1*.
- A declaration *local* to a scope *S* is a declaration located within *S*, but not within any scope contained within *S*. A scope directly contained within *S* is also said to be local to *S*. In the example, *B* and *block_3_1* are local to *block_2_1*. Declarations *global* to *S* are declarations in scopes enclosing *S*. All names except *C* and *D* are global to *block_3_1*, assuming there are no other declarations of *C*.
- The *global scope* is the outermost scope of a program, and the global declarations of that program are those occurring in the global scope. Note that these uses differ from the previous use of the term "global", in that "global scope" and "global declarations" are absolute terms, and "the declarations global to a scope *S*" is relative. The uses are related, but different in an important way.
- Two scopes are *parallel* to each other if they are directly contained within the same scope. *block_2_1* and *block_2_2* are parallel scopes.

Scopes perform a number of functions:

- Scopes provide the basis for grouping declarations and the associated bindings. A declaration local to a scope *S* (sometimes called a local declaration) is associated with *S*. A scope *S* introduces a *contour* C_S that is the local environment associated with and defined by *S*[7]. Associated with C_S is a set of bindings corresponding to declarations local to *S*, and other bindings explicitly made visible within *S* (to be discussed in later sections). This set of bindings is the primary component of the local environment of *S*. We use much of the same terminology for contours as for scopes, such as inner and outer contours, parallel contours, etc. Contours were introduced by Johnston [Johnston 1971] in a model for run-time storage allocation in block structured languages. Our use of the term "contour" is similar to his; his contours contained name-storage cell bindings, ours contain name-entity bindings. The difference is that we extend our use of contours to cover scoping disciplines other than block structure, and we are less concerned with run-time issues.
- Scopes provide the basis for name resolution – that is, for determining which entity is bound to a name. Name lookup can usually be done (conceptually) by searching the contours of one or more scopes for a binding of that name. This issue will be discussed in §2.5.
- Scopes limit visibility for local bindings. Entities may be declared close to the point of use, and the limited visibility helps prevent interference with other uses of the same name.
- Scopes provide the basis for block-structured storage allocation. A simple block-structured language is one having nested scopes which can contain local variable and routine declarations. In an implementation of such a language, entry of a scope at run-time causes allocation of all entities declared within it, and exit of the scope causes deallocation of those entities. The stacking of scopes leads naturally to a simple and efficient storage allocation method. As illustrated in §2.1.2, the storage sharing between parallel blocks was one of the justifications for

[7] This is actually a simplification: a scope may have several contours associated with it. In a language where declarations must precede references, the visibility control rules are described most easily by associating each declaration with a new contour, with that contour containing only the bindings resulting from that declaration, and inheriting visibility from the contour associated with the preceding declaration.

nested blocks in ALGOL 60 [Baumann *et al.* 1964]. Run-time allocation (the lifetimes of run-time entities) is related to compile-time naming and and the ranges of names, but there are different issues involved. This dissertation is primarily concerned with ranges and scope, not with run-time allocation, so the latter will not be discussed further.

- In many languages scopes can be named. For example, in Figure 2.3, *block_1_1* names the outermost scope. In certain cases, named scopes provide an access path to the declarations immediately within a particular scope. The most common manifestation of this kind of explicit access is with records, as in Pascal. For example, given the Pascal declarations in Figure 2.4, one can reference the *day* field of *today* with the notation "today.day", even though the name "day" is not visible outside the record scope. Using the name "today" to name the record causes the current referencing environment to be that described by the record type *today*. In Modula, one can refer to bindings within modules in the same manner one refers to bindings (fields) within records. Bindings created in a subprogram in Ada can be referenced with this "dot" notation also, though the reference has to be contained within that subprogram scope. When calling a subprogram in Ada, the parameter names can be used within the environment of the call (within the actual parameter list), though they are not otherwise visible outside the subprogram scope.

Block-structured scoping was one of the first major language features intended to aid in structured programming [Dijkstra 1972a; Dijkstra 1972b]. The locality of reference it provides allows programmers to choose distinct names for logically separate variables, and yet to reuse the same meaningful names in different parts of a large program. More recently, some researchers have questioned whether nesting of scopes is needed if named scopes exist [Clarke *et al.* 1980; Hanson 1981; LeBlanc and Fisher 1979; Wolf 1985]. While block structure is historically very important, and still a very important feature of many languages heavily used today, it has not been included in some of the newer languages. Other more powerful constructs (discussed later) that make block structure unnecessary are available.

In summary, scopes are (possibly named) syntactic entities, (possibly) containing the declarations of other entities, and providing access to those entities.

2.3.1. Scope Disciplines other than Static Nesting

The previous discussion has been limited to nested scopes. A more general alternative would be overlapping scopes. However, we know of no programming language that permits overlapping scopes. The apparent consensus among language designers is that overlapping scopes are neither necessary nor natural, so such scope usage will not be discussed further, though overlapping scopes are covered by the model of declaration and naming considered in this paper.

```
type Date =
        record
                day     : integer;
                month   : 1..12;
                year    : integer;
        end;

var today :     Date;
```

Figure 2.4: Pascal Record Type and Variable Declarations

One could consider a macro processor such as the one defined for the C language [Kernighan and Ritchie 1978] to have overlapping scopes. A macro may be defined at any point in a source file, and then undefined at any later point, so the range of any particular macro may be viewed as defining a new scope, possibly overlapping the scopes defined by other macros. However, one may also view the entire source file as a single scope for which macro definitions may be both added and removed. SETL has a similar capability, though with an important difference: The macro language for C is actually quite separate from the language C itself, and simply allows the programmer to substitute arbitrary chunks of (parameterized) text for each macro call. However, the macro language for SETL is very closely tied to the rest of the language – macro bindings obey the same scoping rules as any other entity, except that a macro binding can be removed at arbitrary points in the source program.

2.4. Visibility Rules and Declarations

The visibility rules of a language define which name-entity bindings are *visible* – that is, can be referenced – at a particular point in a program. Alternatively, one can say that they define the range of each binding. Visibility rules may be quite complicated; they vary greatly from language to language. But, before discussing how to find entities corresponding to a name, we must describe how names and the entities to which they are bound are made visible.

2.4.1. Locations of Declarations

An important determinant of the visibility of a binding is the location of its declaration in the text of a program. An attempt to declare an entity occurs at a particular point in a program. That point may in general be within one or more nested scopes, and there may be other scopes that do not contain the point of declaration. It was stated in §2.3 that the declarations contained within a given scope can be viewed as creating a set of bindings associated with that scope's contour. However, things are not quite as simple as they seem, because the contour to which a declaration should be added may not be the innermost enclosing contour, though this is usually the case. For instance, in PL/I, implicit declarations are added to the outermost contour. In Ada, labels are implicitly declared in the contour associated with the innermost enclosing subprogram package, or task body, although there may be other declaration scopes (blocks) intervening.

In general, a language specifies rules for deciding to which contour a binding should be added. In all the languages considered, the rules can be written in the form:

> "Add the binding to the innermost enclosing contour (starting with the current contour) satisfying predicate **P**."

P is usually the constant (trivial) predicate "true". That is, declarations are usually added to the current (most local) contour. In PL/I the predicate for implicit declarations is "contour for binding = global contour," meaning that all implicit declarations occur in the outermost scope. P may depend on some property of the entity being declared.

2.4.2. Adding Bindings to Contours

Once the proper contour is found, the binding must be added to the contour's set of bindings. For each language there are rules describing what happens when an attempt is made to add a new binding to the set of bindings of a given scope. Four different actions are possible when an attempt is made to add a binding to the set of visible bindings in a scope, and the new binding's name is already visible in that scope:

Shadowing
: The new binding shadows any other potentially visible binding of the same name, and becomes the only visible one.

Overloading
: The new binding *overloads* another binding (or bindings) of the same name. Overloading occurs when one name is bound to more than one entity, and more than one of these bindings is visible at the same point in the program. When overloading occurs, additional semantic information must be used to distinguish among the bindings. Ada is one language that permits overloading.

Reference
: The new declaration is only a completion of an earlier declaration, and the new declaration is a reference to the old one. An example is the completion of a forward routine declaration in Pascal.

Error
: A binding already exists that cannot be hidden, overloaded, or referenced by the new binding, so the new binding is not legal. This situation occurs most commonly when an attempt is made to declare the same name twice in the same scope in a language which does not support overloading, for example in ALGOL 60 or Pascal.

Shadowing was illustrated by the ALGOL 60 example in Figure 2.2. The Ada example in that section introduced the notion of overloading. The following example further illustrates these four possible actions.

The example in Figure 2.5 is standard Pascal, except for the presence of multiple declarations of the function *times*, which will be discussed shortly. The reader should assume that the declarations shown are the only declarations, except for the declarations of the types *Array* and *Vector*. Then, the first function declaration is a new declaration. Note also the presence of the **forward** keyword, denoting in Pascal that this is an incomplete declaration. The next function declaration has the same name. so two different actions are possible here, depending on the rules of the language: If overloading of functions is allowed, which is not legal in Pascal, then the new declaration of "times" overloads the previous one, because its formal parameters have a different type from those of the first declaration. If overloading is not allowed, as in standard Pascal, then the new declaration is not legal, and causes an error.

The third declaration of "times" has a function heading identical to the first declaration, so by convention it is a completion of the first declaration of "times". The new declaration is a reference to the old one.

```
function times ( a1, a2 : Array ) : Array; forward;

function times ( v1, v2 : Vector ) : integer;
begin
  -- body of function to compute dot-product of two integer vectors
end;

function times ( a1, a2 : Array ) : Array;
begin
  -- body of function to multiply two square arrays
end;
```

Figure 2.5: Example of Different Possible Actions for a Declaration

If the name is not visible already and the declaration is otherwise legal, the binding is normally added to that contour.

The visibility control rules of a language determine into which of the above four cases a new binding falls. Deciding into which of the four declaration classes a new binding falls may involve only examining the current scope, or it may involve a full search for bindings of the same name. In Pascal and ALGOL 60, we need only examine the current scope, since a new declaration will hide any binding with the same name in an outer scope. In Euclid and in some other languages, an attempt to redefine (shadow) a visible binding is an error, so it is necessary to search for other bindings of a name when attempting to add a new binding of that name to a scope. For example, the scope structure in Figure 2.6 would be legal in Pascal, but illegal in Euclid.

2.4.3. Order of Declarations and References

Many more recently designed languages which require explicit declarations also require that a declaration of a name textually precede any reference to that name. Standard Pascal is an example of such a language. Originally, Pascal did not have this restriction. For example, neither the technical report describing revised Pascal [Wirth 1972] nor the Pascal report published by Jensen and Wirth [Jensen and Wirth 1974] require declaration-before-use. However, it appears as an implementation restriction for the CDC6000 compiler [Wirth 1972, p. 40] and as part of the Pascal User Manual [Jensen and Wirth 1974, pp. 8,18] and came to be incorporated in the language definition [ANSI 1983; ISO 1980]. The visibility rules for ANSI/ISO Standard Pascal will be assumed in all further discussions of Pascal unless otherwise noted.

Pascal and some other languages ease this restriction slightly to allow for mutually recursive types. The example in Figure 2.7 illustrates two such types, intended to define alternating elements of a list: a *NodeOne*, then a *NodeTwo*, then a *NodeOne*, and so on. Since each type must contain a reference to the other type, each must follow the declaration of the other according to the declaration-before-use rule, which is impossible. This dilemma is avoided by allowing pointer declarations to reference types not yet declared, as shown in the example.

The same problem arises with mutually recursive routines. The usual solution is to separate the declaration of the routine heading (*i.e.*, its interface information) from the presentation of its body. In Pascal, this separation is done by means of a **forward** declaration, as shown in Figure 2.5.

Another option for the allowable order of declarations and references is illustrated by PL/I: In PL/I, a declaration can appear anywhere within the body of the program, even after references

```
declare
  a : integer;
begin
  declare
    a : integer;
  begin
    ...
  end;
  ...
end;
```

Figure 2.6: Shadowing of Outer Bindings

```
type    NodeOnePtr = ↑NodeOne;
        NodeTwoPtr = ↑NodeTwo;

        NodeOne =
                record
                        { ... data fields ... }
                        next    : NodeTwoPtr;
                end;

        NodeTwo =
                record
                        { ... data fields ... }
                        next    : NodeOnePtr;
                end;
```

Figure 2.7: Mutually Recursive Types in Pascal

to that declaration. The impact of this feature upon language processors is to force all declarations in a scope and all enclosing scopes to be processed before any references in that scope can be processed. The declaration-before-use restriction helps to make one-pass semantic analysis possible – this was one of the major reasons for its introduction (note the progression of the declaration-before-use restriction in Pascal from an implementation restriction to a part of the language definition). An advantage of declaration-before-use is that sequential reading of a program is easier, since declarations are encountered before uses. Unfortunately, this restriction also forces the main body of a program to be at the end of the program text, which has disadvantages for readability. Such programs tend to appear in "reverse order", with the highest level procedures occurring last. The underlying problem is that requiring declaration-before-use results in overspecification: an order is required that has no real meaning to the programmer.

There is a subtle interaction between declaration-before-use and shadowing (hiding) in a language. Since a local binding b_{inner} is visible only *after* the point of its declaration, it is not obvious at what point an outer binding b_{outer} of a name "N" shadowed by another binding b_{inner} of "N" becomes shadowed. One possibility is that b_{outer} is visible up to the point of b_{inner}'s declaration, either to where b_{inner} becomes visible, or to the beginning of $b_{inner's}$ declaration, as is the case in Ada. The other possibility is that b_{outer} is only visible up to the beginning of the scope in which b_{inner} is introduced. The latter choice is the one made in Pascal. Thus, even though a declaration is not visible until after its declaration, it has an effect throughout the entire scope: shadowing bindings in an outer scope.

This issue illustrates another disadvantage of declaration-before-use: continuing the same Pascal example, a reference to "N" occurring in b_{inner}'s declaration scope, but prior to b_{inner}'s declaration, is illegal, but can't be detected by a one-pass compiler until the declaration of b_{inner} is encountered.

2.4.4. Static vs. Dynamic Name Creation

Identifiers (and thus names) are most commonly created statically by the programmer (by writing them in a program). In SNOBOL, they can be created dynamically (at run-time), using the language's string operators, and then used to declare or reference entities. This can also be done in LISP, because symbol table operations are accessible from within LISP programs.

```
declare
  foobar : integer;
  s1, s2, s3 : String;      -- type String declared in an outer scope
begin
  s1 := 'foo';
  s2 := 'bar';
  s3 := s1 catenate s2;
  $s3 := 5;  -- assignment to foobar (using SNOBOL syntax for indirect reference)
end;
```

Figure 2.8: Dynamic Name Creation

Figure 2.8 illustrates how dynamic name creation might appear in a program. The advantage of dynamic name creation is the additional flexibility and power it provides. A general facility for run-time creation and modification of programs requires it. One obvious disadvantage is efficiency, because all binding of such names must be delayed until run-time. Some improvement in efficiency can be gained by determining bindings at compile-time where possible, and delaying until run-time where it is not possible. Of course, in some cases, run-time creation of visible bindings may invalidate previously visible bindings. Another important disadvantage of dynamic name creation, which applies also to other forms of run-time binding creation, is that it becomes impossible to do complete compile-time type-checking and some other semantic checking. A new class of run-time errors is thereby introduced. It is possible to encounter references to unbound names, type mismatches, and other errors at run-time. Of course, many programmers would rather live with these potential problems than give up the expressive power that makes them possible.

Dynamic name creation is most often found in languages that traditionally have been implemented with interpreters. The causality runs in both directions: dynamic name creation is easier to handle in an interpretive system, because all of the necessary information is already present at run-time. Thus, designers and implementors are more likely to add this feature to an interpretive system, because it is easy to add and doesn't degrade performance significantly, whereas this does not hold for a traditional compiled language system. In the other direction, the dynamic name creation requires the a lot of information and analysis at run-time that otherwise would only be necessary during semantic analysis, so it is usually easier to implement in an interpretive system. Dynamic name creation seems to be more "consistent" with interpretive systems. However, a great deal of LISP work is now done using compiled code, but the dynamic name creation facilities and other dynamic facilities are retained for use when needed. Good LISP compilers can usually avoid the cost penalty introduced by the dynamic facilities except in the instances where these facilities are actually used.

2.4.5. Static vs. Dynamic Creation of Bindings

Names can be bound to entities either statically or dynamically. In the extreme case all program analysis can be intermixed with program execution. The motive for deferring binding to run-time is so that it can vary depending on the execution of the program. LISP is the most popular language in which binding is typically deferred. Programs can add bindings to scopes or remove them. Languages with dynamic name creation usually also have dynamic binding creation, and vice-versa, but this is not necessarily true. Some versions of PROLOG [Clocksin and Mellish 1981; Roussel 1975] allow the programmer to add a binding ("assert a clause") and

remove a binding ("retract a clause") at run-time and to dynamically create names[8].

2.5. Visibility Rules: Resolving References

The ultimate purpose of creating entities and binding names to entities is of course to be able to reference those entities. We now describe, given an name reference occurrence at a particular program point, some of the ways of determining to which entity that name refers.

A reference (an occurrence of a name "D") occurs at a particular point in a program, immediately within some scope S. Either a binding of "D" exists in the contour C_S associated with S ("D" is *bound* in C_S), or no binding exists in C_S, in which case we say "D" is *unbound* or *free* in C_S, and we must look elsewhere to find a binding of "D"[9].

We have previously defined the term "visible": A binding is *visible* where the name bound by the binding can be used to denote the entity bound by the binding. Following Reiss's terminology [Reiss 1983], we say that a binding is *directly visible* where the bound name can always refer to the bound entity, regardless of other bindings. Usually, a binding is directly visible in the contour corresponding to the scope where it was declared.

The attempt to find bindings of a name "D" can be described in terms of searching the contours defined by a program's scopes one at a time, examining each contour's set of bindings in turn for a binding of "D". We call this model of binding resolution the *Search Model*. The scope rules of each language define the order in which the contours must be searched. With this model of name resolution, we don't need to know *a priori* in which scopes each binding is visible; rather, we just need to know where bindings are declared and which contours must be searched when resolving a reference.

Each language imposes constraints on the order in which scopes must be searched for a binding of the name (though order may be irrelevant in some cases). As stated earlier, a language (for example Basic) may have a "flat" structure – that is, there is only one scope in which all bindings exist, so that there is only one contour in which to look for bindings. The most common underlying scheme is block structure: the contours containing the point of reference are searched from innermost to outermost, and the innermost binding of the name is selected. If declaration of the same name exists in two or more nested contours, then normally the inner declaration shadows the outer one(s) for the range of the inner binding. A simple example can be found in the innermost block (block *block_3_1*) in the Ada program in Figure 2.3 The name "C" in the assignment "C := 2;" refers to the variable *C* declared in the local scope, and so a binding is found in the local contour. The name "A" in the assignment "A := 1;", however, has no binding in the local contour or in the immediately enclosing contour (*block_2_1*), but a binding is found in the outermost contour (*block_1_1*). This is, in essence, the scheme adopted by ALGOL 60 and its derivative languages. However, the rules may require other contours to be searched first[10], instead of, or in addition to the enclosing contours.

An alternate point of view to doing name resolution by searching contours for one containing a binding of the desired name is to change the association of information with contours.

---

[8] Clauses in PROLOG are not quite the same as bindings as we have defined them in this paper. However, bindings and clauses are related. A full discussion of the relationship is beyond the scope of this paper.

[9] In most languages, if a binding exists in C_S for a reference of "D" immediately within S, then we need not look for other bindings of "D" in other contours. This is not always true, however; in Ada it is sometimes necessary to find all bindings of a name that could possibly be visible at the point of reference.

[10] In Ada, when processing a package body, if a referenced name is not declared within that body, the package specification's contour (a parallel contour) must be searched before the contour enclosing the body's contour. Modula-2 has a similar rule for definition and implementation modules.

Instead of just containing bindings for local declarations, let each contour contain the set of all bindings visible within that contour. We can call this set of visible bindings the *visible set*, and the resulting model of visibility control the *Visible Set Model.* So, in a simple block structured language, the visible set for an inner contour would contain bindings for all local declarations plus all bindings from outer scopes, *except* those bindings that are hidden by some binding in an inner contour. Describing the visibility rules for a language using this model largely reduces to defining the visible set for any contour[11]. It is still necessary to know the local bindings of a contour in order to check for multiple declarations and to handle some other language features. The Visible Set Model is better for describing certain types of visibility control features, but the Search Model is simpler and probably easier to understand for most features, because it breaks the name resolution process into two separate parts:

(1) Declaring entities and adding their bindings to contours.

(2) Searching contours for bindings of names.

Thus, for simplicity's sake, we use the Search Model in this chapter to describe visibility control features. Fuller descriptions and discussions of these and other models of visibility control are in later chapters.

2.5.1. Closures

A *closure* is a function or procedure together with an associated environment – in the Search Model, the environment consists of the contours that must be searched for bindings, the order in which they should be searched, and any other necessary binding information. Usually, the environment is defined implicitly by the location of the function's declaration in a program text. References to a function *f* can usually only occur in places where the environment in which *f* is to be executed is still in existence. In this case, an explicit closure isn't necessary.

However, in languages in which functions are so-called "first-class citizens" – in which their computation entities are values that can be assigned to variables and passed as parameters – references to a function *f* can occur anywhere, possibly outside the textual scope in which *f* is declared. In this case we must bundle *f*'s environment along with *f*, sometimes at run-time, so that when *f* is called, its free references will be properly bound. An example is given in the next section.

2.5.2. Lexical vs. Dynamic Inheritance

The order in which contours are searched normally is tied closely to a program's structure – either its syntactic structure or its run-time structure. The former case is called *static inheritance* or *lexical inheritance*[12], because name resolution is done solely on the basis of the program's unchanging (thus the word "static") lexical and syntactic structure. The latter case is called *dynamic inheritance*[13], because it is based on the changing (thus the word "dynamic") run-time environment of a program.

The most common form of static inheritance is conventional block structure as described earlier; the search is from inner to outer scopes based on the program's syntactic structure.

---

[11] More precisely, one must define the visible set for any point in a program where a reference may occur.

[12] These are usually called static or lexical scoping, but we have chosen not to use the term "scoping" because it is yet another use of "scope", and because it conveys little meaning in this case. The term *binding* is also used in this context, as in "static binding".

[13] or sometimes *dynamic scoping* or *dynamic binding*.

Dynamic inheritance is best illustrated by LISP [Clark and Weisman 1967]. In LISP, binding of variables is based on the order of function activations – contours for all functions activated but not terminated are searched in reverse order of invocation until either a binding is found, or all contours have been searched. Multiple activations of a function declaring a particular name "D" may exist, but a reference "D" will always get the binding from the most recent activation. Free references in a function *f* will not in general bind to the same instance of an entity on different invocations of *f*.

Other forms of run-time environment-based binding are possible, but this call-order based binding is what is conventionally meant by "dynamic binding" or "dynamic scoping." When a reference is resolved using dynamic inheritance, we say that the name is "dynamically bound." If the reference is resolved using lexical inheritance, we say that the name is "lexically bound."

We illustrate this notion with the example in Figure 2.9. If static inheritance is used, then the reference to *A* in procedure *P* will bind to the first declaration of *A*, in the textually enclosing block. With dynamic inheritance, the reference to *A* in procedure *P* will bind to the declaration of *A* in the most recently activated scope (at run time) containing a declaration of *A*. Since the order of execution is: entry to the main program, followed by a call to *Q*, followed by a (nested) call to *P* at label *L2*, the first time *P* is invoked the reference to *A* in procedure *P* will be bound to the declaration of *A* in *Q*. After the return from *Q*, there is a call to procedure *P* from the main program (at label *L1*). On this invocation (the second) of *P*, the reference to *A* in procedure *P* is bound dynamically to the first declaration of *A*. Although the sequence of bindings in this example can be determined by static analysis of the program, that is not true in general.

Lexical and dynamic inheritance both have advantages, but lexical inheritance is now generally preferred in most cases. Some compiled versions of LISP have used a single level of lexical inheritance in some cases for many years, even at the expense of causing compiled and interpreted versions of the same program to have different semantics: a reference can bind to different

```
declare
  A  : integer;          -- first declaration of A

  procedure P is
  begin
    A := 1;
  end;

  procedure Q is
  declare
    A  : integer;        -- second declaration of A
  begin
    L2:P();              -- call P
  end;

begin
  L0:Q();                -- call Q
  L1:P();                -- call P
end;
```

Figure 2.9: Lexical vs. Dynamic Inheritance

entities depending on whether the reference occurs in interpreted or compiled code. Recent dialects of LISP, such as Scheme [Steele Jr. and Sussman 1978a], T [Rees *et al.* 1984], and Common LISP [Steele Jr. 1984], primarily use lexical inheritance. Most recent dialects also guarantee that the choice between dynamic and lexical inheritance does not depend on whether the program is interpreted or compiled, so that the interpreted and compiled versions of a program have identical semantics, except for time considerations.

The reason for this acceptance of lexical inheritance is that it provides better *referential transparency*. This term was first used by Whitehead and Russell [Whitehead and Russell 1927] and later by Quine [Quine 1960] with different (but related) meanings, but Stoy [Stoy 1977] used it with the meaning we intend here. In Stoy's words, we have referential transparency if:

> The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs.

Steele and Sussman [Steele Jr. and Sussman 1978b] give a definition that is more useful in the context of programs and programming (although the definitions are essentially identical). Referential transparency requires that:

> the meanings of parts of a program be apparent and not change, so that such meanings can be reliably depended upon ··· the external behavior of a module should be independent of the choice of names for its local identifiers.

Dynamic inheritance destroys referential transparency, because the meanings of free references, and thus the meanings of the functions in which they occur, are dependent on the calling environment. The variable to which a name within a function f is bound will vary from one call of f to another, so the meaning of f will vary from one call to another.

Dynamic inheritance does have a part to play in developing modular programs, as the example in Figure 2.9 illustrates. The example is based on a Scheme example from Steele and Sussman [Steele Jr. and Sussman 1976, p. 18], but is written in an extended Pascal syntax so that it will be easier for readers unfamiliar with LISP to understand. The first extension is the ability to have a function as the return value of a parameter, creating a closure (as discussed above) in the process. The other extension is dynamic inheritance, which can be forced by preceding a reference by the keyword **dynamic**. Any such reference is resolved using dynamic inheritance, while all other references are resolved using lexical inheritance, as in standard Pascal.

In the example, the function *sqrt* is defined elsewhere, and computes the square root of its argument to a given tolerance *epsilon*. *epsilon* is normally defined in the global scope of the program. *sqrt* references *epsilon* using the **dynamic** keyword, so that it always gets the dynamically innermost binding of *epsilon*. The purpose of the function *GenerateSqrtOfGivenExtraTolerance* is to produce a new version of the *sqrt* function that uses a different tolerance, *factor*×*epsilon*, without modifying the definition of *sqrt* used elsewhere in the program, and without modifying or copying the value of the global definition of *epsilon*, which could affect other parts of the program.

GenerateSqrtOfGivenExtraTolerance defines two auxiliary functions. *DoRebind* takes two arguments: x, which is passed to *sqrt*, and the new value of *epsilon*. When *sqrt* is called within the body of *DoRebind*, the *epsilon* which is the formal parameter of *DoRebind* is the *epsilon* bound to the dynamic reference to "epsilon" within *sqrt*. *NewSqrt* calls *DoRebind*, passing it the value of x and the new value of *epsilon*, computed as the product of the dynamically inherited *epsilon* and *factor* (the argument to *GenerateSqrtOfGivenExtraTolerance*). *GenerateSqrtOfGivenExtraTolerance* simply returns the closure of *NewSqrt*, as in the calls defining *sqrt1* and *sqrt2*. When a closure of *NewSqrt* is invoked, for instance *sqrt1*, it will compute the new *epsilon* using: (1) the *factor* passed to *GenerateSqrtOfGivenExtraTolerance* at the definition of the value of *sqrt1*, and (2) the dynamically inherited *epsilon* at the point of the call to *sqrt*, and *sqrt1* will

```
type    RealToRealFunction = function ( real ) : real;

{ return type of this function is a real function that takes one real argument }
function GenerateSqrtOfGivenExtraTolerance ( factor : real ) : RealToRealFunction;

        function DoRebind ( x, epsilon : real ) : real;
        begin
                DoRebind := sqrt (x)
        end;

        function NewSqrt ( x : real ) : real;
        begin
                NewSqrt := DoRebind (x, dynamic epsilon * factor)
        end;

begin
        GenerateSqrtOfGivenExtraTolerance := NewSqrt
end;

procedure CreateNewSqrts;
var
        sqrt1, sqrt2 : RealToRealFunction;
        answer1, answer2 : real;
begin
        sqrt1 := GenerateSqrtOfGivenExtraTolerance (0.1);
        sqrt2 := GenerateSqrtOfGivenExtraTolerance (0.01);
        answer1 := sqrt1 (2);
        answer2 := sqrt2 (2);
end;
```

Figure 2.10: Dynamic Inheritance and Modular Programs

compute the square root (via *DoRebind*) using the new value of *epsilon*. The effect is to temporarily change the binding of *epsilon* seen by the old function *sqrt*.

Since *sqrt* uses *epsilon* as a free variable, with lexical inheritance, *sqrt* will always use the lexically global definition of *epsilon*. However, with dynamic inheritance, *sqrt* uses the temporary binding of *epsilon* created by the call to *DoRebind* from the closure of *NewSqrt*. So, the reference to "epsilon" must be dynamically bound in *sqrt*, or the new square root function will not work as desired. One alternative way to describe the same computation would be to modify the lexically global value of *epsilon*, and to change it back after computing the square root. This works, but is clumsy. Another alternative would be to pass *epsilon* everywhere as an argument, requiring that *epsilon* be passed through functions that otherwise had no use for *epsilon*, with an obvious decrease in readability.

Different meanings also occur dependent on whether the reference to "epsilon" in *NewSqrt* is resolved using dynamic or lexical inheritance. With dynamic inheritance, as shown in the example, the value of *epsilon* seen by *sqrt* depends on the value of the dynamically inherited *epsilon* at the point of a call to *sqrt1* or *sqrt2*. That is, the value of *epsilon* used by *sqrt1* and *sqrt2* is computed relative to the value of *epsilon* at the point of call. If lexical inheritance is used for the reference to "epsilon" in *NewSqrt*, then the reference will bind to the global binding of

"epsilon," regardless of any other bindings of "epsilon." The value of *epsilon* computed in *sqrt1* and *sqrt2* will depend on the value of the global *epsilon* at the point of a call. Since the reference to "epsilon" in *sqrt* must be dynamically bound, the most consistent choice is the former, using dynamic inheritance for all references to "epsilon."

On the other hand, *factor* must be lexically bound, in order for the reference to "factor" in *sqrt1* to bind correctly to the *factor* parameter in the instantiation of *GenerateSqrtOfGivenExtra-Tolerance* active when *sqrt1* was created as a closure of *NewSqrt*. Remember that a call to *sqrt1* will occur both lexically and dynamically independent of *GenerateSqrtOfGivenExtraTolerance*, so the only way to find the appropriate *factor* is using lexical inheritance at the point where the reference to "factor" is written: inside the body of *NewSqrt*. So, we need both lexical and dynamic inheritance in order to implement this function cleanly.

Dynamic inheritance is also sometimes used for changing standard input and output files temporarily, for exception handling, and for situations in which bindings are determined by run-time searches, such as in data bases. For such reasons, modern LISP dialects [Rees *et al.* 1984; Steele Jr. and Sussman 1978a; Steele Jr. 1984; Weinreb and Moon 1981] have provided some mechanism for using dynamic inheritance to resolve references when desired. Dynamically bound variables in most such dialects are called "special" variables. EL/1 [Wegbreit 1971], though primarily statically bound, also has a feature that allows the programmer to force dynamic inheritance.

Lexical and dynamic inheritance also have their respective efficiency advantages, depending on the particular program and the hardware on which it is run.

Steele claims [Steele Jr. 1976, p. 12] that the primary reason for the use of dynamic binding in LISP was the introduction of stack hardware about the time of early LISP development, though this was not pure cause and effect; rather, each phenomenon influenced the other[14].

2.5.3. Dynamic Type Binding

In addition to the diversity of situations in which name-entity binding occurs, there can be variations in the ways in which various attributes of an entity are bound to the entity (and thereby to the name). As discussed in §2.2 the binding of an object to an entity often occurs separately from the binding of entity to name. It is also possible that the binding of type to an entity denoting type values can occur separately from the binding of name to entity.

As an example, both EL/1 and ALGOL 68 allow entities of type pointer, where the type of the object indirectly referenced is not determined until run-time. In ALGOL 68 the set of possible object types is given, perhaps by using a **union**; in EL/1 the set of possibilities need not be given explicitly. Thus, even though some of the benefits of static visibility rules are realized, others, such as compile-time type checking, are not.

2.6. Explicit Visibility Control

The simple block-structured paradigm is extended in a number of ways, in order to provide more controlled access to names in programs. The ways in which access is explicitly controlled are summarized in the following sections. We call this *explicit visibility control.*

The various forms of explicit scope control were introduced in languages to improve the programmer's control over the use of names – to make them visible only where they are needed,

[14] Indeed, McCarthy states that, early in the development of LISP, a LISP function definition was brought to him that didn't work right because lexical inheritance was needed, while LISP provided dynamic inheritance. McCarthy considered this a bug, but the bug was fixed by adding FUNARGs, which allowed one to obtain lexical inheritance when needed, but did not alter the basic dynamic inheritance discipline of LISP [McCarthy 1981].

and to allow logical grouping of related declarations.

2.6.1. Named Scopes

In most of the examples presented thus far, scopes were either un-named, or were procedures or functions whose names were used solely as a means to call them. Blocks can have names in some programming languages, but these names usually have limited use, for instance to assist in the alteration of flow of control, as with an **exit** statement.

Named scopes can also be used to provide controlled visibility of bindings, in addition to the usual grouping function of scopes. Recall the example of the Pascal record of type *Date* in Figure 2.4. The record is a scope that groups together the related entities *day*, *month*, and *year*. Record values can be manipulated as a single object when the programmer is not concerned with the individual fields. Alternatively, the fields can be accessed individually. In Pascal, objects of record type are always variables, and the fields are also always variables. However, in Mesa and in Ada the entire record object (but not individual fields) can be a constant. In Euclid, either the entire record object or individual fields can be constants.

2.6.2. Qualified References

Qualified references are used to access the individual fields of named scopes. A qualified reference (for example the reference *today.day* using the declarations in Figure 2.4) is constructed by composing a scope name with the name of an entity bound within that scope. The use of qualified references is a form of explicit scope control, and is one of the ways in which named scopes are used. A field within a record only becomes visible within the context of an explicit reference to the record itself.

So, there are two forms of named access or reference. They are:

Simple
: Only the name of the binding being referenced need be given. The appropriate binding is determined by the scope rules of the language.

Qualified
: The name of the scope along with the name of a binding directly visible within that scope's contour[15] must be given. The reference to the scope is first resolved. Then, the reference to the entity declared within the scope is resolved by looking for a binding within the scope's contour. Whether this type of reference is possible depends on the language under consideration, and on the nature of the scope (for example procedure, block, module, etc.). Also, some entities within a particular scope may be accessed in this manner while others may not[16].

Qualified references allow the hierarchical organization of names, while permitting access to names on a lower, more deeply nested level. For example, the fields of a record are not normally directly visible. Thus, the names of those fields do not clutter the name space where the record is declared, but the fields still can be accessed with a qualified reference. In addition, several record variables with the same type may be visible at the same point. The names of their fields would clash if they were not hidden from direct access. In Ada, one can use a qualified

---

[15] For PL/I and COBOL records, the entity being named need not be *directly* contained within the named scope's contour, but rather can be nested one or more levels within it. The only requirement is that the reference be unambiguous. This is a fairly difficult feature to implement, though several algorithms exist [Busam 1972; Gates and Poplawski 1973].

[16] In Ada package specifications, all entities declared before the keyword **private** are visible via qualified reference, while all entities after it are "private" to the package, and not visible.

reference to refer to a binding in a named outer scope when that binding would otherwise be hidden by a simple reference to an intervening redefinition.

2.6.3. Modules

A generalization of the notion of a record is to allow components (that is, fields) other than variables or constants. This idea leads to the concepts of *modules*, *packages*, and *classes*, which are discussed in this section, and which we will refer to generically as modules, unless we are talking about a feature of a specific language (for example, packages are Ada's version of modules).

A *module* is very closely related to a record. The primary distinction is that the word ''module'' is usually used to mean something more general – a module is typically a named scope that may contain types, functions, variables, other modules, or (usually) any entity definable in a particular language, all defined within that scope.

In Modula-2, there is only a single instance of each module, although the module may define a Pascal-like type used in the creation of multiple entities, as will be illustrated in §2.6.7. A language feature more similar to record types is to treat a collection of entities, including variables, types, functions, etc. as a type, and to allow the programmer to create instances of that type. In Smalltalk, instances can be created dynamically. However, in some other languages, for example Euclid, instances are statically declared as variables of that type.

Modules often have special operations associated with them for controlling visibility. These will also be discussed in §2.6.7.

2.6.4. Named Inheritance and Smalltalk-80

In all examples presented so far, the search for bindings of a name has been based solely on the program's syntactic structure or its run-time structure (the run-time environment). For example, with lexical inheritance, if no binding of a name is found in the current scope, the next lexically enclosing contour is searched, and so on. Dynamic inheritance is similar except that the next dynamically enclosing contour is searched next.

An alternative is for the programmer to state explicitly within the body of a scope what scope should be searched next, by specifying the name of that scope. We call this alternative the use of *named inheritance*. An example of a language in which this concept is heavily used is Smalltalk-80 [Goldberg and Robson 1983].

Before we give an example from Smalltalk-80, we must give a brief introduction to the terminology of Smalltalk so that the reader can understand the example.

The primary notion in Smalltalk corresponding to a type is a *class*. A typical class consists of a collection of variables, which can be thought of as record fields, together with operations, called *methods*, defined within the range of those variables. An object of a given class, called an *instance*, is analogous to a record variable with associated local functions. A method is invoked by sending a *message* naming that method to a class instance. This corresponds closely to calling a procedure in other languages. A class can be named, providing a named scope.

Named scopes are particularly useful in Smalltalk, because a new class can be defined as an extension of an old one called its *superclass* by means of named inheritance. The new class is called a *subclass* of its superclass. A superclass plays the role of an outer scope in the sense that all variables in methods defined in the superclass become variables and methods in the new class except those that are locally redefined. The user creates instances of a class, just as a user creates variables of a given type. Named inheritance is, in essence, an extension of block structure in which bindings can be inherited from an arbitrary named scope instead of from only the textually enclosing scope.

The search to resolve a reference by a class instance always begins in the instance's class. If a class c redefines a variable v defined in its superclass, a method defined in the superclass will use the v defined local to c when invoked from within an instance of c. Thus the binding is determined by the class of the invoking instance, rather than by the context of the method definition. This convention should not be confused with dynamic inheritance, because the chain of inheritance is determined by the chain of superclasses, not by a call stack. However, it is also different from conventional lexical inheritance, because the point of search is determined by the instance's class, not by the class where the referencing method is defined.

The classes in Figure 2.11 illustrate named inheritance in Smalltalk-80. These are skeletons of actual classes in the Smalltalk-80 system. *Streams* are ordered collections that have a current position associated with them. *PositionableStream* is assumed to be a subclass of *Stream*. Hence *WriteStream* is an indirect subclass of class *Stream* (via *PositionableStream*). The intention is that instances of *WriteStream* are ordered character collections that can be written to, but not read from. *ReadWriteStream*'s instances can be read from as well as written to. The class *WriteStream* defines (among others) the method *nextPut*, which takes a character argument and adds it to the collection. Class *ReadWriteStream* defines the method *next*, which returns the next character from its collection. If an instance of *ReadWriteStream* receives the message "next", it simply invokes the method *next* defined in its class description.

However, suppose that instance receives the message "nextPut: a". Since class *ReadWriteStream* does not define that message, the class's superclass in searched, then the superclass's superclass, and so on. In this case, the method *nextPut:* is defined in *ReadWriteStream*'s superclass (*WriteStream*), so that method is invoked.

This is what we call named inheritance – the search for a binding of a name is specified by giving a named scope as the next to search. In Smalltalk, these scopes are class descriptions. A class may either implicitly pass a message on to its superclass (simply by not handling the

---

```
class name WriteStream
superclass PositionableStream
 ...
instance methods

  nextPut: aCharacter
        -- adds argument aCharacter to instance's collection
        -- at current position
 ...

class name ReadWriteStream
superclass WriteStream
 ...
instance methods

  next
        -- returns next character in instance's collection
 ...
```

Figure 2.11: Named Inheritance in Smalltalk-80

---

message locally) or explicitly specify that the immediate superclass's method is to be used instead of a method in the current class. Remember that block structure results in inheritance also. Named inheritance is just an extension of block structure.

2.6.5. Multiple Inheritance

In all previously discussed methods of visibility control, a contour could have only a single parent contour, which was the next contour to be searched. A scope inherited bindings only from a single parent scope, whether this scope was defined lexically, by the dynamic environment, or by a name (as in Smalltalk-80). A generalization of this situation is to allow a scope to inherit bindings from more than one parent. This generalization is called *multiple inheritance*.

Language designers have considered a variety of forms of multiple inheritance, differing in the meaning of a program in which a name is bound in more than one of the inherited scopes. One possible extension of Smalltalk-80 is described by Borning and Ingalls [Borning and Ingalls 1983]. In that proposal, a class can specify several superclasses instead of just one, inheriting the methods of all the superclasses. All superclasses of a class may have to be searched in order to find a method. Smalltalk-80 has a class *ReadStream* in addition to the classes *WriteStream* and *ReadWriteStream* defined above. An instance of *ReadStream* can be read from, but not written to. *ReadStream* defines the method *next*, as does *ReadWriteStream*. With multiple inheritance, Borning and Ingalls suggest defining *ReadWriteStream* as follows:

```
class ReadWriteStream
superclass ReadStream, WriteStream
  ...
instance methods

        -- no definition of "next" required here
        -- "next" inherited from ReadStream
        -- "nextPut:" inherited from WriteStream
```

The method *next* no longer must be defined in *ReadWriteStream* because it is inherited from *ReadStream*. *nextPut* is still inherited from *WriteStream*. This is not possible without multiple inheritance, because neither *ReadStream* nor *WriteStream* can properly be a subclass of the other.

In addition to this implicit search mechanism, Smalltalk-80 extended with multiple inheritance allows the programmer to select a method in a particular class using a qualified reference: *class-name.method-name*.

The multiple inheritance scheme is more complicated than that in the ALGOL languages, but it still fits within the search model of visibility, because binding resolution can still be done by a well-defined search through scopes (the class definitions). If there is more than one binding matching a reference, the search rules must specify which one to select, usually by defining a specific order for searching parent contours and using the first matching binding found.

LISP Machine LISP [Weinreb and Moon 1981] provides another example of multiple inheritance. Its multiple inheritance feature is called *Flavors*, and is more general than the proposed definition of multiple inheritance for Smalltalk-80[17]. Other systems [Bobrow and Stefik 1982; Curry *et al.* 1982; Moon 1986; Schaffert *et al.* 1986] have also included multiple inheritance.

[17] A more recent version of LISP Machine LISP contains a significantly revised version of Flavors [Moon 1986; Symbolics, Inc. 1986].

2.6.6. Open and Closed Scopes

Normally, the declarations in the scopes either statically or dynamically enclosing a given scope are visible within that scope (except possibly for those hidden by more local declarations). That scope is known as an *open scope*, and is the only kind present in (for example) Pascal, Algol 60, and LISP.

A *closed scope* is one in which declarations appearing outside the scope are not normally visible within the scope. Modules in Modula [Wirth 1977] and Euclid [Lampson *et al.* 1981] are examples of closed scopes. Both of these languages also have open scopes.

The following example illustrates open and closed scopes. The syntax belongs to no particular language. In the example, modules (delimited by **module** ... **end**) are closed scopes and blocks (delimited by **declare** ... **begin** ... **end**) are open scopes. The comments in Figure 2.12 illustrate which references are legal, and which are not, because of the presence of nesting and open and closed scopes.

A scope may be closed relative to some declarations, and open for others. In Euclid, one can declare a binding to be *pervasive*, which causes that binding to be visible within all closed scopes (in addition to open scopes) textually contained within the scope where the binding is declared.

In general, a language may contain any number of scope classifications and entity classifications, and each scope classification may have different rules concerning to which entity classifications it is closed, and to which it is open. However, no existing language approaches

```
module M1
  A   : integer;

  module M2
    B   : integer;

    declare
      C   : integer;
    begin
      C := 1;  -- legal, C declared locally
      B := 1;  -- legal, B visible since a block is an open scope
      A := 1;  -- illegal, since M2 is a closed scope
    end

    C := 1;          -- illegal, C is declared in an inner scope
    B := 1;          -- legal, B declared locally
    A := 1;          -- illegal, since M2 is a closed scope
  end M2;

  C := 1;            -- illegal, C is declared in an inner scope
  B := 1;            -- illegal, B is declared in an inner scope
  A := 1;            -- legal, A declared locally
end M1;
```

Figure 2.12: Open and Closed Scopes

this level of generality.

When the concept of nested scopes was first introduced, and for a long time afterwards, only open scopes were included in languages. Closed scopes eliminate the unrestricted accessibility of global bindings. However, closed scopes alone are not very useful, because *some* visibility across the boundary of the closed scope is necessary to provide for communication between the interior and the exterior of the closed scope. The **import** and **export** statements, discussed in the next section, are used with closed scopes to provide *controlled* visibility across this boundary.

2.6.7. Imports and Exports

An **import**[18] statement causes a binding or bindings visible in another scope to become visible in the scope immediately containing the **import** statement. The effect of an **import** is sometimes to add the binding to the set of bindings associated with the current scope's contour. Usually, an **import** applies to a name visible in the immediately enclosing scope. An example illustrating the effect of the addition of an **import** statement to the example in Figure 2.12 is given at the end of this section.

Alternatively, an **import** statement might name a scope, and cause the import of all the bindings local to that named scope into the current scope. This kind of **import** statement is discussed in the next section.

The primary purpose of the **import** statement, used along with closed scopes, is to allow (or require, depending on the circumstances and viewpoint) the programmer to state exactly what bindings are needed within a particular scope, instead of having all global bindings visible. Wulf and Shaw [Wulf and Shaw 1973] argue that the automatic visibility of all globals can make programming more error-prone. The degree of visibility control in conventional block structure is insufficient, so bindings sometimes must be visible to procedures in which they are not needed. Readability of block-structured programs can also suffer, because procedure headers can be separated from their bodies by a large distance. The features of block structured access can cause these and other problems [Clarke *et al.* 1980; Hanson 1981; LeBlanc and Fisher 1979].

The **export**[19] statement is the converse of the **import**. It causes a binding in the current scope to be made visible in the enclosing scope, normally without the use of a qualified reference, although in some languages that option is under programmer control. The exported binding must either have been declared in the current scope, or exported to the current scope from an inner scope. This feature is used in Modula, Euclid, and other languages to allow packages to make locally defined variables and procedures visible, and thus useful, to the enclosing environment.

Figure 2.13 is a modification of Figure 2.12, with the addition of an import of *A* and an export of *B*. The statement "**import** *A*" causes all references to *A* inside *M2* (including inside the block) to become legal (following the semantics of Euclid and Modula). The statement "**export** *B*" causes *B* to be visible within the body of *M1* (the statements following *M2*).

One of the primary uses of the **export** mechanism is to facilitate *encapsulation* for the purpose of *information-hiding* [Parnas 1972]. For example, if a type can be a class-like collection of variables, local types, functions, etc., then by exporting selected functions only and allowing those functions to operate on variables of the type, *abstract data types* [Liskov and Zilles 1974] can be described. Bindings not explicitly exported are normally not visible outside the module where they are declared. This encourages the use of information-hiding by enabling the programmer to distinguish explicitly between operations, variables, and other entities available outside the module, and those only intended to be used locally within the module. The representation of

[18] The keyword **use** is used in Modula, though Modula-2 uses the more commonly adopted keyword **import**.

[19] The keyword **define** is used in Modula, though Modula-2 uses **export**.

```
module M1
  A   : integer;

  module M2
    import A;
    export B;
    B   : integer;

    declare
      C   : integer;
    begin
      C := 1;  -- legal, C declared locally
      B := 1;  -- legal, B visible since a block is an open scope
      A := 1;  -- now legal, because of "import A"
    end

    C := 1;              -- illegal, C is declared in an inner scope
    B := 1;              -- legal, B declared locally
    A := 1;              -- now legal, because of "import A"
  end M2;

  C := 1;                -- illegal, C is declared in an inner scope
  B := 1;                -- now legal, because of "export B"
  A := 1;                -- legal, A declared locally
end M1;
```

Figure 2.13: Import and Export Statements

the abstract objects and the implementation of the abstract operations via function definitions can be hidden from the users of that type. Parts of the program outside that module can be restricted to using the operations defined by the abstract data type, so the integrity of objects of that type is protected.

A similar capability is provided in Ada, without an explicit **export** construct, by a convention that package specifications export all bindings except those declared to be *private*. Private bindings are visible only within the defining package. Visibility of the non-private bindings of a package can be achieved with a **use** statement, discussed in the next section.

A more general form of **export** was introduced in the language L [Cormack 1983]. In L, one is permitted to define the visibility of a name to be either *local, containing, intermediate, global*, or *permanent* when the name is declared. The visibility choice determines how far out from the current scope the new binding is visible. *Local* visibility corresponds to the normal case in other languages: the declaration is visible in the current scope, and possibly in scopes enclosed by the current scope. *Containing* visibility corresponds to a conventional **export**, making the binding visible in the immediately enclosing scope. The name of an enclosing scope must be given when declaring a binding with *intermediate* visibility – the binding is then visible as far outward as that scope. If a binding has *global* visibility, it is visible in all scopes between the global scope and the current scope, inclusive. *Permanent* visibility means that the binding is visible outside of the program, and corresponds to a file name.

Alexander Wolf [Wolf 1985; Wolf *et al.* 1986b] presents a model for controlling visibility based on the *provision* and *requisition* of access. Using this model, for a binding b to be visible within a scope S, b must provide access to S, and S must request access from b. This provision and requisition may be either explicit or implicit, depending on the particular programming language. An open scope, as in ALGOL 60, requests everything visible in the scope enclosing it. A binding in ALGOL 60 is provided to the scope where that binding is declared and to all scopes nested within that scope. An **import** statement in conjunction with a closed scope requests specific bindings while not requesting others. An **export** statement provides specific bindings to the parent scope and all scopes enclosing the scope containing the **export**. The problem with import and export is that they are not precise enough to represent all possible provision and requisition relationships desirable in a program. One can define statements, based on the concepts of provision and requisition, that allow the required level of precision. For example, one can define a **provide** statement that provides access of a particular binding to a specific named scope or scopes. Likewise, one can define a **request** statement that requests access only to specific bindings. With such a fine level of control, one can express any desired provision and requisition relationship. Wolf defines an extension of Ada called Ada/PIC that includes such statements.

As with **import** statements, **export** statements could be used in languages with dynamic inheritance, but we know of no example of a language where this is done.

2.6.8. Opening of Scopes

The **import** statements discussed in the previous section only imported individual bindings, requiring that each binding to be imported had to be explicitly named. This section describes a form of **import** that makes all of a named scope's bindings visible in the scope where the **import** occurs.

Usually, the **import** statement names a module (or package or record, etc.), and the bindings local to the named module become visible by means of a simple reference in the scope of the **import**, where otherwise the bindings in question would either not be visible at all, or visible only by means of a qualified reference. We call the process of importing all the bindings of a scope *opening a scope*, because the barrier to visibility formed by the scope boundary is being opened. Note that an "open scope", which was defined earlier, is not the same as a scope that has been "opened", in the sense of the current discussion.

Since such an **import** statement involves an indiscriminate importation of bindings, the chance of a clash between a locally declared binding and an imported one is not insignificant. Defining such a clash to be an error results in two problems:

(1) The user of a module must be aware of all of the bindings exported by a module (and thus imported by opening that module), and avoid using any clashing names, even if the user does not use some of the exported bindings.

(2) The implementor of a module can not add to the interface of the module (the bindings exported by the module) without fear that programs that use the module will become incorrect because of name clashes. One can argue that such an addition to a module interface should not affect any programs that use the module.

One solution to this problem is to define a rule for resolving such clashes between locally declared bindings and imported bindings, usually in favor of the locally declared binding.

The Ada **use** statement is an example of a scope-opening **import** statement. An Ada scope can "use" one or more packages, importing the non-private bindings from the named packages. When two or more bindings clash, the following rule determines which bindings are visible in a scope S affected by **use** clauses:

(1) The set of "potentially visible" bindings consists of all non-private bindings of all packages appearing in **use** statements affecting S.

(2) A potentially visible binding b_1 is not visible if another binding b_2 clashes with b_1, and b_2 is visible within S in the absence of any **use** clauses.

(3) If two potentially visible bindings clash, then all potentially visible bindings with the same name are not visible. That is, all potentially visible bindings of a name are visible if they are all overloadable, and none of them are visible otherwise (assuming part (2) does not apply).

(4) All other potentially visible bindings are visible.

This type of **import** statement eliminates the need for specifying long lists of names in an **import**, but discourages the programmer from naming exactly what bindings are needed in the current scope.

Opening a scope has at least two purposes. The first is to allow a temporary shorthand notation for a set of entities that will be referenced often over some bounded region of a program. The second purpose is efficiency: in the absence of optimization, computing the address of a record field involves first computing the address of the record, and then augmenting that address by the offset of the field. When a record scope is opened, the implementation can factor out the access to the record rather than computing it for each field reference.

When a module is used to implement an abstract data type, an **import** statement that imports all bindings from that named module also fails to protect the details of the representation of the abstract data type. The **use** statement in Ada does not have this problem because private declarations are not visible outside of the package under any circumstances. The **use** statement in Ada causes only non-private declarations in a named package to become visible in the scope of the **use** statement. Usually, this problem is avoided by making only explicitly exported bindings visible outside a module, whether or not those bindings must be explicitly imported to be used.

There are two ways to view scope opening as thus far described. One is as an expanded form of the basic **import** statement, where the imported bindings are copied into the importing scope. Scope opening can also be viewed as a form of named inheritance, since it makes the bindings of some other, named scope directly visible. One view may be more suitable for describing and understanding a particular language or language construct, while the other view may be more suitable for another language or construct.

Pascal has a different way of avoiding errors when opening a scope. Pascal's version of scope opening is the **with** statement, which is used to open a record scope. When a record scope is *opened* in Pascal, a new local scope is created in which all bindings declared in the opened scope are visible without qualification. Since the bindings of the opened scope are the only bindings in the new scope, there is no possibility of an error due to clashing bindings. Another point of view is that the opened scope is added as a new scope at the current location, nested within the current scope. For instance, (using the same declarations of the record type *Date* and the variable *today* as in Figure 2.4), the example in Figure 2.14 would be a valid Pascal code segment. The **begin...end** comprises the body of the **with** statement, defining the scope in which the fields of *today* can be referenced.

The **open** statement in Mesa can be used to open both records and modules.

2.6.9. Visibility Control in SETL

An early version of the language SETL [Schwartz 1973] had visibility control rules which were probably more general than any other existing language. One feature of the 1973 version was the use of scope level numbers, which could be explicitly attached both to scopes and to variables. A scope number attached to a variable declaration meant that the variable was visible in

```
type Date =
        record
                day     : integer;
                month   : 1..12;
                year    : integer;
        end;

var today :     Date;

begin
  today.day := 3;
  { "day" by itself cannot be referenced here - it must be preceded   }
  { by the scope name "today." (unless another declaration of "day" exists, of course }
  ...
  with today do
  begin
        day := day + 1;
        { "month" and "year" can also be referenced here without qualification }
  end;
  ...
end;
```

Figure 2.14: Scope Opening

all inner scopes whose level numbers (assigned to each scope by the programmer) were no greater than the level number of the variable. The effect was that a scope could be closed to certain variables, and open to others, determined by the level number of the scope and the level numbers of the variables. This feature is a generalization of the closed scope features discussed earlier, where the number of scope classifications is program-dependent, rather than just language-dependent, since it is determined by how many different level numbers are used. Also, the "classification" of an entity, where its visibility is concerned, is determined by its associated scope level number. This feature can be handled in the Search Model by attaching a *search predicate* to a search, which represents a condition that must be satisfied by any binding found, and which may be augmented with additional conditions at each step in the search. The predicate in this case requires that the level number of any binding found be greater than or equal to the level number of all scopes encountered in the search.

That version of SETL also had a very general **import**-like statement, that could make bindings in parallel scopes and even more deeply nested scopes visible, as long as the imported name was in the same procedure as the "import" statement. Precise details about this and other SETL features are beyond the scope of this dissertation. SETL now has more conventional visibility control rules [Dewar 1979; Schwartz *et al.* 1986]

2.6.10. Rebinding Entities

In all of our previous discussion, each entity has only had one name bound to it. In other words, there has been only one binding of each entity. However, it is possible for more than one binding of a particular entity to exist.

If more than one binding of an entity is visible at the same point in a program, one says that the entity is *aliased*. Often, aliasing is confusing and undesirable, because a programmer usually expects different names to refer to different objects.

Aliasing can also occur at different steps in the process of mapping a name to a storage location. For example, a global variable *g* and a formal parameter *f* of a procedure *P* are separate entities, but they will both refer to the same storage location if *g* is passed to *f* in *P*, and *g* is visible within *P*.

Lower-level versions of aliasing are possible in FORTRAN and PL/I. The COMMON statement in FORTRAN permits the programmer to cause different blocks of variables (with possibly different names and types) to overlay the same storage area, resulting in aliasing at the storage level. The PL/I **defined** clause causes a new variable to occupy the same storage area as a preceding variable. However, since this dissertation is not concerned with the steps of the name-value binding that introduce this type of aliasing, it will not be discussed further.

A comprehensive discussion of aliasing, particularly as it relates to visibility control issues, is contained in [Schwanke 1978].

So, we return now to multiple bindings of entities. The usual method is to bind a new name to an entity by referencing an old name already bound to that entity. Euclid is used to illustrate this method: In Euclid, an entity can be re-named using a **bind** statement. Within the effective range of the **bind** statement, the old binding of the entity may not be referenced, and the entity must be referenced by the new name. Referencing the old binding is forbidden in order to eliminate the aliasing that otherwise could be introduced. Euclid was designed so that all forms of aliasing could be detected and flagged as errors[20].

For example, in Euclid one could create the capability of referencing the record field *today.day* with the simple name *day* as within the **with** statement in Figure 2.14 by including

```
bind var day to today.day
```

in the declarations section of the scope in which the shortened reference form is desired. The difference from Pascal is that *today.day* is still a legal reference within the **with** statement in Figure 2.14, but *today.day* is not a legal reference within the range of the above **bind** statement in Euclid.

Other forms of the Euclid **bind** statement include:

```
bind sixthEntry to a(6)          -- a(6) is an array reference
bind t to today
```

Ada and ALGOL 68 each have rebinding constructs that provide a similar capability. As in Euclid, renaming can introduce aliasing.

Another version of re-binding is the **bind** statement in Bliss [Wulf *et al.* 1971], which simply binds a name to the value of an expression at block entry. The Bliss **bind** defines a named constant whose value is recomputed each time the block is entered. The expression can designate an address – *e.g.* *a*(6) as above, or the address of any other storage entity, so the Bliss **bind** statement can be used for the same purposes as the Euclid **bind** statement. The Bliss version simply operates on a lower level (storage addresses) when used to refer to entities.

The reasons for giving an entity a new name are similar to those for opening a scope. A new name is created in order to create a temporary shorthand notation for an entity that will be referenced often. The new binding may also be introduced for efficiency: if referencing an entity involves a non-trivial computation (such as in array referencing or field selection), then the computation will only be done once if the entity being referenced is given a new, simple name.

---

[20] In Euclid terminology, the old binding is still *known* (visible) although it may not be referenced. It is known so that it can't be redeclared (as discussed earlier, a known (visible) Euclid declaration can't be hidden by redeclaring a name; any attempt to do so is an error).

2.6.11. Summary of Explicit Visibility Control

All the different operations for explicit scope control actually consist of only a few basic mechanisms, described here in terms of the Search Model. The first is copying bindings from one contour to another. The effect of **imports** and **exports** can be represented by copying bindings, or by adding an explicit search predicate that restricts the search to find only bindings that have been imported or exported. Another mechanism is removing bindings from a contour. Scope opening is simply a change in contour search order – the opened scope becomes the first scope to be searched in the Search Model. Closed scopes are also a modification of contour search order.

2.7. Miscellaneous Features

Features concerned with naming that were not covered in the above major categories are discussed in this section.

2.7.1. Separate Compilation

In languages such as Ada and Mesa, package specifications, their bodies, and references to those packages and their contained declarations can exist in separately compiled source files. Resolving references while compiling one source file can require knowledge of the declarations in another source file. One of the motives for the provision of named scopes is to facilitate the use of module libraries that can be configured and maintained using separate compilation. A language must define which bindings in another compilation unit are visible.

2.7.2. Standard Environments

Many languages (for example Algol 68 and Ada) have "standard environments," which are composed of all declarations implicitly defined before the compilation of any program. The standard environment includes predefined types, procedures, and operators. Ada's standard environment contains, among other things, the functions and procedures for input and output. The scope of these declarations is often an implicit scope containing the outermost scope of the program. Alternatively the scope may be the outermost program scope itself.

2.7.3. More General Binding Mechanisms

Our discussion has concerned the use of names to introduce and to reference entities. As mentioned previously, multiple instances of an entity may be bound to a name during program execution in the presence of recursion. A more general way to reference entities is by means of indirection, through the use of pointers or reference types.

Some programming languages introduce yet more powerful mechanisms, through the use of pattern matching or associative search. Examples of such languages are Planner [Hewitt 1969], Conniver [McDermott and Sussman 1973], LEAP [Feldman and Rovner 1969], or the various dialects of PROLOG [Clocksin and Mellish 1981].

2.7.4. Persistent Storage

As a practical matter, programs must deal with persistent objects – objects with lifetimes greater than the programs that manipulate them. The most common examples of persistent objects are files, but any program object may be persistent.

If the concept of a persistent object is well integrated into a programming language, then name-entity binding for references to persistent objects can be handled just like any other reference. In the language L [Cormack 1983], the lifetime of an entity is defined separately from its scope, so entities with persistent lifetimes can be declared within an L program just like other entities.

Things are not always so clean, however. The most common case is that persistent objects such as files are not really handled within the programming language, and must be handled by giving a reference to the file system, which binds the reference and returns a descriptor to a file.

More recently, research on object-oriented databases and programming environments has motivated increased attention to these issues. Most of the underlying binding ideas have already been introduced, such as the encapsulation, inheritance, and closure mechanisms discussed in §2.6. The interested reader could look at PS-ALGOL [Atkinson and Morrison 1985] for more information.

2.8. Summary

We have presented a comprehensive survey of the language features currently existing which are concerned with declaration and naming. A tabular view of these features appears in Table 2.2. We have also presented the general principles underlying these features, in order to develop useful and simple models of scoping in programming languages. It is hoped that these ideas will help in the understanding of programming languages by making their differences and similarities more apparent.

Language	Dynamic or Static Inheritance	Dynamic or Static Binding Creation	Visibility Control Features					Scope Types[1]	Rebind Entities	Inner Bindings Shadow Outer Bindings
			Local Bindings	Open & Closed Scopes	Import Export	Scope Open	Multiple Inh.			
Ada[4]	S	S	•	•	•	•	NO	BFR[2]M	NO	•
Algol 60	S[5]	S	•	NO	NO	NO	NO	BF	NO	•
Algol 68	S	S	•	NO	NO	NO	NO	BF	NO	•
APL	D	S	•	NO	NO	NO	NO	GF[3]E	NO	•
Basic	N/A	Fixed	NO	NO	NO	NO	NO	G	NO	N/A
Bliss	S	S	•	NO	NO	NO	NO	GBFE	NO	NO
C	S	S	•	NO	NO	NO	NO	GBF[3]E	NO	•
CLU	S	S	•	NO	export	NO	NO	GBFR	NO	Illegal
COBOL	N/A	S	NO	NO	NO	NO	NO	GR	NO	N/A
Common LISP	S+D	D	•	NO	•	•	•	BFRM	NO	•
EL/1	S[6]	S	•	NO	NO	NO	NO	GBF	NO	•
Euclid	S	S	•	•[7]	•	NO	NO	GBFM	•	Illegal
FORTRAN	S	S	NO	NO	NO	NO	NO	GF[3]	•[8]	NO
L	S+D	S	•	NO	NO	NO	NO	BGFP	NO	•
LISP	S+D[9]	D	•	NO	NO	NO	NO	GBF[3]	NO	•
LM LISP	S+D[9]	D	•	NO	NO	NO	•	GBF[3]	NO	•
Mesa	S	S	•	•	•	•	NO	GBFM	NO	•
Modula	S	S	•	•	•	•	NO	GBFRM	NO	•
Modula-2	S	S	•	•[7]	•	•	NO	GBFRM	NO	•
Pascal	S	S	•	NO	NO	•	NO	GFR	NO	•
Plain	S	S	•	•	•	NO	NO	GFRM	NO	•
PL/I	S	S	•	NO	NO	NO	NO	GBFRE	NO	•
PROLOG	D	D[10]	•	NO	•[11]	NO	NO	GF[3]	•[11]	N/A
SETL73	S	S	•	•	import	•	NO	GBF	•	Illegal
SETL79	S	S	•	NO	•	•	NO	GF	NO	Illegal
Simula 67	S	S	•	NO	NO	•	NO	GBFM[12]	NO	•
Smalltalk	S	S	•	NO	NO	NO	•[13]	GBFM[12]	NO	•
SNOBOL	D	D	•	NO	NO	NO	NO	GF[3]R	NO	•
TEMPO	D[14]	D	•	NO	NO	NO	NO	GB	NO	•

[1] G=Global (distinguished level, such as **Pascal program ... end**, B=Blocks, F=Functions or Procedures, R=Records, M=Modules (or Packages), E=External Scope (visible outside of compilation unit), P=Permanent (permanent visibility - visible to other programs)

[2] Bindings in enclosing subprograms may be referenced using selected notation also.

[3] Restricted to one level.

[4] Overloading of designators allowed.

[5] A full implementation of Algol 60 requires dynamic inheritance for integer labels passed as integer arguments, but most implementations are not complete, and use strictly static inheritance.

[6] But has a feature to force dynamic inheritance when desired.

[7] Has pervasive identifiers also.

[8] Equivalence statement gives a limited form of renaming of run-time objects, but not of entities.

[9] Usually static in compiled LISP, except for "special" variables.

[10] Can add and retract clauses at run-time.

[11] Only in some versions.

[12] Actually classes, which are related to modules.

[13] In version described in [Borning and Ingalls 1983].

[14] An alternate version with static inheritance is described.

Table 2.2: Visibility Control Features of Common Programming Languages

CHAPTER 3

Models of Visibility Control

This chapter summarizes and discusses the Search Model of visibility control introduced in Chapter 2. It also expands on the Visible Set Model. The Range Model, a model of visibility control closely related to the Visible Set Model, is introduced.

Other models of visibility control which have appeared in the literature are also discussed. All of these models are presented as examples of ways of describing the visibility control rules of programming languages. Various basic concepts of these models have appeared in language definitions in the past, as will be discussed in Chapter 5. This is a prelude to formalizing the requirements of a model of visibility control, and presenting a better model of visibility control.

3.1. Definitions

We first present a few basic definitions which are needed in this and following chapters:

Definition 3.1. Binding Set

A *binding set* is a set of name-entity bindings. It is a set, as opposed to a multiset[21], because there must exist some equality test for potential elements of a binding set, and two "equal" bindings may not be members of the same binding set.

Definition 3.2. Visibility Construct

A *visibility construct* is a language construct that affects the visibility of names over some region of a program. Declarations and scopes are the most common examples of visibility constructs. An occurrence of a visibility construct in a program is an *instance* of the visibility construct.

Definition 3.3. Visibility Region

A *visibility region* is a region of a program that necessarily has the same set of visible bindings throughout, by virtue of the programming language's semantics. A visibility region might not be textually contiguous, as will be demonstrated later. Boundaries of a visibility region are normally demarcated by visibility constructs. Two program locations may have visible sets containing the same bindings without belonging to the same visibility region.

In a specification, if the program is represented by an abstract syntax tree, then a visibility region will correspond closely to a set of nodes in the abstract syntax tree representing the subject program.

3.2. The Search Model of Visibility Control

This section defines the *Search Model*, a model of visibility control, in which references are resolved by searching contours for a binding of the referenced name.

---

[21] Also known as a "bag."

A *binding* is a name→entity association. Its meaning is that in certain contexts, the name may be used to denote the associated entity. Bindings are created by implicit or explicit declarations.

A *scope* is a program unit used for defining a group of declarations and program statements. Each scope may be viewed as defining a *contour*, which contains the bindings associated with that scope.

The *range* of a binding is the region of source text in which the binding's name may be used to reference the entity bound to it by this binding. The range of a binding is defined by two sets of rules:

(1) The *binding placement rules* define for each declaration (implicit or explicit), to which contours the resulting bindings should be added. They also define the effect of import and export statements, scope opening, and related constructs.

(2) The *contour search rules* define which contours to search and in what order, when attempting to find a binding of a given name.

Each step in the search may restrict which bindings are valid results of the search. In Euclid, when continuing a search outside of a module (which is closed to all bindings except pervasive bindings), a restriction to the search is added which specifies that only pervasive bindings are valid results of the search. A *search predicate* is associated with each search. The search predicate initially consists of the restriction that any binding found match the referenced name. The search predicate may be augmented with additional restrictions at each step of the search by conjunction of the new restrictions with the old search predicate.

There are two types of references: *simple references* and *qualified references*:

(1) A simple reference consists simply of a name. The name is bound to a binding whose range includes the current source position. The contour search rules (and possibly overloading rules) are used to find the appropriate binding.

(2) A qualified reference consists of two parts: a left-part whose associated entity *e* is found using the contour search rules of the language, and a right-part which is a name. *e* must be a scope, and the desired binding is found by searching for the right-part name in the set of bindings associated with *e*'s contour[22]. Note that this is a recursive definition, since the left-part may itself be a qualified reference.

This definition of the Search Model is not complete: for example, it does not describe how ordering of declarations and references in a language that requires declaration-before-use is handled. These issues will be discussed in later chapters. Until then, this description of the Search Model should suffice – the important point is the style in which things are described using the Search Model.

The implementation of the visibility control rules for Ada described by Blower [Blower 1984] is an excellent example of the use of the Search Model.

3.3. The Visible Set Model of Visibility Control

The philosophy of the Visible Set Model could be stated as, "Have information easily accessible when and where needed, as opposed to computing it at the point of demand." The Visible Set Model defines the visibility control problem as determining what set of bindings are visible at any point in the program where a reference may occur. We call this binding set the *visible set* at that point. If the visible set for a point is well defined, then it is easy to find the

[22]The rules for resolving qualified references in COBOL and PL/I are more complicated, and do not fit into this model for qualified references. However, qualified references in all more modern languages do.

appropriate binding of a name at that point. In the absence of overloading, there will only be a single binding of a given name in that visible set, and the set defines a function from names to entities. In the presence of overloading, the overloading algorithm must be used to select among multiple bindings of the same name visible at a point.

Simple and qualified references:

(1) A simple reference consists simply of a name. The reference is resolved by looking in the visible set associated with the point of reference for a binding of the referenced name. The visible set represents a function from names to entities in a language without overloading. In the presence of overloading, the visible set represents a one-to-many mapping from names to entities. If a name is overloaded, then the language's overloading algorithm must be used to select among the entities bound to that name at a given point.

(2) A qualified reference consists of two parts: a left-part whose associated entity e is found using the visibility control rules of the language, and a right-part which is a name. e must be a scope, and the desired binding is found by searching for the right-part name in the set of *local* bindings associated with e. This is a recursive definition, just as in the Search Model. However, in the Visible Set Model, when the left-part is just a name, its binding is found using the Visible Set Model method for resolving simple references.

Defining lookup in the Visible Set Model is very straightforward. The more difficult part in defining the visibility control rules of a language is defining the visible set associated with any contour. In a block-structured language, the visible set for a contour c is defined in terms of its set of local bindings and the visible set for the immediately enclosing contour c'. c's visible set is obtained by computing the union of c's local bindings and c''s visible set, omitting from the result any bindings from c''s visible set that are hidden by c's local bindings. If c is partially closed, then the bindings visible in c' are inherited by c only if they satisfy some predicate associated with c. Other visibility control disciplines can be defined by similar operations.

The bootstrap Ada compiler described by Blower [Blower 1984] uses the Visible Set Model to implement visibility rules.

3.4. The Range Model

The Range Model defines the visibility control problem as determining where each binding is visible, as opposed to the Visible Set Model, where the set of bindings visible at each point must be defined. The *visibility range* of a binding is the region of the program in which the binding's name may be used to reference the entity bound to it by this binding. The range may be defined in terms of contours or some other suitable unit.

One method of describing the visibility range of a binding is by propagation. We propagate the visibility of a binding in a step-wise manner from the point of its introduction to all contours where it is visible.

We define the propagation in terms of contours and dependencies between contours. Each binding is already visible in the contour corresponding to the contour where it is declared. The following procedure *Make_Visible* assumes visibility of the binding in that contour, and does the propagation. A dependence of a contour c_{inh} on contour c is denoted by the pair (c, c_{inh}). *clash* returns **true** if two bindings clash, as described in Chapter 2. *inherits* returns true if the dependence between the contours allows the binding to be visible in the dependent contour.

```
procedure Make_Visible (
    b_new : Binding,
    c : Contour);
    ∀ contours c_inh s.t. ∃ dependency D=(c ,c_inh)
        if ∃b :Binding in c_inh s.t. clash(b ,b_new) or inherits(b_new ,D) = false then
            return /* stop propagating visibility along dependencies */
        else
            add c to b_new 's visibility range
            Make_Visible (b_new ,c_inh)
```

For a given binding b_{new} and a contour c, *Make_Visible* propagates the visibility along dependencies from c. Visibility of b_{new} is only propagated if the dependence causes b_{new} to be inherited and b_{new} isn't shadowed by some binding in the dependent contour. *Make_Visible* must be iterated over all bindings to define the entire visibility of a program.

The procedure *Make_Visible* is fairly simplistic, and not complete. However, this *Make_Visible* does handle general forms of restricted inheritance, including (partial) closed scopes and inheritance-style import/export.

The effect of *Make_Visible* is to define a mapping from bindings to contours. Lookup in the Range Model is straightforward. It involves finding bindings matching the referenced name that have the current contour in their visibility range. *Make_Visible* will vary greatly from language to language, while the definition of lookup should be standard for all languages.

The definitions of *Make_Visible* and *Lookup* (defined more precisely in §5.3) together define the visibility control rules of the language being defined. *Make_Visible* will vary greatly from language to language, while *Lookup* should be standard for all languages.

Another way to describe the visibility range of a binding is to treat each dependence between contours along with the *inherits* function as a constraint to be satisfied.

We explain the technique by means of a simple example: Let the visibility construct *VC* be a simple declaration of a variable *foo* in a language with declaration-before-use. *VC* is related to two contours, c_p immediately preceding it textually, and c_f immediately following it. The dependence attached to *VC* and *inherits* together form the constraint that if a binding is visible in c_p and does not clash with the new binding of *foo*, then it is visible in c_f.

Each such constraint is called a *propagation constraint*. The propagation constraints of all the visibility constructs of a subject program form a set of constraints that must be satisfied, where each variable is the visibility range of a binding. The basis constraints are that each binding is visible at the point where it is declared (or whatever the language specifies). The constraints formed by the propagation constraints are satisfied only when each binding is visible everywhere it should be.

To define the visibility control rules of a language using this model, we define a propagation constraint for each type of visibility construct (and thus for each type of dependence between contours), which defines the propagation of bindings at an instance of the visibility construct based on the specific attributes of that instance.

3.5. Reiss's Model of Visibility Control

This is a summary of Reiss's models of visibility control [Reiss 1983]. Reiss defined two separate, but related, models of visibility control. The first was manifested in a specification language intended for use in generation of symbol table modules, while the second was a formal model of visibility control. His emphasis was on the former, and we will follow that emphasis. His formal model will be discussed in §3.5.5. Any reference to his model of visibility control means the specification language unless otherwise specified. His model is important to us for two

reasons:

(1) It is one of only two other attempts to define a formal model of visibility control in programming languages that we know of.

(2) The model contains features different from any of the models we have independently developed, so its relationship to our models is interesting.

This summary is not comprehensive; we concentrate particularly on features of the model that are relevant to this research.

Reiss provides for multiple classes of entities and names. Names represent the lexical tokens associated with entities. Different entity classes have different information associated with them, and may require different scoping rules.

He describes scope in terms of three attributes of entities:

extent scope
: The smallest (textual) scope containing all portions of the source program in which the name can refer to the entity.

direct visibility scope
: A scope in which a name refers to a particular entity regardless of other definitions of the name in other scopes. There may be more than one direct visibility scope per entity. The most common example of this is the scope in which an entity is declared.

lifetime scope
: The outermost scope of the source program for which the entity will exist at run time.

Lifetime scope is not a major concern of the research described in this dissertation; however it is discussed some in later chapters.

The three primary domains for symbol processing are: **NAME**, **OBJECT** (entities), and **SCOPE**. Secondary domains exist for representing attributes of elements of the primary domains, and other information.

3.5.1. Standard Functions

The model defines standard functions for performing symbol processing operations.

The functions for creating elements of the primary domains and setting attribute values are NEWOBJECT and SETVALUE. DEFINE creates a binding of a name and an object.

The access functions are:

FIND
: takes a source token and returns the appropriate name.

LOOK_UP
: maps from a name to the set of all objects that can be associated with that name (anywhere in the program).

RESTRICT_VISIBILITY
: restricts a set of objects according to rules defining the relationship between where an object is visible and where it is declared. The form of these rules will be discussed later.

RESTRICT_CLASS
: restricts a set of objects according to an expected class.

RESTRICT_SIGNATURE
: restricts a set of objects according to the signatures of the objects. The signature and the equality test for signatures are user-provided.

RESTRICT_USER
restricts a set of objects according to an arbitrary user-supplied Boolean function.

RESOLVE
checks that a final object set contains a single element, and returns that element or an error value.

VALUE_OF
Used to obtain values of attributes.

3.5.2. Definitions

Declarative rules define what the result of a declaration is – *i.e.*, whether the declaration is legal and where the binding is visible.

Object class, scope class, and scope attributes are used to select one of the following four choices for the proper extent, visibility, and lifetime scopes for the resulting binding:

(1) the current scope is chosen.

(2) the parent of the current scope is chosen.

(3) the parent of the current scope is tested recursively with the current object.

(4) the association is illegal (error).

These rules also define what happens when a new declaration has the same name as a previously defined binding, based on the classes of the old and new objects, and on whether the old object is directly visible in the current scope. The four possible actions are redefinition, reference, overloading, and error.

3.5.3. Extensions

Reiss provides extensions to the model for more complete handling of symbol processing.

One of the most important of these is the IMPORT function, which allows a scope to be added to the set of visibility scopes for a given object. A DEPORT function is also provided for removing a scope from the visibility set for an object.

The RESTRICT function can be used to assign a new scope class to a scope, in order to assign new properties to the scope. This is useful, for example, in Ada where the effect of declarations in a package depend on whether they occur in the public or private parts of the package specification, or in the package body.

Other, mostly implementation-oriented extensions are also available.

3.5.4. Summary of Reiss's Model

The general approach to name resolution is as follows: declarations create associations between names and objects (what we call bindings). Declarative rules define where bindings are directly visible, how the visibility of bindings is propagated through nested scopes[23], and what happens in the event of duplicate declarations. FIND and LOOK_UP are used to find all bindings of an identifier, after which RESTRICT functions are used to successively restrict the set of bindings with various conditions.

This model is the basis of a system for generating a symbol processing module from a declarative specification. The interface between the the symbol processing module and the rest of

[23] Whether a particular scope is open or closed to a particular object is determined by the OPEN_SCOPE mapping, which contains all pairs <*s*:SCOPE_CLASS, *o*:OBJECT_CLASS> such that scopes of class *s* are open to objects of class *o*.

the language processing system is via the function and procedure calls discussed above (plus others)[24]. It is up to the remainder of the compiler to determine when and where symbol processing functions are applied. Order is implicit in the scoping model. However, there is an operation to set the current scope, so there is at least some provision for moving around in a program independent of textual order.

In the case of scope class restriction, this temporal ordering of scoping operations is particularly important, because the order of declarations and class restriction is critical to which declarations are exported in Ada packages. Problems with Reiss's model will be discussed in §9.2.1, after the issues involved in designing a model of visibility control are discussed in detail.

3.5.5. Reiss's Formal Model

Reiss's formal model of visibility control is composed of several basic relations, along with functions that access and/or modify the relations. The basic relations include **DIRECTLY_VISIBLE, NESTED_IN,** and **EXTENT** (the extent scope of an object), among others. These relations represent the visibility structure of a program. The functions include **LOOKUP**, which accesses the basic relations, **DEFINE**, which modifies the relations, and others. The basic definitions used in this model, such as the meaning of "directly visible," are the same as those used in the specification language.

Ordering of declarations and uses is still implicit in the formal model, so the meaning of redefinitions is still dependent on this implicit ordering.

3.6. Wolf's Model of Visibility Control

As discussed in §2.6.7, Wolf, Clarke, and Wileden [Wolf 1985; Wolf *et al.* 1987] introduced a model for visibility control based on the provision and requisition of access, and proposed language constructs (*provide* and *request* operations) that make provision and requisition of access as precise as desired. All access (visibility) is described in terms of separate provision and requisition of access. In general, access is provided by any type of entity, while access is requested by scope entities. An entity E is normally visible in scope S only if E provides access to itself to S, and S requests access to E.

Provision and requisition of access within a program are described in terms of a directed *visibility graph* $g = (N, A_r, A_p)$ which can be split into a requisition graph $g_r = (N, A_r, \varnothing)$ and a provision graph $g_p = (N, \varnothing, A_p)$, where the A's are the edge sets. A provision edge (n_i, n_j) means that n_i provides access to itself to n_j. Requisition edges go from the requestor to the requestee. Correctness of a program normally requires that $A_r^R \subseteq A_p$, where $A_r^R = \{ (n_i, n_j) \mid (n_j, n_i) \in A_r \}$.

A *visibility function* v is a function that maps a program representation x to a visibility graph g. For each visibility mechanism m[25] there is a visibility function v_m, where $v_m(x) = r_m(x) \cup p_m(x)$. r_m and p_m respectively produce the requisition and provision graphs for x. r and p are in turn composed of functions operating on different kinds of entities E_i:

$$r_m(x) = r_{m,E_1}(x) \cup \cdots \cup r_{m,E_n}(x)$$

$$p_m(x) = p_{m,E_1}(x) \cup \cdots \cup p_{m,E_n}(x)$$

---

[24] Actually, two different levels of interface are provided. We have discussed the higher-level interface, as the lower-level interface is less interesting for our purposes of comparison.

[25] This seems to correspond closely to a visibility construct, as defined in the introduction to this chapter.

Wolf also defines a *nesting graph*, which is used to represent the nesting structure of an ALGOL 60 program. Nodes in the graph corresponds to scopes, with an edge from a scope to each immediately contained scope. For ALGOL 60, the graph is a tree. In the example Wolf gives, only subprograms are considered, and the visibility function defines the visibility graph edges in terms of "parent", "sibling", and "ancestors" relations on the nesting graph.

The only example given by Wolf is a very simple one, and it is unclear how language features such as declaration-before-use and shadowing would be handled, both in the nesting graph and in the visibility functions which must translate the nesting graph into the corresponding visibility graph.

CHAPTER 4

Requirements for a Model of Visibility Control

A primary goal of this research is the definition of the visibility control rules of programming languages. To do this, one must define the meaning of any name reference in a program – that is, to which binding the name refers.

The first step in developing a method of defining visibility control rules is to develop a model of visibility control – a way of talking about visibility control rules and the constructs that affect the visibility of names in programs. Once we have an acceptable model of visibility control, we can base a specification method on the model.

Chapter 2 described most of the visibility control features present in programming languages today, and introduced two basic models of visibility control (the Search Model and the Visible Set Model) that can be used to describe these features. Chapter 3 summarized these two models and presented other models for describing visibility control rules. This chapter describes in more depth the requirements a satisfactory model of visibility control must fulfill, beginning with a few general requirements, and continuing with specific requirements due to the presence of the visibility control features discussed in Chapter 2 and the influence of these requirements on our designs of models of visibility control. Each specific requirement is typeset in italics, while discussion and elaboration of each requirement is typeset in the normal roman font. A table summarizing these requirements is presented at the end of the chapter.

As a result of our investigations into the requirements of visibility control features, I have formed some opinions about the desirability of some of these visibility control features, and discuss these opinions in this chapter.

4.1. General Requirements

This section discusses the general requirements of a model of visibility control. Such *a model of visibility control should be natural*, corresponding closely to the way a person thinks about the visibility control rules of a language. If the model is natural then language descriptions are more likely to be easy to understand.

The model should be complete. It should be general enough to describe all (or at least most) visibility control features used in programming languages with reasonable ease. Ideally, the model should be general enough so that it is likely to capture new notions of visibility control that may arise later. It is difficult to judge whether a model has this generality, since it is impossible to predict what visibility control concepts will be developed. However, a model that consists of a few general concepts that cover a wide range of visibility control features is more likely to meet the test of time than a more complex model that contains specific features to model specific visibility control features.

Difficulty in describing arcane features that are not included in newer languages should not be considered a significant drawback. Indeed, a correspondence between difficulty in describing a visibility control feature and difficulty in understanding or implementing that feature should be considered a plus, because the model could help a language designer to identify potentially troublesome visibility control features.

The model should be useful for reasoning about the visibility control rules of a language. This is useful to the language designer also.

Finally, *the model should be suitable as the basis for a formal specification method from which an implementation can be generated.* It should be possible to generate an acceptably efficient implementation so that the formal specification method can be used to develop production language processing systems. The required generation process should not be excessively complex, or implementors will be unlikely to implement it.

4.2. Visibility Features and Their Requirements

This section discusses the visibility control features described in Chapter 2 along with the resulting requirements on a model of visibility control. The structure of this section roughly parallels the structure of Chapter 2, in the order in which the language features are presented.

4.2.1. Names, Entities, and Bindings

A model of visibility control should support the three basic objects: names, entities, and bindings, as defined in Chapter 2.

There is a fair amount of latitude in precisely how the model defines an entity, but *there must be a mechanism for defining and retrieving attributes of an entity.*

The concept of a binding in the model must support the fact that the relation between names and entities in a programming language may be a general n:m mapping, since there may be more than one name per entity, and more than one entity per name.

4.2.2. Scopes and Contours

The model of visibility control should support a general concept of a scope, including the notions of blocks, subprograms, records, modules, etc., as defined in §2.3. As also defined in §2.3, a contour defines the local environment introduced by a scope. For the moment, I will continue to use the definition of a contour in which a single contour corresponds to a single scope. This definition will be extended and made more precise in §4.4.

The grouping provided by scopes also forms the basis for qualified references, such as references to fields of records, subprograms (Ada [USDoD 1983]), or modules. *For a qualified reference $s.f$, one must be able to perform a lookup restricted to the fields of scope s*, in order to find the binding of f local to s. That is, the lookup is restricted to the bindings directly visible in s. One way to satisfy this requirement is to make it possible to reference the set of bindings local to any contour. Another, more general method is described in the next section.

4.2.3. Declarations

A declaration causes the creation of entities, and possibly the binding of one or more names to one or more entities. These bindings may be local to the contour where the declaration occurred, or they may belong to some other contour. A binding "belongs to a contour" if the binding's apparent point of declaration is within that contour. The most common form of a binding belonging to a non-local contour is when the binding belongs to an enclosing scope, such as with PL/I implicit declarations. In general, *it must be possible to declare a binding in an arbitrary contour in a program.*

The model of visibility control must provide a general mechanism for detecting multiple declarations of a name, and for performing appropriate actions when multiple declarations are detected.

Two bindings *clash* according to the naming rules of a language iff both bindings cannot be directly visible at the same point. In most languages, two bindings clash iff their names are identical. However, deciding whether two bindings clash can be more complex, especially in the presence of overloading. *The model of visibility control must provide a mechanism for defining whether two bindings clash.*

As discussed in §2.4.2 there are four possible conditions when two bindings clash: shadowing, overloading, reference, and error. The model must allow the describer (the person describing the visibility rules of a language) to select one of these conditions and define the meaning of clashing bindings for each condition. Part of this may be subsumed into determining whether two bindings clash: for example, one may be able to define the meaning of "clash" such that two bindings that overload one another do not clash. Then, if two bindings clash, only three conditions are possible: shadowing, reference, and error. Henceforth, the discussion will assume that overloaded bindings do not clash with one another.

Since a declaration may cause a binding to be added to an arbitrary contour in the program, the mechanism for detecting multiple declarations must be capable of operating on an arbitrary contour. There are a number of commonly used methods for detecting multiple declarations (conventional block structure with shadowing is presumed here):

(1) Look, using the normal visibility rules, for a visible (not necessarily directly visible) binding that clashes with the new binding. If the existing binding was declared in the current contour, then the new binding either references the old one or is erroneous. One must be able to determine a binding's declaration contour in order to use this method.

(2) Look for an existing binding b_1 declared in the contour to which the new binding is to be added, where b_1 clashes with the new binding b_2. In other words, look for a *directly* visible binding that clashes with b_2. If such a binding b_1 exists, then the new binding b_2 either references b_1 or is erroneous. One must be able to determine the set of bindings declared in any given contour in order to use this method.

A more general way to look at the multiple declaration problem is to define a separate set of visibility rules that define which bindings are visible when determining, given a particular binding b_1 occurring in a particular place in a program, whether another binding b_2 exists such that b_1 and b_2 clash.

If a particular binding b_3 can be proven not to clash with b_1 by virtue of its location in the program (its contour) or some other property not dependent on the attributes of b_3 itself, then there is no reason for b_3 to be visible when checking for multiple declaration. The simplest example of this occurs in a block-structured language with shadowing: a binding b_1 will shadow any binding b_2 in an outer scope that clashes with it, so no outer binding can cause the reference or error actions. Thus, it is only necessary to examine the local bindings to check for multiple declaration. This is what method (2) above is doing, reflecting that only bindings in the current scope need be considered when looking for illegal multiple declarations. Method (1) achieves the same result in a round-about way, by first doing a normal lookup and then restricting the results of the lookup to the current scope, if appropriate. However, both methods (1) and (2) are just restrictive operational descriptions of methods for finding illegal multiple declarations, each assuming that the visibility rules require that only the current scope need be examined when checking for multiple declarations. The ability to define a different set of visibility rules for this purpose permits us to eliminate this assumption.

The model of visibility control must provide the ability to define more than one class of visibility, for use in different contexts, (such as normal reference as opposed to checking for clashing bindings).

This ability also makes it possible to restrict a lookup to the bindings declared local to a given scope, as is required for a qualified reference (§4.2.2).

We must be able to describe the appropriate action (shadowing, reference, or error) once

multiple declarations (clashing bindings) have been detected[26]. The error action merely requires the ability to report a message to the user and a method of gracefully preventing a cascade of error messages due to a single error. Shadowing has more to do with resolving references, and will be discussed in §4.2.5. The reference action is more complicated, and is sometimes an example of a more general feature called a variable attribute, which is discussed in §4.2.4.

Order of Declarations and References

In order to handle declaration-before-use, *the model of visibility control must be able to represent the order of visibility constructs and the order of references with respect to visibility constructs.*

The model of visibility control must also be able to handle both forms of shadowing, as described in §2.4.3. In the first form of shadowing, an outer binding of a name "N" is visible up to the point where a new declaration of "N" occurs, as in Ada (the outer binding may either be shadowed at the beginning of the new declaration, or just prior to the point where the new binding becomes visible). In the second form (as in Pascal), an outer binding of "N" is visible only up to the beginning of a scope that contains a local declaration of "N". From the beginning of the scope to the local declaration of "N", neither the local binding of "N" nor any outer binding of "N" may be referenced. Following the terminology used for Euclid [Lampson *et al.* 1981], one can say that the local binding of "N" is *known* but not visible in this region of text.

The generalization of known but not visible bindings is just a form of allowing more than one kind of visibility, as introduced in the previous section.

Dynamic and Static Name and Binding Creation

The model of visibility control should be able to handle both dynamic and static creation of names and bindings. Indeed, this should not affect the model of visibility control significantly, as these operations should be similar in the static and dynamic cases.

4.2.4. Variable Attributes

A *variable attribute* is an attribute of an entity or binding whose value varies over the range of visibility of a binding. This section discusses language constructs that cause modifications to attribute values, and how to describe them.

§4.2.3 stated that the model of visibility control must allow the expression of the reference action when bindings clash. The essential language concept that results in the reference action is the use of two separate visibility constructs (usually declarations) to define a single entity. In Pascal, a "forward" declaration defines the interface of a procedure, while the completion of the declaration of the procedure defines the execution semantics of the procedure (the code attribute). In Modula-2 [Wirth 1982], declaration of an *opaque* type consists of a declaration of the type name only in the definition module, with a complete declaration of the type in the corresponding implementation module.

In the Pascal example, the entity corresponding to the procedure can be created when the forward declaration is processed, and modified to contain the code attribute when the body of the procedure decaration is processed. This approach works in this case because the code attribute isn't needed in the binding resolution or type checking processes.

The opaque export in Modula-2 does not satisfy this condition. When a name-only type declaration occurs in a definition module, resulting in an opaque export of that type, the details of the type are not available in the definition module or in its clients. So the entity and the binding

[26] Remember that, according to the definition of "clash", overloaded bindings do not clash.

of that entity visible in the definition module and its clients differ from the entity and binding visible in the implementation module.

A related example is the *read-only* export in Modula [Wirth 1977]. A variable exported with the **readonly** attribute may be referenced but not assigned to in the scope to which it is exported.

These three different constructs are discussed together because their syntactic representations in programs are similar – in two separate parts, with one part modifying or qualifying the meaning of the other part. For the read-only export, the second part is the export statement. But, there are two different concepts represented by these constructs. The opaque and read-only exports create different views of an entity (from the programmer's viewpoint), while the Pascal forward declaration is simply a two-part declaration of an entity, with a uniform meaning throughout its range. Indeed, it is simply a notational convention introduced to make mutually recursive procedures possible when declaration-before-use is required.

Thus, the Pascal forward declaration and similar constructs can be handled with the ability to change the value of an attribute of an already existing entity. The value of this attribute is uniform throughout the range of the binding of the entity. So, the only additional requirement is that the value of the attribute not be used until its final value is assigned. In this case, it simply means that the code attribute can't be used in the binding resolution process. There is no reason to do so, so there is no problem. The Pascal forward declaration does not result in a variable attribute, as defined at the beginning of this section, because the value of the code attribute does not vary over the range of the procedure binding.

The opaque and read-only exports differ from the Pascal forward declaration because they result in an entity that has an attribute whose value varies over the range of the entity's binding. One approach to this is to add contextual tests to determine the attribute's value at a given program point. Another approach is to consider the variants of the entity as separate (but dependent) entities bound to the same name, with the resulting bindings having disjoint ranges. This is the approach used in this dissertation.

Another instance of a variable attribute occurs in languages that require declaration-before-use, and allow qualified references nested within the object whose fields are being referenced. This can occur in Ada, because a binding in an enclosing subprogram S can be referenced, even if the binding is shadowed at the point of reference, by qualifying the reference with the name of the subprogram.

A variable attribute must be used because there may be many declarations local to S, and thus many visibility regions. The attribute *ref_env* of the entity describing S defines the referencing environment for a qualified reference to a field of S, but the proper referencing environment depends on where within S the reference occurs, just as for a simple reference local to S. This referencing environment is the visibility region immediately preceding the scope nested in S that (directly or indirectly) contains the qualified reference. Thus a new binding of "S" must be produced with each new declaration in S, with the *ref_env* attribute modified to reflect each new binding. The appropriate binding of "S" must be visible at each place where a qualified reference may occur.

In summary, *the model of visibility control must support variable attributes.*

4.2.5. Resolving References

Different languages have widely varying rules for resolving references, so any model of visibility control must have a general and powerful method of representing these rules. In the Search Model introduced in §2.5, this requirement is met in part by allowing arbitrary search order of contours. In the Visible Set Model also introduced in that section, the same goal is achieved by providing powerful primitives for the definition of the visible set for each program

region. *A model of visibility control must be able to represent arbitrary order of inheritance of visibility of bindings from one scope to another*, whether this inheritance order is due to nesting, named inheritance, or other visibility constructs. As discussed in §4.2.3, a model of visibility control must support more than one class of visibility, so, for the generality and orthogonality, *a model of visibility control must support a different inheritance ordering for each class of visibility*.

Suppose we are using the Search Model, with the means for specifying any search order of contours. The remainder of this section demonstrates that a wide variety of visibility control features can be handled with only the capabilities already described in this chapter.

Lexical and dynamic inheritance introduce no special requirements for this model of visibility control. They both require arbitrary search order of contours. The only difference is that the search is based on dynamic structure with dynamic inheritance, instead of lexical structure. The model of visibility control can remain the same in both cases – the application is just slightly different.

Closures also introduce no special problems for a model of visibility control. Arbitrary order of inheritance (handled by arbitrary search order in the Search Model) is what is needed to handle closures. The difficulties lie more in the implementation – the contours that may be required by the invocation of a closure must be kept until they are no longer needed.

Named inheritance and multiple inheritance can also be handled by arbitrary order of inheritance. The programmer defines this order explicitly by naming the scopes instead of defining the order by the lexical structure of the program.

4.2.6. Explicit Visibility Control

Open and Closed Scopes

Open scopes are the most common case. Closed scopes restrict the visibility of bindings across scope boundaries, and the model of visibility control must provide some mechanism for defining this restriction. A closed scope generally creates a barrier to the visibility of all bindings global to it. Pervasive bindings are an exception to this: they are visible across the barrier created by closed scopes. The first step in generalizing pervasive bindings is to allow different classifications of scopes $SC_1, \cdots, SC_n$, and different classifications of bindings $BC_1, \cdots, BC_m$, along with a specification of a *door function*

$$door : binding\ class \times scope\ class \rightarrow \{\ open, closed\ \}$$

that, for any binding class BC_i and scope class SC_j, defines whether scopes of class SC_j are open or closed to bindings of class BC_i. The classification of a binding may be determined by the classification of the bound entity.

Imports and Exports

Import and export statements cause explicit changes to the normal visibility of one or more bindings. However, one must be very careful in the definition of the meaning of import and export statements[27], because there are many subtleties. For example, when a binding b is imported from scope S_1 into scope S_2, there are two significantly different meanings possible:

(1) The result is equivalent to creating a new declaration within S_2, where the new declaration is similar or identical to the declaration that resulted in binding b.

[27] The most common form of import (export) is intended here: a single binding is imported (exported) from one scope to another.

(2) The result is equivalent to making scope S inherit visibility of b, thereby making b visible within S, just as in the usual inheritance due to nesting of open scopes.

Language definitions often do not explicitly define which of these two meanings an import (export) has. Rather, the meaning of an import (export) of b is implicitly defined by what operations are allowed on b once it has been imported (exported), and by what action results from an imported (exported) binding clashing with another binding.

Which of these forms of import (export) is chosen determines how import (export) must be handled in the model of visibility control. (1) can be handled with the capability of adding a binding to any contour of a program, or with appropriate use of different visibility classes. (2) is a form of inheritance, but the capabilities thus far described are insufficient. Indeed, when we import a binding from scope S_1 to S_2 (with the inheritance form of import), we are creating a path of inheritance that is closed to all bindings but the specific binding being imported. This is a further generalization of the door function described above: The decision whether S_2 is open to bindings from S_1 is based not only on the S_2's scope classification and the classifications of the bindings visible in S_1, but also on whether each binding is named in an import statement in S_2. Other factors can affect the decision: the set of bindings available for import may include all bindings *visible* in S_1, or it may include only those bindings *declared* in S_1. The final generalization of inheritance, which is capable of handling all forms of inheritance discussed, is now presented.

Definition 4.1. Inheritance Link

Inheritance between two scopes is defined by an *inheritance link* (S_1,S_2) from the *providing scope* S_1, from which bindings are inherited, to the *inheriting scope* S_2, which inherits bindings. Whether a binding visible in S_1 is visible in S_2 is defined by a revised door function, as first introduced at the beginning of this section. This is called the *inheritance restriction function*:

$$inherits : binding \times link \rightarrow \{ \textbf{true},\textbf{false} \}$$

inherits determines whether a link is open or closed to a binding based on properties of the binding and properties of the link. Iff *inherits* returns **true**, the binding is inherited via the link if it is visible in the providing scope. Possible properties of the binding used by the inheritance restriction function are whether it is pervasive or not, whether it is a type or a variable binding, and so on. Possible properties of the inheritance link used by the inheritance restriction function include the visibility construct that caused the link (an open or closed scope, an import or export, etc.). In addition, the inheritance restriction function may specify that certain instances of links are open only to particular bindings, as in an import statement that only causes inheritance of specific, named bindings.

Given this definition of an inheritance link, the visbility control structure of a program can be represented by a graph, with nodes representing visibility regions, and inheritance links for edges. This idea will be discussed in more depth in Chapter 6.

A model of visibility control should support a general form of inheritance, allowing nonlocal inheritance, and providing precise control over which bindings are inherited from a providing scope to an inheriting scope.

Import/Export Statements that Create Declarations

A form of import (export) statement was discussed earlier that was similar to creating a declaration in the scope inheriting the new bindings. The actual requirement is that the exported binding must have the same apparent affect in the inheriting scope as a binding declared in the inheriting scope. This requirement is satisfied by the use of different visibility classes and control

over inheritance. In any visibility region *vr* in a program where a binding *b* behaves as if it were declared in the scope to which *vr* is local, *b* must be visible with the visibility class assigned to local declarations.

Opening of Scopes

The first form of scope opening (§2.6.8) is a bulk import, and can be handled with the same mechanisms as a simple import. The Ada **use** must, in addition, decide which bindings are actually to be imported.

The other form of scope opening is exemplified by the **with** statement in Pascal. This form of scope opening is a special case of the first form, and so can be handled by the same mechanism. A new contour is created, which inherits visibility of all bindings corresponding to fields of the record being opened.

Multiple Inheritance

The requirements thus far described are sufficient for describing multiple inheritance, except for one issue: the discrimination between inherited bindings based on ordering. This issue arises in LISP Machine Flavors [Weinreb and Moon 1981], where each scope (flavor) can specify a list of scopes (other flavors) from which bindings are to be inherited. Each of these scopes in turn can specify a list of scopes to inherit bindings and so on. The rules of the language specify that the appropriate binding referenced by a name is the one found by performing a top-down, depth-first walk of the tree of scopes formed by the inheritance structure. For example, if a scope S_1 imports the bindings from two other scopes S_2 and S_3, in that order, and both S_2 and S_3 declare bindings of "N", then a reference to "N" in S_1 should be bound to the binding declared in S_2.

Describing this feature requires *the ability to select among the bindings inherited by a contour based upon some ordering.*

4.2.7. Miscellaneous Language Features

Separate Compilation

The visibility control information for one source file may be needed in order to process another source file. This doesn't really affect the model of visibility control; however, the implementation must provide a mechanism for saving and retrieving visibility control information.

Standard Environments

The only requirement caused by standard environments is that *it must be possible to represent visibility control information independent of the processing of any specific source code.* This is because the entities defined in the standard environment are often not defined in the source language for which the standard environment is defined. Rather, they are often defined or implemented in another language, and the standard environment provides bindings to these "foreign" entities.

4.3. Declaration-Before-Use and Grouping of Bindings

As discussed in §2.4.3, most recent languages that require explicit declarations also require that a declaration of a name precede any reference to that name, with certain limited exceptions. The primary original motivation for this requirement was to make one-pass semantic analysis possible by forcing all declarations to precede any references, as in the *User Manual* for Pascal [Jensen and Wirth 1974] and in ANSI/ISO Standard Pascal. The impact of declaration-before-use on a model of visibility control is the requirement that the model be able to represent the

order of declarations within a scope, in addition to the order caused by the nesting of scopes.

In the Search Model described in Chapter 2, all bindings resulting from declarations local to a single scope are grouped together in the single contour associated with that scope, even when the language requires declaration-before-use. In this model of visibility control, resolving a reference consists of searching contours sequentially until a contour containing a binding of the referenced name is found. The granularity of the search is the contour, and the group of bindings associated with each contour. Little was said about how the order of declarations is represented, or what the implications of declaration-before-use on a model of visibility control are. Obviously, the order must be represented somehow, and a name reference within a scope is only legal if the declaration precedes the reference.

Another approach is to keep all bindings separate, as opposed to grouping them together by contour. The only exception is the case where more than one binding becomes visible at the same point. One example where this happens is in languages where uses may precede declarations, such as in PL/I. All bindings declared in the the same scope become visible at the beginning of the scope, at precisely the same point.

Each method has advantages and disadvantages. This section compares the two methods, with the goal of selecting the better method and understanding how to overcome the disadvantages of the chosen method. Table 4.1 contains a comparison of the two methods, listing their advantages (+) and disadvantages (-).

The table comparing the two different approaches (grouped versus ungrouped bindings) illustrates two major tasks that a model of visibility control must handle well. The first task is the representation of ordering present in programs, on two levels: the sequence of declarations, one after another, and the nesting of scopes, one inside another. The need for this ordering is obvious. The other task is the representation of grouping of bindings. This grouping is necessary for lookups limited to a single scope, and for retrieving the bindings of a single scope.

The two opposing approaches represent different priorities on the two tasks. Grouping the bindings into contours places a higher priority on the grouping task. However, this forces a two-level ordering: the ordering of contours, and the ordering of bindings within contours.

Keeping the bindings ungrouped places a higher priority on the ordering task: all ordering can be handled by a single, simple mechanism – the ordering of declarations with declaration-before-use can be handled by treating declarations similarly to new scopes. However, this requires a separate mechanism for describing the set of bindings associated with a particular scope.

The single biggest problem with either approach is with grouped bindings. The Visible Set Model and Range Model are very difficult to apply with grouped bindings. The Search Model works well with either grouped or ungrouped bindings. It seems wise not to eliminate any useful models at this point, so it is best not to group bindings. Also, the complexity required by the separate mechanism for describing a contour's bindings is less than that of the two-leveled ordering required with grouping. Our task then is how to overcome the drawbacks of ungrouped bindings.

One method is to add an auxiliary mapping:

$$Contour \rightarrow Binding\ Set$$

that gives the set of bindings associated with each scope. Each time a declaration is added or removed, this mapping must be updated.

Another possibility is to define a visibility class vis_i such that, for a scope S, only bindings local to S are visible with visibility class vis_i. This provides an immediate solution to the first

Grouped Bindings	Separate Bindings
(+) Easy to check whether a binding of a name exists within an arbitrary scope. Necessary for import, export, selected reference.	(-) Difficult
(+) Easy to get all bindings declared in a given scope: Needed for scope open	(-) Difficult
(+) Easy to check whether a binding of a name exists in the current scope. Necessary for check for redeclaration.	(-) Difficult. (+) But can use level numbers to determine what scope a binding was declared in, and thus check for re-declaration.
(-) Maintaining proper order of declarations in declaration-before-use requires a separate mechanism. Bindings must be ordered within a scope, in addition to the ordering of the scopes.	(+) Can treat a new declaration with declaration-before-use as a new scope entry. Both types of ordering (scope-level and declaration-level) can be handled by the same mechanism.
(-) Doesn't correspond well to the Visible Set Model. The two levels of ordering make it more difficult to describe the correspondence between visibility regions and visible sets. They also make it more difficult to compute visible sets.	(+) Corresponds well to the Visible Set Model. Ordering of declarations corresponds (usually) to an ordering of transformations on visible sets.
(+) Works well with Search Model.	(+) Works well with Search Model.
(-) Doesn't correspond well to the Range Model. The two levels of ordering make it more difficult to define the visibility range of a binding.	(+) Corresponds well to the Range Model. Ordering of declarations corresponds (usually) to an ordering of visibility regions, and thus an order for propagation of visibility of a binding.

Table 4.1: Comparison of Grouped and Ungrouped Bindings

problem with separate bindings in Table 4.1. It provides a very inefficient solution to the second problem. However, a more efficient solution will be made apparent in later chapters.

The grouping of bindings can be viewed as an optimization, particularly in the Search Model. The grouping in the model facilitates grouping in the implementation. A group of bindings can be implemented as a hash table, so the search has fewer steps than if all bindings are ungrouped.

4.4. Contours: Design Decisions and Terminology

This chapter has discussed many language features and the resulting requirements on a satisfactory model of visibility control. Some of the limitations of the models of visibility control, as presented in Chapter 3, have also been discussed. This section presents some design decisions resulting from these considerations, along with some definitions that will be useful later. The decisions described here will apply throughout the remainder of this dissertation, except where stated otherwise.

Ungrouped bindings have two main advantages:

(1) All of the models considered thus far can be easily handled.

(2) Both the order of scopes and the order of declarations within scopes can be represented with a single, simple mechanism.

Therefore, ungrouped bindings will be assumed from this point on. Grouped bindings will be considered again as an implementation optimization. Contours are now defined, based on this decision.

Definition 4.2. Scope Contour

Each scope in a program has a corresponding contour: the referencing environment created by entering that scope. The contour corresponding to a program scope is called a *scope contour*.

Definition 4.3. Declaration Contour

When a language requires declaration-before-use, each new declaration creates a new contour which is the referencing environment after that declaration takes effect. The contour corresponding to a declaration is called a *declaration contour*.

This view of declarations is not common – however, it is not without precedent. In the definition of Euclid [Lampson *et al.* 1981], which has declaration-before-use, the meaning of each declaration is described in terms of that declaration creating a new scope in which the new declaration is visible.

Most of the terminology for scopes and declarations in §2.3 also applies to scope and declaration contours. The declaration contours associated with the declarations local to scope S are said to be local to S's contour. One contour C_1 is global to another contour C_2 if C_2 depends on C_1. In a language with declaration-before-use, the order of nesting of declaration contours corresponds closely to textual order of declarations.

There is no fundamental difference between a scope contour and a declaration contour, but it is useful in practice to distinguish between the two because there are several differences in the way they are handled in most languages. One difference involves the shadowing of bindings. The two kinds of contours have different effects on shadowing.

Suppose we have a language that allows shadowing and declaration-before-use. (Euclid is one of the few languages with block structure and declaration-before-use that does not allow shadowing). A binding of a name in one scope can shadow bindings of the same name in outer scopes, but not bindings of the same name declared in the same "normal" scope. The difference is easily explained in terms of scope and declaration contours. A binding b_1 of the name "N" introduced within scope S's contour or within a declaration contour local to S can shadow binding b_2 of N global to S. However, b_1 cannot shadow binding b_3 of N also within scope S but introduced in a contour outside the one in which b_1 is introduced. The area immediately local to a scope is usually a single name space in which multiple declaration of a single name is an error.

There are also two forms of shadowing, as discussed in §4.2.3. In the first form, b_2 is visible up to the declaration of b_1, where it becomes shadowed by b_1. This is called *declaration contour-relative shadowing*. In the second form, b_2 is visible only up to the beginning of the scope S containing b_1. This is called *scope contour-relative shadowing*.

Scope and declaration contours differ in other ways also, but these other differences tend to vary more from language to language, and the shadowing example illustrates the kind of difference that may occur.

4.5. Correspondence of Scopes, Contours, and Visibility Regions

This section reviews these three kinds of objects in order to eliminate any confusion about their correspondence. A scope is a syntactic unit, such as a block, a procedure, a module, or a

record, that can have declarations associated with it that can create bindings. The contour associated with a scope defines the referencing environment associated with that scope. By definition, a visibility region is a region of a program that has the same referencing environment throughout. Thus, each contour C has a visibility region vr associated with it, where C defines the referencing environment within vr.

The difference between a scope and a visibility region is a matter of precision: a scope is a syntactic unit normally bounded by enclosing markers (such as **begin** ··· **end**), containing an arbitrarily large amount of program text, often including nested scopes. Using terminology such as "the referencing environment within scope S" is only meaningful if we define the precise locations within S where that referencing environment is valid, taking into consideration nested scopes and so on. The concept of a visibility region is the step to achieve the necessary precision.

Another complication arises with the definition of a scope in a language with declaration-before-use. Euclid is such a language, and each new declaration creates a new scope starting immediately after the declaration and ending at the end of the current "regular" scope (a procedure, a module, etc.). The distinction between the two kinds of scopes is important in a language that allows shadowing, as discussed in the preceding section. So, the term "scope" is reserved to refer to a "regular" scope. In a language with declaration-before-use, a scope may have many contours associated with it: each new declaration results in a new contour, and a new visibility region corresponds to that contour.

4.6. Summary of Requirements

The general and specific requirements of a model of visibility control are listed in Table 4.2. Some of the individual requirements discussed are coalesced in the table where appropriate.

General Requirements
• The model should be *natural.* • The model should be *complete.* • The model should be useful for reasoning about the visibility control rules of a language. • The model should be suitable as the basis for a formal specification method from which an implementation can be generated.
Specific Requirements
• A model of visibility control should support *names*, *entities*, and *bindings*. • There must be a mechanism for defining and retrieving attributes of an entity. • The concept of a binding in the model must support a general *n:m* mapping between names and entities. • The model of visibility control should support a general concept of a visibility region, including visibility regions corresponding to both scope contours and declaration contours. Different kinds of scopes, containing one or more visibility regions, must be representable. • It must be possible to declare a binding in an arbitrary contour in a program. • The model of visibility control must provide a mechanism for defining whether two bindings clash. • The model of visibility control must provide the ability to define more than one visibility class, for use in different contexts. • The model of visibility control must be able to represent the order of visibility constructs and the order of references with respect to visibility constructs. • A model of visibility control must support variable attributes. • A model of visibility control must be able to support a general form of inheritance, including: (1) arbitrary order of inheritance of visibility of bindings from one contour (the providing contour) to another (the inheriting contour), with potentially a different inheritance ordering for each class of visibility, (2) non-local inheritance, (3) precise control over which bindings are inherited from a providing contour to an inheriting contour. and (4) the ability to select among the bindings inherited by a contour based upon some ordering. • It must be possible to represent visibility control information independent of the processing of any specific source code.

Table 4.2: Requirements of a Model of Visibility Control

CHAPTER 5

Choices in Designing a Model of Visibility Control

The purpose of this chapter is to discuss the possible decisions in designing a model of visibility control. Some of the decisions are specific to models of visibility control, while some are more general, and apply to any specification or modeling effort. We seek the answers to the following questions for each possible design choice:

- Is this really an important *conceptual* design parameter, or is it really just an implementation issue that can be largely or completely hidden from the person specifying the visibility control rules of a language? Is there some other related design parameter that is better? Issues to be considered here are how the design parameter affects the style of a specification: how does the style affect the thinking of a language designer, for example?
- Is this design parameter independent from the other design parameters, or is it dependent on some other decision?
- What is the model of visibility control resulting from each possible combination of decisions? Is the model reasonable, and what are its advantages and disadvantages?
- How do these models relate to Reiss's and Wolf's models of visibility control, and the other existing models of visibility control? Do the choices suggest models that are better than any of these existing models? Are there other models more representative of the possible design parameters?

5.1. The Flow and Database Models of Information Maintenance

There are two primary models of maintaining information about a program being processed (the subject program), corresponding closely to the two primary methods (batch and interactive) of doing the processing. These models of information maintenance are called the *flow model* and the *database model*, respectively. These two models are of interest in a discussion of visibility control because visibility control rules are commonly implemented using both models. We must understand the models to be able to understand the range of choices available for the implementation of a visibility control rules specification for a language.

These models and their differences are described in depth in the following sections. These models are presented at this point because an eventual goal is a specification language that can be translated into a practical implementation, and therefore it is necessary at times to consider implementation issues when designing and discussing models of visibility control. There is a natural tendency to consider these issues only implicitly, sometimes making invalid assumptions based on "folklore" or other factors, often resulting in a model that depends too much on these assumptions about implementation issues. It is preferable to expose any assumptions about implementation at the beginning, so that their validity can be checked, and the influence of any invalid or unnecessary assumptions can be eliminated. Indeed, this analysis can aid in discovering the proper level of abstraction for the problem under consideration.

I make the preceding observation with the benefit of hindsight. Early in the progress of this research, use of the flow model was assumed, when the emphasis was more on implementation techniques. This assumption resulted in many difficulties in the development of a good model of visibility control, and these difficulties weren't resolved until I realized that an unnecessary and restrictive assumption was being made.

5.1.1. The Flow Model

The flow model receives its name from the way batch processing of a program works. The most common example of the flow model is found in a conventional compiler. We can view the subject program as "flowing" through the compiler, with the compiler maintaining information about the subject; in particular the compiler's information at any given time is accurate only at the location in the program currently being processed. This is how symbol tables traditionally have been implemented – with a monolithic information structure that is modified each time a declaration is processed, a scope is entered or exited, and so on. This is sometimes called *single-threading* [Schmidt 1985].

Alternatively, we can view the information about the subject as flowing *through the subject*, being updated as necessary so that at any point in the subject program, the information at that point is accurate. This alternative view may seem more natural for some attribute grammar implementations of programming languages, which do not on the surface describe a batch computation, but which use compilation techniques similar to those used by a batch compiler, and are indeed often implemented as batch processors.

The primary distinguishing characteristic of the flow model is that the value of the information structure is defined for any point in the subject program as a modification of the value of the information structure prior to the last construct in the subject that required a change in the information structure. In other words, a stream of changes to the information structure (caused in a symbol table by declarations, scope entry and exit, etc.) defines a stream of information structures (symbol tables), each valid after the program construct (visibility construct) that caused the production of that information structure.

In the implementation of the visibility control rules in a conventional compiler, the stream of resulting symbol tables is only conceptual. Each successive symbol table is defined in terms of a modification of the previous one, but is produced in the implementation by destructively modifying the previous one. This optimization is possible because the previous table is no longer needed after the modification.

5.1.2. The Database Model

When editing or viewing a program with an interactive system, the user can access the elements of the just-described conceptual stream of information structures in any order. For the sake of efficiency, it is prudent to maintain information about the entire program, using information about various parts of the program as necessary, so that we need not entirely recompute each element when it is needed. A good example of such a language processor is a language-oriented program editor such as PAN [Ballance *et al.* 1987]: the user is free to move about the subject program, making changes, and the editor must obtain the correct information in any location to determine whether the changes are legal. I call this type of information management the database model because the language processor must maintain a database of information about the entire subject program, accessing and updating information in that database according to the current location in the subject program. The database may be monolithic, or it may be highly distributed, as in some attribute grammar implementations [Reps and Teitelbaum 1985]. In this case, the database consists of the attributes spread throughout the program tree. Interactive programming systems do not have to use the database model, but they usually do because of the advantages of the database model for incremental update.

An easy way to determine whether the flow model or the database model is being used for a model of visibility control is to look at the operations: the flow model will have operations like *Enter_Scope*, *Exit_Scope*, and won't use explicit subject program positions. However, it is possible to have a *Set_Current_Position* operation. The database model will have operations like *Create_Scope*, but *not Enter_Scope* and *Exit_Scope*, and will use explicit subject program

positions (either with each update or access operation, or in a *Set_Current_Position* operation).

5.1.3. Static vs. Changing Subject Program

The flow model is traditionally used only in batch processing, with a static, unchanging subject program. The use of the flow model actually arose out of the nature of sequential batch processing of a program. In sequential batch processing, it is only necessary to keep the information necessary for the current point in the computation, so it is natural to destructively modify the information structure as the computation progresses. In other words, the stream of input is processed, producing a (conceptual) stream of information structures, each one corresponding to a particular point in the input stream.

However, using the flow model is usually not efficient when processing the subject program interactively, because a change at any point can affect the value of the information structure at any later point, possibly causing much recomputation. It may be possible in some cases to avoid most of this recomputation through the use of clever data structures [Driscoll *et al.* 1986; Hoover 1987].

Since an efficient implementation of an interactive system is difficult using the flow model, the database model is usually used when interactive processing is required. All necessary information is always present in the database, and context (*e.g.*, the user's position in the program being edited) is used to determine which information in the database is currently relevant (such as the current set of visible bindings).

The flow model style of implementation is actually an optimization of the database model intended for batch processing – the optimization is to keep only the information that is necessary at any given point in the sequential processing of a program. The database model is very general, allowing a wide range of operations and processing styles (wholly batch or interactive, or anything in between). However, this generality is achieved at the cost of some efficiency.

Several optimizations are commonly used in the flow model. The most important space optimization is the destructive modification of an old version of the information structure to produce a new version, because the old version is no longer needed. The database model must, in some form, maintain all of the information from all of the sequence of symbol tables as defined by the flow model.

One optimization of the flow model as compared to the database model is that it can throw away information that is no longer needed as it processes a program. On exiting a scope, all information about that scope's bindings can be deleted from the symbol table if the language requires declaration-before-use (except in cases such as exports). It may be possible to implement a symbol table as a stack. One common implementation method for symbol tables in the flow model is to compress the above stack into a single hash table.

5.1.4. Implicit vs. Explicit Source Positions

In the flow model, the current position within the subject program for which the information structure is accurate is implicit, embodied within the state of the information structure. For example, with a flow-style symbol table, when a lookup is performed, the position of the lookup in the subject program is determined solely by the current state of the symbol table at the time of the lookup.

In the database model, it is possible to access information about any location of the subject program by moving around in the subject. When a new variable reference is added to the subject program, the symbol table database is examined to determine whether the variable is visible in the visibility region containing the reference. That is, we need to know what bindings are visible at any point in the subject program. Since it must be possible to reference information about the program at any point in the program, we must explicitly tell the database what source position we

want information for. This can be done by supplying a source position for every database operation, or by having the database assume the source position is unchanged until told otherwise.

5.1.5. Model of Subject Program Structure

An information structure based on the flow model need only maintain information about the subject program that is currently relevant, and thus may only need to understand a particular view of the program. A flow model symbol table (in the usual case) is only concerned with the currently visible nested scopes, and thus its view of the program can simply be a list of contours corresponding to those scopes.

With the database model, the information structure must be capable of handling the structure of an entire program. In the symbol table example again, a block structured program consists of a tree of scopes, so the symbol table's view of the program will have to be tree structured. In the presence of import/export and named scoping, the tree becomes a DAG. In more complicated examples, such as an early version of LISP Machine LISP [Weinreb and Moon 1981], the DAG may have to be replaced by a more general graph.

5.1.6. Flow Model and Database Model as Design Choices

This section considers whether this (flow model vs. database model) is an important design parameter for models of visibility control.

The flow model is first analyzed with respect to the normal flow of visibility control information and dependencies in a program of a given programming language. Some visibility constructs have no affect on any name resolution or other visibility control-related language constructs following that visibility construct. For example, an ALGOL 60 block may contain declarations, but those declarations have no affect on visibility past the end of the block. The visibility at the point immediately following the block is precisely the same as at the point immediately prior to the block. The proper way to model the visibility control rules of ALGOL 60 using the "one visibility region $\leftrightarrow$ one symbol table" paradigm is as a *tree* of symbol tables, not as a simple sequence. The traditional flow model where a single sequential thread of symbol tables is produced is an optimization: each element of the sequence is produced by destructively modifying the previous element, so that only one symbol table is kept at any time. The sequence is obtained by a pre-order traversal of the tree of symbol tables just mentioned. In traditional symbol table implementations [Graham *et al.* 1979], when returning after traversing a branch of the tree, the information added to the symbol table by that branch is deleted, restoring the symbol table to an earlier state. This necessitates that the symbol table be structured so that this deletion of information is easy. Some degree of abstraction must be given up for the sake of efficiency in order to do this linearization.

The concept of a tree of symbol tables (before it becomes corrupted by linearization for efficiency) is very useful. Each node in the tree represents the visibility control information (the symbol table) for a particular visibility region of the subject program, and that visibility region alone. Each node may be related to other nodes, but only a visibility region's node is needed in order to perform any visibility control operation within that visibility region. Each node is a *view* of a specific visibility region of the subject program, a view concerning only visibility control information. If one can define each such view, and the relationships between views, then the goal of defining visibility control rules has been reached.

The database model is now considered with respect to these ideas. Instead of many symbol tables, we have a single database that contains all visibility control information for the entire subject program. To perform a visibility control operation, we specify the operation and where the operation occurs. The database model must provide a *view* of the visibility control information, and the operation must be performed with respect to that view, possibly altering the view. The

view is determined by the state of the database and the position in the subject program.

This "view" seems to be the proper level of abstraction. At this level, the database model and the flow model are identical, because they can both provide the same view, and all visibility control operations can be performed with respect to that view. To define the meaning of the visibility control rules of a language, one must define this view for any visibility region of a subject program. It may be useful to define the relationships between views, because:

- It may be easier (less work, better abstraction, etc.) to define each view in terms of other views rather than from scratch.
- It may help in producing an efficient implementation. The implementation may be able to maintain differences between views, instead of maintaining each view separately. Traditional operations such as *Enter_Scope* and *Exit_Scope* just define the relationship between two views (at an implementation level), and can be implemented efficiently because they define one view as a transformation of another.

If the model of visibility control can be defined at this level of abstraction, we can push the distinctions between the flow model and the database model entirely into the implementation. It is then the responsibility of any such implementation to provide the necessary efficiency.

5.2. Specification Burden on Definition vs. Use

This design coordinate was discovered by examining the differences between the Search Model and the Visible Set Model. In the Search Model, the result of a declaration or some other visibility construct is usually very simple: a binding is added to a contour, or another action of the same level of complexity. Specifying the meaning of each visibility construct is correspondingly simple. However, this is only a partial definition of the visibility control rules of a language. In the Search Model, a large part of the specification is in the definition of how references are resolved, both when resolving a reference and when checking for duplicate declarations.

In the Visible Set Model, resolving a reference is trivial. All of the specification effort goes into defining the meaning of each visibility construct. This is also true in the Range Model.

The difference arises because of a fundamental difference between definition (corresponding to a visibility construct) and use (lookup). The different models here arise from different emphasis on the two operations. In the Visible Set Model, definition is the operation of interest. In the Search Model, use is emphasized, although definition is still of interest. This is obviously not a binary choice, but a continuum, and though the Visible Set Model is at one extreme, it is unclear whether the Search Model is at the other extreme. So, what *is* at the other extreme is of interest. The location of the Search Model on the continuum is also of interest.

At one end of the continuum (emphasis on use), definition is trivial, and the entire specification burden is on defining the meaning of a use (lookup). The Search Model fulfills this requirement at least in the basic case of simple declaration and lookup in a block structured language. Declaration consists of three operations: entity definition, binding of the entity to a name, and association of the binding with a contour.

The first two steps are simple and completely local, not dependent on other parts of the subject program. The third step is simple in a block-structured language: The binding is added to the current contour after checking for duplicate declaration. The check for duplicate declaration is done with a lookup using special search rules, reflecting the different visibility for "knownness" (§4.2.3). The only apparent way to simplify this would be to eliminate the check for duplicate declaration, and to do this check somehow at the point of lookup. However, this only makes sense if duplicate declaration is never an error in a language. The only languages I am aware of with this property are macro languages, which usually have very simple visibility rules, and thus aren't of much interest.

More complex language constructs are now considered. One example that illustrates some of the issues involved is the Ada **use** statement. The public parts of one or more packages may be "opened", importing the public bindings into another package. The rules for resolving the clashing of bindings in two or more packages were presented in §2.6.8, and are repeated here for ease of reference:

(1) The set of "potentially visible" bindings consists of all non-private bindings of all packages appearing in **use** statements affecting S.

(2) A potentially visible binding b_1 is not visible if another binding b_2 clashes with b_1, and b_2 is visible within S in the absence of any **use** clauses.

(3) If two potentially visible bindings clash, then all potentially visible bindings with the same name are not visible. That is, all potentially visible bindings of a name are visible if they are all overloadable, and none of them are visible otherwise (assuming part (2) does not apply).

(4) All other potentially visible bindings are visible.

This description corresponds closely to the description of the effect of **use** statements in the Ada reference manual [USDoD 1983]. Another, easier-to-understand method of describing their effect exists which is unrelated to the method of description in the manual:

(1) For a given scope, collect all packages opened by **use** clauses affecting that scope. This includes all **use** clauses in that scope and in all enclosing scopes. If any of these scopes is a package body, then **use** clauses in the corresponding package specification are also included.

(2) Collect all public bindings in the packages opened by the **use** clauses in step (1). For each name in this set of bindings, if two or more bindings of that name clash (are homographs), then all bindings of that name are not visible.

(3) Place all bindings not eliminated in step (2) in a scope enclosing the package STANDARD. Package STANDARD is a scope containing predefined types and operations enclosing every library unit, and consequently the main program. The scope enclosing STANDARD can be called the "external" scope. Note that there is potentially a different external scope for each scope, because nested scopes can contain **use** clauses, and thus each scope can potentially be affected by a different set of use clauses (in step (1) above).

(4) Given this external scope, bindings in the given scope can be resolved using the normal visibility control rules, without regard for **use** clauses.

There are three options for representing the result of the **use** statement in the visibility control structure of the subject program:

(a) Add the bindings made visible by the **use** to the current contour, using the first method described above for determining these bindings.

(b) Compute the contents of the "external" scope for each scope, and make these external scopes enclose the main body of the program. The bindings in these external scopes are then made visible by the normal inheritance rules. Since each of these external scopes appears to enclose the global scope of the program, there must be a way to select the appropriate external scope for a lookup occurring in a particular scope.

(c) Modify lookup to search the packages opened by **use** clauses and to resolve the effect of clashing bindings in these **use** clauses.

(a) is more similar to the way other (simpler) imports are handled in the Search Model (as previously described), and is the method presented in §2.6.8. (c) is more in keeping with the philosophy of placing the burden of specification on the definition of lookup. Unfortunately, it is more difficult to describe the above rules in terms of a search, because all imported packages must be

searched before deciding whether a binding found by the search is visible (this is possible, of course, but it's not very clean). (b) is a compromise: part of the effect of **use** clauses on each scope is computed in advance by computing the contents of the external scope for that scope, but the effects of shadowing are not determined, as they are in (a). By making choices such as this one, we move in one direction or another along the continuum of emphasis on definition or use.

Another language feature is the presence of open and closed scopes. In the Visible Set Model, the closed scope serves as a barrier to all bindings whose class the scope is closed to. All other bindings in the visible set for the region immediately outside of the closed scope are in the visible set for the region immediately after entering the closed scope. The specification of lookup is unaffected, because the visible set inside the closed scope reflects properly the set of bindings visible there.

In the Search Model, a closed scope is handled by terminating the search when reaching a closed scope, if the scope is closed to all classes of bindings that are valid results of the search. The presence of a more general form of inheritance, where an inheritance restriction function determines whether a given inheritance link is open to a specific binding or class of bindings (as discussed in the Imports and Exports sub-section of §4.2.6), illustrates a slight weakness of the Search Model as thus far described. The search for a binding of a name may cross a partially closed inheritance link (a scope boundary). Unless it is known that the scope is closed to all possible valid results of the lookup, then the search must continue, even though the scope may turn out to be closed to any bindings found on the search. One must continue the search, and screen any bindings found. A simple and general method of doing the screening is to add a predicate to a search – in addition to matching the referenced name, the binding must satisfy the predicate. This is called the *search predicate*. Each time a partially restricted inheritance link (such as a partially closed scope) is encountered, the predicate is augmented to further restrict the predicate as required. The search predicate is simply the conjunction of all inheritance restriction functions of inheritance links on the path from the reference to the point where the binding is found. The search may halt when the search predicate becomes identically **false**.

5.3. Where is Binding Visible vs. What is Visible Here

This discussion arose out of one of the major distinguishing characteristics in the models discussed thus far. The Visible Set Model defines what bindings are visible in any visibility region of a program. The Range Model defines in which visibility regions any given binding is visible. I call these the What Visible and the Where Visible approaches, respectively. There is a very strong equivalence between these two ways of looking things, and this section explores their similarities and differences. section. The Search Model and Reiss's model of visibility control [Reiss 1983] are also discussed.

Each of these models of visibility control is first defined:

The Visible Set Model

The Visible Set Model defines which bindings are visible in a visibility region by defining a binding set containing the visible bindings (the visible set) for each visibility region of a program. That is, it defines for any program of a given language, a function

$$What\_Visible : Region \rightarrow Set\ of\ Binding$$

Lookup is done simply by examining the visible set. The function

$$Lookup : Name \times Region \rightarrow Set\ of\ Binding$$

is defined as

$$Lookup\,(n:Name, r:Region) = \{\, b:Binding \mid b \in What\_Visible\,(r) \wedge b.name = n \,\}.$$

One can define each visible set in terms of another visible set (or sets), where declarations or other visibility constructs are responsible for the differences between visible sets for different visibility regions.

The Range Model

The fundamental characteristic of the Range Model is that it defines the visibility control rules of a language, and thus the visibility in a program, by defining the *visibility range* of each binding. That is, it defines where each binding is visible in terms of the visibility regions in which the binding is visible. What I call the visibility range is sometimes referred to as the "scope" of a binding, as in the definition of ALGOL 68 [Van Wijngaarden 1976]. More formally, the Range Model defines a mapping:

$$Where\_Visible : Binding \rightarrow Set\ of\ Region$$

which provides enough information to resolve references, though not in the most convenient form. What is needed is a mapping

$$Lookup : Name \times Region \rightarrow Set\ of\ Bindings.$$

To bridge the gap from the *Where_Visible* mapping to the *Lookup* mapping, the mapping

$$All\_Bindings : Name \rightarrow Set\ of\ Bindings$$

is defined, which gives all bindings of a given name. This is a standard mapping for the model of visibility control.

The mapping

$$Restrict\_Region : Region \times Set\ of\ Bindings \rightarrow Set\ of\ Bindings$$

restricts a set of bindings to those bindings that are visible in the given visibility region. *Restrict_Region* is defined by

$$Restrict\_Region\,(r:Region, sb:Set\ of\ Bindings) \equiv$$

$$\{\, b:Binding \mid b \in sb \wedge r \in Where\_Visible\,(b) \,\}$$

$$Lookup\,(n:Name, r:Region) \equiv Restrict\_Region\,(r, All\_Bindings\,(n))$$

The definition of how to compute *Where_Visible* for any program of a given language, together with this definition of *Lookup*, define the visibility control rules for that language.

In the Range Model, each visibility construct determines whether a binding visible in a region adjacent to that visibility construct is visible in another adjacent region. There is more than one way to describe the meaning of each visibility construct, but these details are omitted for now.

The What Visible approach is represented by the Visible Set Model, while the Where Visible philosophy is represented by the Range Model.

Equivalence of the Two Approaches

There is a strong equivalence between these two approaches. The What Visible approach defines the visible set for each visibility region. The Where Visible approach defines, for each binding, a set of visibility regions where that binding is visible. The information content is the same in both cases, though the form is different: given the information defined by either model, the definition of lookup is trivial (and can be standardized across languages). This equivalence is

proven by demonstrating a one-to-one mapping between the mappings defined by each model. The demonstration is not entirely rigorous, but the validity of the demonstration should be clear.

The essential function defined by the Visible Set Model is

$$What\_Visible : Region \rightarrow Set\ of\ Binding,$$

where $(r,sb) \in What\_Visible \iff (b \in sb \iff b$ visible in region $r)$. This function is equivalent to (in the sense that they contain precisely the same information) the mapping

$$Has\_Visible\_Binding : Region \rightarrow Binding$$

where

$$(What\_Visible(r)=sb \text{ and } b \in sb) \iff ((r,b) \in Has\_Visible\_Binding).$$

This is simply a distribution of the domain of *What_Visible* over the elements of each domain element's image. The fact that *What_Visible* is a function is important, because otherwise *What_Visible* would contain more information than *Has_Visible_Binding*, and the implication would be true only in the forward direction.

Has_Visible_Binding is equivalent to the mapping

$$Visible\_In : Binding \rightarrow Region$$

where $(r,b) \in Has\_Visible\_Binding \iff (b,r) \in Visible\_In$. The equivalence here is obvious.

The essential function defined by the Range Model is

$$Where\_Visible : Binding \rightarrow Set\ of\ Region.$$

where

$$(b,sr) \in Where\_Visible \iff (b \text{ visible in region } r \iff r \in sr).$$

Visible_In is equivalent to *Where_Visible*, where

$$(b,r) \in Visible\_In \iff Where\_Visible(b)=sr \text{ s.t. } r \in sr.$$

Another way of stating the equivalence is to define *Where_Visible* in terms of *Visible_In* by

$$Where\_Visible \equiv \{ (b, \{ r \mid (b,r) \in Visible\_In \}) \mid \exists (b,r') \in Visible\_In \},$$

or *Visible_In* in terms of *Where_Visible* by

$$Visible\_In \equiv \{ (b,r) \mid r \in Where\_Visible(b) \}.$$

Therefore, *Where_Visible* and *What_Visible* are equivalent, by associativity of equivalence.

The Search Model

The Search Model is hard to classify here. What is defined is what a reference means, which in a sense defines what is visible at any given point, but only indirectly. More accurately, the Search Model defines a binding as being visible where it is declared, and also where it is explicitly made visible (*e.g.*, by an import or export that is equivalent to creating a declaration in the receiving scope). The Search Model also describes where to look for valid bindings when doing a lookup by following inheritance links. Much of the work is done at lookup, as opposed to the Visible Set Model or the Range Model, where the bulk of the specification is in the meaning of a declaration, and lookup is trivial. The Search Model is an "on-demand" type of specification – whether a binding is visible at a given point is only determined if there is actually a reference to that binding's name at that point. The only way to determine the "visible set" for

a region would be to perform a lookup on every name used anywhere in the program, with the resulting bindings comprising the desired visible set. However, it seems reasonable to say that the Search Model defines the set of bindings visible within any visibility region.

Reiss's Model

Reiss's model of visibility control is also difficult to classify. His method defines where an object is directly visible, and where it is visible using rules that specify how the visibility propagates through nested scopes. This is similar to the Range Model in that it defines *where* each binding is visible, except that the restriction by visibility is delayed until lookup.

Differences in the Where Visible and What Visible Approaches

Equivalence of information content of visibility control specifications certainly does not make them identical. Indeed, any two visibility control specifications should be equivalent at some level. The equivalence between the two approaches discussed in this section is just more clear than usual. The style of specification induced by a visibility control model is also of interest, as discussed in the introduction.

The What Visible approach places the emphasis on what is visible at any point in a program, and how that is modified by various visibility constructs. What is visible is specified in terms of what regions each visibility construct can affect, and the effect each visibility construct has on the set of visible bindings (the visible set) of each such region. This information is available as a mapping from each region to the visible set for that region.

The Where Visible approach places the emphasis on the meaning of a declaration: where the binding resulting from the declaration is visible. Where bindings are visible must be specified in terms of what bindings each visibility construct can affect, and the effect each visibility construct has on the visibility of each such binding. This information is available in the most straightforward form: a mapping from a binding to the visibility regions where the binding is visible.

On one hand, there is an emphasis on the net effect of all declarations and other visibility constructs on individual visibility regions, while on the other there is an emphasis on individual declarations and bindings.

It is my opinion that this distinction between methods of specifying visibility control rules is a valid and useful design parameter, defining a visible set for each region vs. defining the visibility range of each binding. They are significantly different ways of describing the visibility control rules of a language, regardless of their equivalence, and regardless of possible implementations. Both methods have been used to describe real languages, though not in exactly the way described, and certainly not with the same terminology. The Where Visible approach was used in the definition of Modula-2 [Wirth 1982], while the What Visible approach was used to define ALGOL 60 [Backus *et al.* 1960].

5.4. Relationships Between Design Choices

The discussion of the Flow Model vs. the Database Model resulted in the conclusion that the distinction isn't a proper one at the level of description which is the aim of this research. However, it also resulted in the conclusion that the proper level of abstraction for a model of visibility control should show no distinction between the Flow Model and the Database Model, and should provide "views" of the visibility control information.

Table 5.1 illustrates the results of combinations of decisions in the two remaining design coordinates. This table describes the model of visibility control resulting from each pair of choices. The two choices are emphasis on definition versus emphasis on use, and definition of

	What Visible	Where Visible
Definition	The Visible Set Model	The Range Model
Use	The Search Model	Reiss's Model

Table 5.1: Coordinates for Visibility Control Models

what is visible in each region versus where each binding is visible.

Two interesting observations resulted from the creation of this table. The first was that each position in the table could be filled with either one of the models formalized in this dissertation or with Reiss's model, though this was certainly not intentional.

The second was a much better understanding of Reiss's model by itself and in relation to our models. Reiss's model filled in a missing point in the coordinate space which I was aware of but was unable to characterize properly. This helped to demonstrate that these design coordinates were indeed orthogonal.

All of these models appear to be reasonable, after a less-than-complete examination. Some of their relative advantages and disadvantages will be discussed in later chapters.

One should not infer from this discussion that these are the only four major choices possible when designing models of visibility control. Indeed, three models introduced in this dissertation (the Search Model, Visible Set Model, and Range Model) are only rough sketches, with emphasis on selecting basic principles and applying those basic principles to all requirements of a model of visibility control. Reiss's model was designed independent of and prior to this research, and presumably with ease of use in mind instead of an emphasis on basic concepts, so it doesn't fit as cleanly into the coordinate space as the other three models. A designer of a model of visibility control may choose to mix some of the features of the models of visibility control discussed, blurring the distinctions between the different choices.

CHAPTER 6

The Inheritance Graph

In this chapter, all of the discussion in previous chapters is put to use to design a natural and powerful model of visibility control based on the *inheritance graph*, called the *Inheritance Graph Model*. The goal in designing this model is to satisfy the requirements presented in Chapter 4 in the most straightforward manner possible, concentrating on basic concepts. The intent is not a complete specification language that can be directly used in a programming system (such as a compiler or a language-oriented editor) to define the visibility control rules of languages. Thus, implementation details and embellishments are avoided as much as possible in this chapter, because much of the character of a specification language intended for translation depends on the details of the programming system being developed.

Likewise, commitment to any of the choices discussed in Chapter 5 is delayed or avoided as much as possible, to avoid confusing the basic concepts with other issues.

The first section informally introduces the concepts of the Inheritance Graph Model using examples. §6.2 defines each of the features of the Inheritance Graph Model more precisely.

The precise meaning of an inheritance graph is defined in §6.3, and error handling is described in §6.4. The next two sections describe in detail how the requirements of a model of visibility control are satisfied by the Inheritance Graph Model, and how many complex visibility features, including closed scopes, import and export, and the Pascal **with** statement, can be described using the Inheritance Graph Model.

§6.7 describes how an inheritance graph can be constructed if par of the construction of the graph depends on the resolution of references using the graph. Ambiguities and inconsistencies that can result from such dependencies are discussed in §6.8.

The final section is a summary of the Inheritance Graph Model.

6.1. Introduction to the Inheritance Graph Model

This section presents an introduction to the Inheritance Graph Model by means of two examples. Each of the features of the Inheritance Graph Model mentioned in this section is described more precisely and in greater detail in §6.2.

6.1.1. ALGOL 60 Visibility Rules Example

An inheritance graph consists of vertices representing visibility regions, and directed edges representing inheritance of visibility between visibility regions (corresponding to the inheritance links defined in §4.2.6). A vertex corresponds directly to a visibility region, so the two terms are used interchangeably depending on the desired emphasis. Both vertices and edges are labeled with semantic information.

A program fragment consisting of a block containing two nested blocks is illustrated in Figure 6.1. Assume ALGOL 60 visibility rules: all bindings resulting from declarations local to a scope are visible throughout the scope, so each scope is composed of a single visibility region. Each block in the program is labeled with an identifier of the form vr_n, denoting the visibility region corresponding to that block.

```
vr0
begin block1  vr1
        A : Integer
        B : Integer
        begin block2  vr2
                C : Integer
        end
        begin block3  vr3
                A : Boolean
        end
end
```

Figure 6.1: Nested Blocks Program Fragment

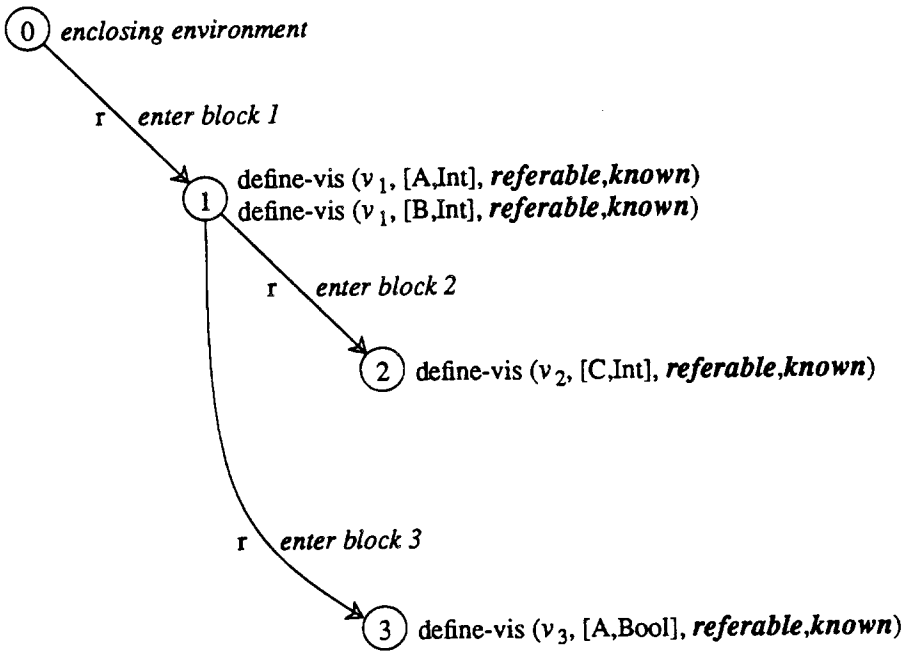


Figure 6.2: Inheritance Graph for Nested Blocks Example: ALGOL 60 Semantics

The inheritance graph corresponding to the program fragment in Figure 6.1 is illustrated in Figure 6.2. Each vertex in the graph is labeled with the number i of the visibility region vr_i corresponding to the vertex.

The describer (the person describing the visibility control rules of a language using this model) may define one or more *visibility classes*, which are used to satisfy the requirement for more than one class of visibility. For example, a binding is "visible" (following the usual definition) where it can be referenced in a program, and this visibility class can be assigned the

name ***referable***[28]. Similarly, we may say that a binding is "known" in all visibility regions where no clashing binding may be declared, and assign this visibility class the name ***known***. Since there may be more than one visibility class, and these visibility classes are independent, it no longer makes sense to say simply that a binding b is "visible" in visibility region vr – one must specify the visibility class. One says that b has visibility ***vis_class*** in vr. As a short-hand notation, one may say that <b, ***vis_class***> is visible in vr (or visible at the vertex corresponding to vr), or b is ***vis_class*** in vr. For example, one may say that b is ***known*** at vr_i. The total visibility in a visibility region can be represented by a set of binding/visibility-class pairs, following the style of the Visible Set Model. The names of the visibility classes have no inherent meaning, but rather are given their meanings by how they are used by the describer. The describer is free to use and name as many visibility classes as he/she desires.

Visibility of a binding b is initially defined at a particular vertex, usually the vertex corresponding to the visibility region resulting from the declaration that produced b. The Inheritance Graph Model primitive *define_vis* is used to define a particular class (or classes) of visibility at a certain vertex. For instance, the binding [*A*,*Int*] is defined with ***referable*** and ***known*** visibility at vertex v_1 in Figure 6.2. [*A*,*Int*] is defined to be ***referable*** and ***known*** at v_1, so <[*A*,*Int*],***referable***> and <[*A*,*Int*],***known***> are visible at v_1. Different classes of visibility of a binding may be defined at different vertices.

Visibility of a binding/visibility-class pair <b, ***vis_class***> is inherited from the vertex where initial visibility of b is defined to other vertices via edges of class ***vis_class***. Each edge represents inheritance of a single visibility class. In the example, <[*A*,*Int*],***referable***> is inherited via the edges from v_1 to v_2 and v_3. These edges are labeled with "r" in the figure to denote that they represent inheritance of ***referable*** visibility, and are called ***referable*** edges. The edges correspond to the nesting of *block2* and *block3* within *block1*, where the binding [*A*,*Int*] is declared. There are no ***known*** edges in this graph because in ALGOL 60's visibility rules, a binding is ***known*** only in the scope containing the binding's declaration: in the example graph, <[*A*,*Int*],***known***> is only visible at the vertex where ***known*** visibility of [*A*,*Int*] is defined.

Edges (also known as inheritance edges) may correspond not only to inheritance due to nesting of block-structured scopes, but also to inheritance between contours resulting from a list of declarations in a language with a declaration-before-use requirement, or inheritance resulting from an import or export statement. Different edges, corresponding to inheritance of different visibility classes, can flow in different directions in the graph. This is illustrated in the next example.

6.1.2. Pascal Visibility Rules Example

Figure 6.4 contains the inheritance graph corresponding to the same program fragment (Figure 6.1), but assuming Pascal visibility rules. The graph is significantly more complex than the graph which assumed ALGOL 60 rules, because of Pascal's declaration-before-use requirement and the use of scope contour-relative shadowing. Because a reference to a binding cannot precede its declaration, each declaration introduces a new visibility region. Visibility region vr_i follows the corresponding declaration, as shown in Figure 6.3. Figure 6.3 is identical to Figure 6.1 except for the visibility region annotations. Visibility region vr_i is represented by vertex v_i in the inheritance graph. Vertices corresponding to visibility regions local to the same syntactic scope are arranged vertically in a straight line; the vertex for a nested scope is below and to the right of the enclosing scope.

[28] The bold italic font will be used solely for names of visibility classes.

```
vr0
begin block1  vr1
        A : Integer  vr2
        B : Integer  vr3
        begin block2  vr4
                C : Integer  vr5
        end
        begin block3  vr6
                A : Boolean  vr7
        end
end
```

Figure 6.3: Nested Blocks Program Fragment: Pascal Semantics

The graph representing the Pascal-semantics version of the example program contains both ***referable*** and ***known*** edges (annotated with a "k" in the graph). To reduce clutter in the figure, when two edges (corresponding to different visibility classes) go from v_i to v_j, they are represented by a single edge in the figure, annotated with both visibility classes.

Several characteristics of Pascal visibility control semantics are apparent from the inheritance graph in Figure 6.4.

(1) In a sequence of declarations, ***known*** edges go in both directions: in sequential text order, and in the opposite direction, meaning that ***known*** visibility of a binding is inherited from the vertex where the binding is defined forward and backward to all visibility regions in the scope (excepting nested scopes). Thus, a name may not be redeclared in the same scope. However, a nested scope does not inherit ***known*** bindings, so a nested scope is free to redeclare bindings declared in outer scopes.

(2) ***referable*** edges go only forward in text order, meaning that references to bindings may only occur after their declarations. An exception must be made for a forward reference in a pointer type declaration to a type declaration in the same scope, so it is legal to reference a ***known*** binding in that case even if it is not ***referable***.

(3) ***referable*** edges also go into nested scopes, meaning that bindings declared in outer scopes may be referenced.

Thus far, the fact that the binding of the Boolean *A* declared in *block3* shadows the binding of the Integer *A* declared in *block1* has been ignored in the discussion of inheritance of visibility of bindings. The Inheritance Graph Model provides a mechanism called *clash resolution* which allows actions such as shadowing to be specified by the describer. In the example, both <[*A*,*bool*],***known***> and <[*A*,*Int*],***referable***> are inherited at vertex v_6. The specification of clash resolution for Pascal states that a ***known*** binding shadows a ***referable*** binding of the same name, so <[*A*,*bool*],***known***> shadows <[*A*,*Int*],***referable***> at v_6, and <[*A*,*Int*],***referable***> doesn't actually become visible at v_6.

In Pascal, the set of vertices where a binding *b* is ***known*** (*i.e.*, *b.name* can't be redeclared) is a subset of the vertices where *b* is ***referable***. This is true in most languages. The exception is Euclid, where a binding's name *b.name* can't be redeclared even in some places where *b* can't be referenced.

Inheritance of visibility of bindings can be further restricted by annotating an edge (v_i,v_j) with an inheritance restriction function *rf* (or simply a *restriction function*). *rf* is applied to each pair <*b*, ***vis_class***> visible at v_i, and each pair for which *rf* returns **true** is inherited by v_j.

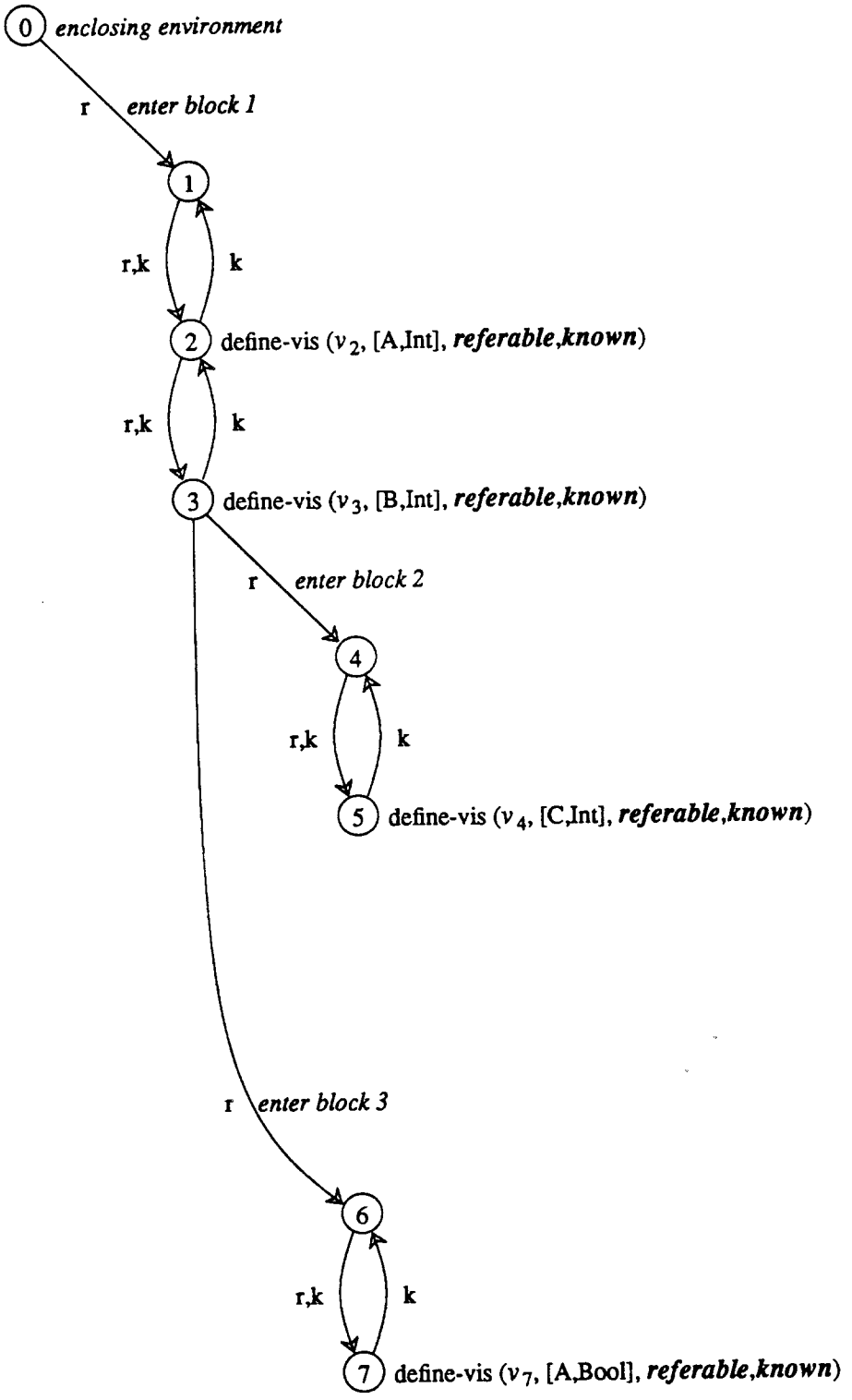


Figure 6.4: Inheritance Graph for Nested Blocks Example: Pascal Semantics

Given the above definitions, a binding/visibility-class pair is visible at all vertices where it is inherited and not shadowed. The meaning of a reference in a visibility region can be defined by the describer in terms of the binding/visibility-class pairs visible in the visibility region by defining a "lookup function." The meaning of references in different contexts (*e.g.*, a reference in an expression as compared to checking for another declaration of the same name in a scope) can be defined by defining several lookup functions.

6.1.3. Dynamic Inheritance

Both inheritance graph examples have assumed lexical (static) inheritance. However, the Inheritance Graph Model is not restricted to languages with lexical inheritance. In a language with dynamic inheritance, visibility regions still represent contours, but the structure of the inheritance graph varies dynamically with the execution of the program. The structure of inheritance graphs corresponding to dynamic inheritance is typically quite simple.

6.2. Definition of the Inheritance Graph

The Inheritance Graph Model consists of a few simple primitives, listed here:

Names, Entities, and Bindings
: As previously defined.

Visibility Classes
: Describer-defined classes of visibility.

Vertices
: A vertex corresponds to a single visibility region.

Edges
: An edge is labeled with a particular visibility class, and represents inheritance of visibility of bindings with that visibility class.

define_vis (v:Vertex, b:Binding, vc:VisibilityClass)
: Defines initial visibility of binding *b* with visibility class *vc* at vertex *v*.

redefine_vis (v:Vertex, old_b, new_b:Binding, vc:VisibilityClass)
: Defines initial visibility of binding *new_b* with visibility class *vc* at vertex *v*, simultaneously shadowing inheritance of binding *old_b*.

visible (n:Name, v:Vertex, vc:VisibilityClass)
: Returns all bindings *b* of name *n* inherited at vertex *v* with visibility class *vc*.

clash (b1, b2: Binding)
: Describer-definable function that returns **true** iff *b1* and *b2 clash* according to the visibility rules of the language.

Clash Table
: Describer-definable table that defines the meaning of the function *clash_resolve*. Describes the meaning of two clashing bindings visible at the same vertex (such as shadowing one of the bindings, or an error).

define_error_check (v: Vertex, error_check: **procedure** *(Vertex))*
: Associates the describer-defined error-checking function *error_check* with vertex *v*.

There are no other primitives in the Inheritance Graph Model. The following sub-sections define these primitives in more detail. All other functions appearing in this chapter are describer-defined functions which use the primitives of the Inheritance Graph Model to describe the semantics of a particular visibility construct.

6.2.1. Inheritance Graph Vertices

Each vertex in the inheritance graph corresponds to a single visibility region. The *net visibility* at a vertex is a set of binding/visibility-class pairs that defines what bindings are visible with what visibility classes at the vertex. The visibility at a vertex may be broken down into three parts, representing steps in defining the net visibility of the vertex. Figure 6.5 shows an expanded version of a vertex v, illustrating the three steps.

The first step collects all binding/visibility-class pairs inherited by v via edges from other vertices, taking into account inheritance restrictions on the edges.

The second step adds the effect of definitions and redefinitions of binding/visibility-class pairs. A *definition* simply adds a binding/visibility-class pair to the visibility of the vertex. A definition can be created by the *define_vis* primitive:

define_vis (v:IGVertex, b:Binding, vis_class:VisibilityClass)

which defines visibility class *vis_class* of b at vertex v.

Redefinitions, which define visibility of a binding/visibility-class pair while shadowing visibility of another binding/visibility-class pair, will be discussed in §6.2.6.

Adding the effect of definitions and redefinitions produces an intermediate visibility for the vertex. The third step is to do the clash processing: to check for any clashing bindings and perform any actions specified by clash resolution. These actions may include shadowing some

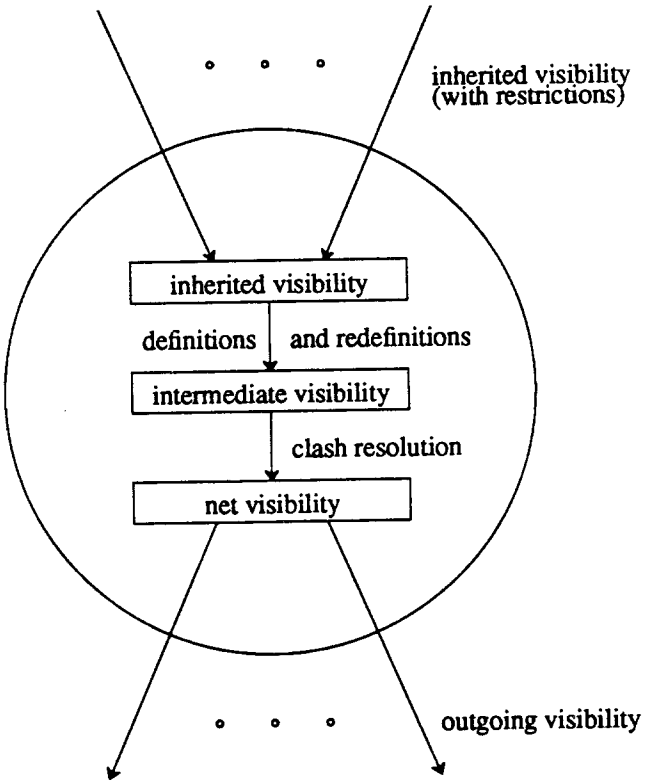


Figure 6.5: Three Steps of Inheritance Graph Vertex

binding/visibility-class pairs, resulting in a restricted net visibility. The details of the clash function and clash table, which together specify the clash resolution process, will be given in §6.2.3 and §6.2.3. This net visibility is what is visible to normal lookups performed in the visibility region corresponding to the vertex, and is available for inheritance to other vertices. The intermediate steps in defining the net visibility for a vertex can be accessed if desired, as will be discussed in section §6.2.4.

6.2.2. Inheritance Graph Edges

Inheritance graph edges correspond to the inheritance links defined in §4.2.6. They provide a way of describing restricted or unrestricted inheritance of binding/visibility-class pairs between visibility regions. Each edge e_i is labeled with a visibility class vc_i, indicating which pairs may potentially be inherited via that edge. e_i is also labeled with a restriction function rf_i. The form and effects of restriction functions will be defined in §6.2.7, but the restriction functions for the examples prior to that section do not affect the meaning of the inheritance graphs, and can be ignored.

6.2.3. The Clash Function and Clash Table

The clash function *clash* is describer-defined, and determines whether two bindings clash. It takes two bindings as arguments, and returns **true** if the bindings clash, and **false** otherwise. By the definition of clashing bindings, a binding does not clash with itself, so the two actual parameters to *clash* will never denote the same binding. Furthermore, two bindings are defined to clash only if their names are equal. This does restrict what the describer can define clash to mean, but it corresponds to the fact that names are the most important attribute of bindings used in resolving references. Thus, *clash* 's parameters will always have the same name, so it does not need to test the names for equality. The following procedure is the default clash function, and is appropriate for Pascal and for all languages where clashing is determined solely by equality of names:

```
procedure clash (b1, b2 : Binding) : boolean
    return (true)
```

In a language with overloading, such as Ada, some bindings with the same name do not clash, and *clash* must be written appropriately.

clash is required to be commutative and associative. The order in which a pair of bindings is passed to *clash* is not defined, so a non-commutative *clash* would not make sense. Associativity is required so that *clash* defines an easily-computed partition on a set of bindings.

All pairs of bindings visible at a vertex are tested for clashes, regardless of the visibility class associated with each binding. The visibility classes of the binding/visibility-class pairs are used by the clash table.

The clash table defines a function *clash-resolve* that determines how a clash between two bindings visible in the same visibility region is resolved. The resulting action depends primarily on the visibility classes. The following table is the clash table for Pascal, and specifies the clash resolution process as discussed in the introductory examples:

Pascal Clash Table		Visibility Class for b_2	
		referable	***known***
Vis Class for b_1	***referable***	error("description error")	**shadow** b_1
	known	**shadow** b_2	error("clashing declarations")[29]

This clash table should be interpreted as follows: If b_1 is ***known*** and b_2 is ***referable***, then b_1 shadows b_2, and inheritance of <b_2, ***referable***> is halted by the clash processing. The clash table is required to be symmetric across the main diagonal, meaning that clash resolution is commutative. Clash resolution is also required to be transitive with respect to shadowing, in the sense that if

$$clash\text{–}resolve(\{<b_1, vc_1>,<b_2, vc_2>\} = shadow\ b_2$$

and

$$clash\text{–}resolve(\{<b_2, vc_2>,<b_3, vc_3>\} = shadow\ b_3,$$

then

$$clash\text{–}resolve(\{<b_1, vc_1>,<b_3, vc_3>\} = shadow\ b_3.$$

If both bindings are ***known***, there is an error, because this means that both bindings were declared in the same scope. The only other possibility is that both bindings are ***referable*** – this should never occur in the inheritance graph for Pascal. If if does, it indicates that there is an error in the description of the inheritance graph for Pascal.

The clash table for ALGOL 60 is identical to the clash table for Pascal, except that two ***referable*** bindings visible in the same visibility region do not signify an error in the description, an are therefore ignored.

Each clash table entry may also be a conditional statement, if the visibility classes of the bindings alone is insufficient information for selecting the appropriate action.

Clash processing is used to define a general form of shadowing. In previous chapters, shadowing was described as the result of the interaction between visible bindings. The concept of shadowing has been extended to include multiple visibility classes. The clash table describes the meaning of interactions between various visibility constructs of clashing bindings, and shadowing of one of the bindings is just one of the possible results.

6.2.4. Definition of Lookup

Lookup is defined by the describer in terms of the basic visibility classes used in the definition of the visibility control rules of the language. The primitive function *visible* is provided for use in defining the meaning of lookup. Unless otherwise stated, in any mention of what is visible at a vertex, the net visibility is intended. The describer may define more than one lookup function in different contexts. *visible* is defined as follows:

```
function visible (
    v          : IGVertex,
    n          : Name,
    vis_class: VisibilityClass) : Set of Binding;

    return ({b:Binding | b.name = n,
            <v,vis_class> inherited by v,
            taking into account definitions and redefinitions,
            inheritance of visibility via inheritance graph edges,
            restriction functions, and clash resolution})
```

[29] This entry ignores the possibility of forward declarations, and will signal an error for a legal forward declaration. A revised version of this table entry will be given in §6.5.2.

Note that this does not define how this set of bindings is computed, or even a precise definition of the set. These issues will be discussed in later sections on the precise meaning of an inheritance graph.

Pascal requires two separate lookup functions – one for normal references, and one for use in the special case of reference while defining a pointer type, since forward references in the definition of a pointer type in Pascal are legal, as in

```
type
        StackPtr = ^Stack;

        Stack = ...
```

The lookup function for normal references is

```
function standard_lookup (
    n       : Name;
    v       : IGVertex) : Binding;

    return (restrict_unique (visible (n, v, referable), n, v))
```

The lookup function for references occurring in pointer type definitions is:

```
function pointer_type_lookup (
    n       : Name;
    v       : IGVertex) : Binding;

    return (restrict_unique (visible (n, v, referable) ∪ visible (n, v, known), n, v))
```

restrict_unique is a describer-defined function to test that the set returned by *visible* contains a single element. If so, *restrict_unique* returns that element; otherwise it creates an error message and returns *error_binding*, a special describer-defined binding used as a place-holder to reduce error-checking in other places.

```
function restrict_unique (
    bindings : Set of Binding;
    name : Name;
    object : IGObject;
    ) : Binding;

    num_bindings : Integer;

    num_bindings = size (bindings);
    if num_bindings = 1 then
        return (select b:Binding from bindings);
    elseif num_bindings = 0 then
        error (object, "'%s' referenced but not visible", name_to_ident(name));
        return (error_binding);
    else
        error (object, "More than one declaration of '%s' visible", name_to_ident(name));
        return (error_binding);
```

In the Pascal inheritance graph, if more than one binding is found, then there are clashing declarations which will be detected by the clash processing, and it may not be desirable to report errors in both the clash table and in *restrict_unique*.

In addition to *visible*, *inherited_visible* and *intermediate_visible* are defined so that the intermediate visibility steps at an inheritance graph vertex can be accessed. These primitives should not be needed often.

6.2.5. Error Conditions

It is possible to add error checks to a vertex, which are evaluated after the visibility at the node is computed. The arguments to *define_error_check*, shown in Figure 6.6, are a vertex and a procedure that performs the necessary checks, and calls error if necessary.

6.2.6. Redefinitions

A *redefinition* is the replacement of a visible binding by another binding. The new binding binds the same name, but the entity is different. The new entity usually depends on the old entity, and is often just a slight modification of the old entity. Redefinitions are used to represent variable attributes (§4.2.4). Their usage will be discussed further in later sections.

The following algorithm represents the steps involved in determining the effect of a redefinition: A redefinition can be created by the *redefine_vis* primitive in Figure 6.7.

redefine_vis redefines the visibility of an old binding by replacing it with a new binding. The two bindings are required to be identical as far as inheritance and clashing are concerned,

```
procedure define_error_check (
    v : IGVertex,
    error_check : procedure ( IGVertex ));

    add "error_check" to the error conditions at vertex v,
    to be executed after visibility at v is computed
```

Figure 6.6: Procedure define_error_check

```
procedure redefine_vis (
    v           : IGVertex,
    old_binding     : Binding,
    new_binding     : Binding,
    vis_class: VisibilityClass)

    procedure check_redefine_closure (v : IGVertex);
        if (old_binding ∈ inherited_visible (v, old_binding.name, vis_class)) then
            error("old binding not visible at redefinition point");

    if clash (old_binding, new_binding) then
        halt inheritance of visibility of old_binding
        define visibility (new_binding , vis_class)
    else
        error("Improper redefinition of old binding: bindings don't clash");
    define_error_check (v, check_redefine_closure);
```

Figure 6.7: Procedure redefine_vis

meaning that they must clash.

check_redefine_closure must be a closure, because it depends on the value of the lexically bound *old_binding*. The desired effect could be achieved in a language without first-class functions by making *define_error_check* and the other inheritance graph operators language primitives. Then, the argument to *define_error_check* could be the code to be executed, such as a function call with arbitrary arguments, instead of a function with standardized arguments.

6.2.7. Restriction Functions

Each edge (v_i, v_j) with some visibility class ***vis_class*** is annotated with a restriction function *rf*. *rf* is applied to each pair <*b*, ***vis_class***> visible at v_i, and each pair for which *rf* returns **true** is inherited by v_j. Each edge has a restriction function associated with it, but the default is the null restriction function which returns **true** for all bindings. The restriction may depend on the visibility construct corresponding to the edge, the class of the binding or entity being tested for inheritance, or other details of the binding or entity, such as the binding's name.

Restriction functions are used to alter "normal" inheritance of visibility in programs. For example, in Euclid a closed scope *S* inherits only those bindings declared with the keyword **pervasive**. All other bindings declared outside of *S* are not visible within *S*. The restriction function for the edge from the visibility region prior to a closed scope to the first visibility region in the closed scope is:

```
function restrict_pervasive_only (
    b       : Binding) : Boolean;

    return (b.entity.pervasive = true);
```

6.3. Meaning of the Inheritance Graph

Given the preceding definitions of inheritance graph vertices, definitions and redefinitions of binding visibility, edges and restriction functions, the clash function and clash table, and describer-defined lookup functions, the meaning of an inheritance graph can be reduced to a single issue: determining the set of binding/visibility-class pairs inherited by (visible at) each vertex of the inheritance graph.

The meaning can also be defined using the Search Model approach, defining how to find any binding/visibility-class pairs visible at a vertex matching a given name. However, this approach has some problems, and will be discussed later.

Each vertex defines three visible sets[30] (as illustrated in Figure 6.5), where the elements of each visible set are binding/visibility-class pairs. Each visible set is defined by a function dependent on other visible sets. The first step computes the *inherited visibility*, which is the union of all pairs inherited via inheritance edges. The second step adds definitions or redefinitions, computing the *intermediate visibility* – redefinitions replace old pairs. The third step eliminates any bindings shadowed as a result of the clash function and clash table, giving the *net visibility*.

There may be cycles in the inheritance graph, so determining a node listing giving an order to evaluate the functions to compute the visible sets is not, in general, straightforward. Determining the meaning of a particular inheritance graph involves finding a least fixed point assignment to all visible sets in the inheritance graph. If there is more than one fixed point, none of which is least, the description of the inheritance graph is *ambiguous*. If there is no fixed point, the description of the inheritance graph is *inconsistent*. Either of these problems can result from the actual

[30] In the context of multiple visibility classes, a visible set is a set of binding/visibility-class pairs.

visibility control rules of the language being described, or from an improper description of those rules in terms of an inheritance graph. §6.8 will discuss these issues in more detail.

6.4. Error Handling

An error message is produced at each vertex where two clashing bindings reach, and the clash table entry for the visibility classes associated with the bindings indicates an error. This may result in many error messages for a single program error, for example in Pascal where ***known*** edges go in both directions. If two clashing declarations occur in the same scope S, an error will be noted at every vertex corresponding to a visibility region in S.

One possible solution is to alter the result of a clash to prevent visibility of a binding from being inherited past the point of an error. This is viable, but it complicates descriptions enough to make them noticeably more difficult to understand.

Another, cleaner solution is to produce the duplicate error messages and require the error reporting mechanism to eliminate the duplicates. This requires a global error management package. The only additional requirement is that each error must define an object (such as a vertex, a binding, or an entity) with which it is associated. It is then easy to cull all errors associated with an object for duplicates. The language system of which this is a part must define where and how an error message is reported for each class of object. A multiple-declaration error would be associated with one or both bindings, and could be printed with the binding's declaration.

6.5. Handling of Requirements

This section describes how each of the requirements outlined in Chapter 4 is met by the Inheritance Graph model of visibility control. The general requirements will be discussed later. For the moment, the discussion will be restricted to the specific requirements of a model of visibility control.

6.5.1. The Basics

Names, entities, and bindings are included as basic objects of the model. The handling of attributes is not explicitly defined, but is straightforward. Bindings and entities can be defined as records, with extra fields as needed to define the class of a binding or entity, or other attributes. The n:m mapping between names and entities is supported, because there is no restriction on the association of names to entities in bindings. The meaning of associating more than one entity with a single name or vice-versa depends on the structure of the inheritance graph, the clash function and table, and so on.

Direct support for different classes of visibility is provided by two mechanisms: multiple describer-defined visibility classes, and multiple describer-defined lookup functions for use in different contexts.

Ordering of visibility constructs and ordering of references with respect to visibility constructs is represented directly with directed edges in the inheritance graph. A very powerful form of inheritance is provided by allowing inheritance edges from any vertex to any other vertex, and by allowing describer-defined functions for restricting inheritance of bindings via edges.

6.5.2. Definition of Clash, and Effect of Clashing Bindings

The describer defines the meaning of clash by defining the function *clash*. The clash table allows the describer to select the appropriate action when bindings clash. The shadow and error actions are straightforward. The different kinds of reference action can be described using the inheritance graph as follows:

If the new declaration is simply a completion of the old declaration (as in a Pascal forward declaration), the handling of the new declaration depends on whether it is possible to tell solely

by examination of a declaration whether it is a completion of a forward declaration. In Pascal, the declaration that completes an earlier forward procedure declaration looks just like a declaration of a parameterless procedure, so this isn't possible.

If it *is* possible to tell by examination that a declaration is the completion of a forward declaration, then the modification of the attribute can be separated completely from the binding resolution process. The completing declaration does not create a new binding: rather, the appropriate course of action is to wait until all visibility information is available, find the binding created by the forward declaration, and modify the appropriate attribute.

If it is not possible to tell by examination of a declaration whether it is a completion of a forward declaration, then we must handle it as if it is a completely new declaration, and check for the presence of a forward declaration if a clash occurs. Thus, the declaration creates a new binding b_{new}. If there is no prior forward declaration of the same name, then the declaration is a new one, and b_{new} becomes visible according to the usual rules. If there is a prior forward declaration, its binding $b_{\textbf{forward}}$ clashes with b_{new}. The resulting action is to shadow all visibility classes of b_{new} and to modify $b_{\textbf{forward}}$ with the new information from b_{new}.

The Pascal clash table given in §6.2.3 did not handle forward declarations properly. The entry for a ***known/known*** clash should be replaced by a call on the following procedure:

```
procedure resolve_clash (b1 , b2)
    b1,b2 : Binding;

    -- determine if one of the bindings is a forward declaration
    -- is_forward_ref is describer-defined
    if is_forward_ref (b1) then
    |   bold := b1;
    |   bnew := b2;
    else if is_forward_ref (b2) then
    |   --We assume a pointer implementation here, so modifying bold modifies b2
    |   bold := b2;
    |   bnew := b1;
    else    -- not a completion of a forward declaration
    |   error ("clashing declarations")
    |   return

    -- shadow all visibility classes of bnew
    shadow_all (bnew);

    -- update_forward_ref is describer-defined,
    -- and modifies attributes of bold with new information
    -- may include additional error checking
    update_forward_ref (bold, bnew);
```

The final possibility for a reference action is if the new declaration results in a binding of a modified version of the old entity. This must be handled as a variable attribute, as defined in §4.2.4. Variable attributes are discussed in the next section.

6.5.3. Variable Attributes

Variable attributes require the ability to create a new, modified binding b_{new} that depends on another binding b_{old} such that *clash* (b_{old}, b_{new}). b_{new} shadows b_{old} at the point of definition of b_{new}. This is done using redefinitions, as defined in §6.2.1.

Redefinitions should be done only in contexts where it is known that an old binding must already exist; otherwise there is an error. If it is not known *a priori* whether or not an old binding already exists[31], a check that an old binding is visible should be done before doing a redefinition.

A redefinition doesn't involve use of the clash table; it is placed with the definitions, and replaces the old binding with the same name. An example of the use of variable attributes in an inheritance graph will be presented in the next section.

6.5.4. General Concept of Visibility Regions and Scopes

A visibility region can be created corresponding to any visibility construct. A particular kind of scope is characterized by a pattern of visibility regions (vertices) and edges in a subgraph of an inheritance graph. Different forms of scope (records, modules, procedures) and variations in these in different languages are reflected in the form of the subgraph:

- What vertices are used, and where definitions are placed on the vertices.
- What visibility classes are used.
- What the connectivity of the subgraph is, both internal edges, and edges between vertices in the subgraph and other vertices in the inheritance graph. More specifically, a subgraph for a scope is partially characterized by what visibility classes are inherited from one vertex to another, in what direction(s) each visibility class is inherited, and what inheritance restrictions are on the inheritance edges.

This section illustrates how several common kinds of scopes and related features may be described using an inheritance graph. The task of designing the form of inheritance graphs for a programming language consists primarily of designing a few subgraph-schemata of this form, as well as the clash function and clash table.

The following examples illustrate the schemata for several kinds of scopes. These schemata may be used as a guide for designing inheritance graphs for languages that include these kinds of scopes.

Simple Open Scope

This example illustrates the prototypical subgraph for a simple open scope. The language illustrated by the example requires declaration-before-use, but unlike Pascal, a binding from an outer scope isn't shadowed until the declaration of a clashing binding in an inner scope. The graph is in Figure 6.8. "dv" edges in the graph correspond to the ***directly_visible*** visibility class. "r" edges continue to denote the ***referable*** visibility class, with the same meaning. To aid understanding of the graph, the nodes of the graph are arranged so that edges between visibility regions local to the same scope are vertical on the page, while edges from outer to inner scopes go from left to right, as in the earlier examples.

A binding *b* is ***directly_visible*** in each visibility region *vr* such that: (1) *vr* is local to the scope containing *b*'s declaration, and (2) *b* may be referenced in *vr* without qualification. According to the semantics described for this graph, ***directly_visible*** edges connect the vertices corresponding to the local visibility regions of a scope in text order. There is a ***referable*** edge corresponding to each ***directly_visible*** edge, and in addition, a ***referable*** edge from the visibility region immediately preceding scope entry to the first visibility region in the scope.

[31] In a Modula-2 implementation module, one doesn't know whether or not a type definition is a completion of an opaque type declared in the corresponding definition module.

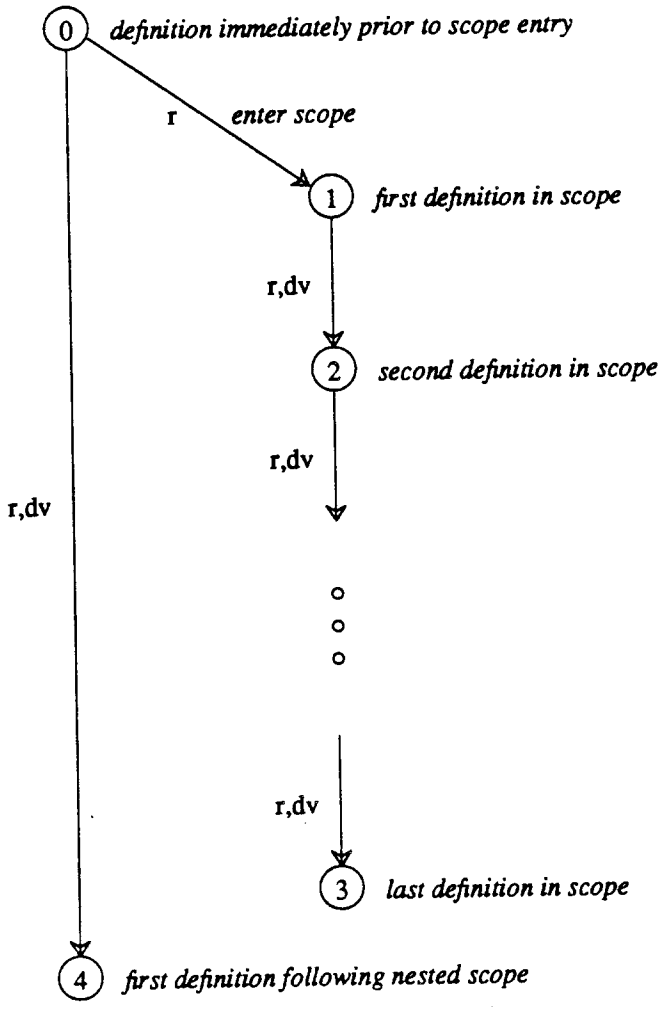


Figure 6.8: Inheritance Graph for Open Scope

Pascal-Style Record Types and Qualified References

A Pascal record type consists of a list of field declarations, each of which may in turn be another record type declaration, nested to an arbitrary depth. A record is a form of scope; the interesting characteristic of a record is that qualified references may be made to the fields of the scope. In Pascal, no qualified references to the fields of the record may be made within the nested scopes comprising the record declaration. Figure 6.9 illustrates the inheritance graph structure for a Pascal record. To avoid unnecessary detail, the fields of the record are named *f-1* through *f-n*, the fields of a nested record type at field *f-m* are numbered *f-m-1* through *f-m-n*, and so on. Also, to save space in the figure, ***referable*** and ***known*** are abbreviated by "***r***" and "***k***," respectively. The name of the record is "R".

Any record-type entity, whether a named record type or a nested anonymous record type declared as the type of a record field, has a *ref _env* (reference environment) field used for qualified references. To resolve the reference "R.f-3" in visibility region $vr_{current}$, a binding *b* of

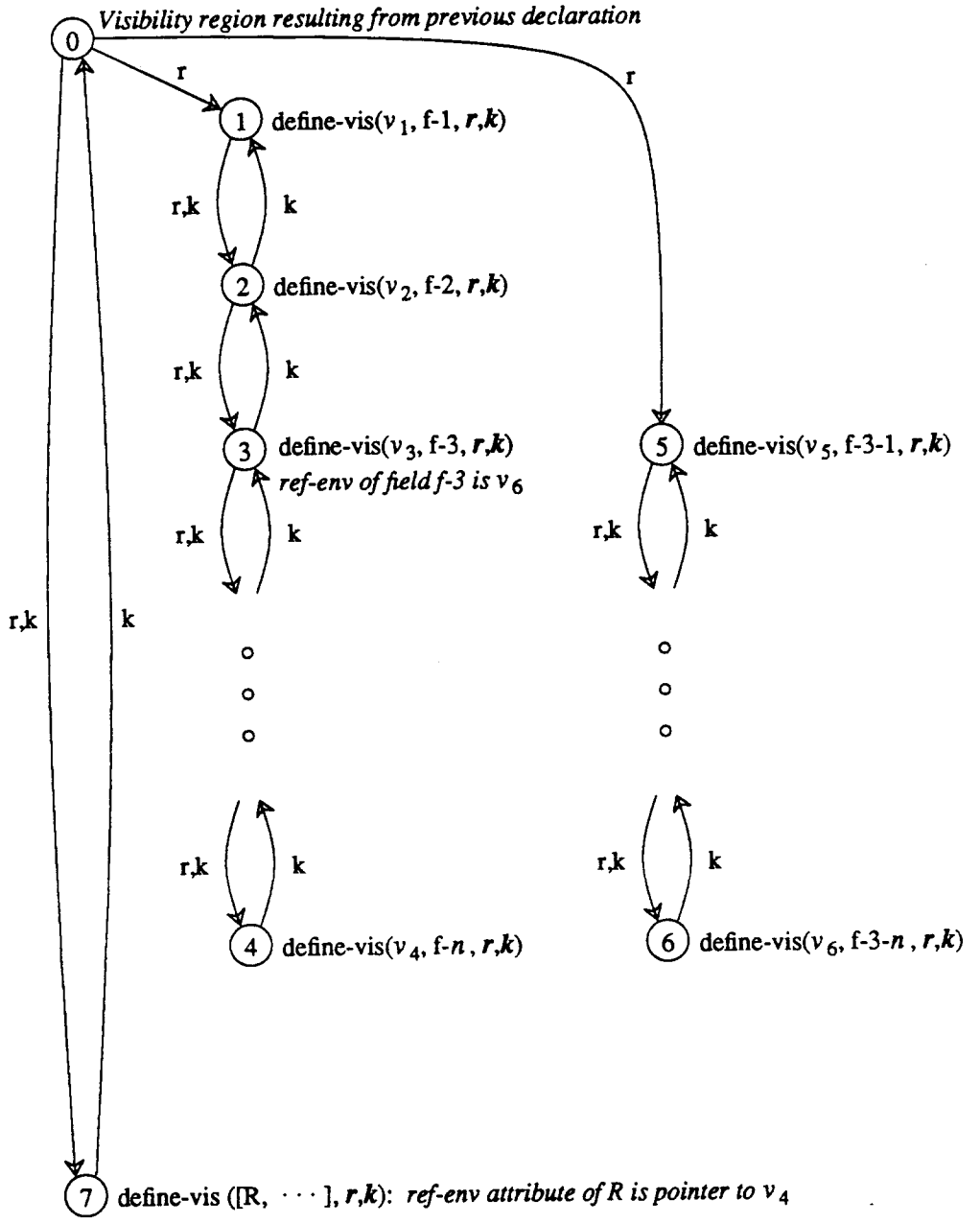


Figure 6.9: Pascal Record Type Inheritance Graph

"R" is found using the normal lookup function, *standard_lookup* (R,$v_{current}$),[32] where $v_{current}$ is the vertex corresponding to $vr_{current}$. The example assumes that *b* exists, is unique, and is a binding of a record variable. Then *b.entity.type.ref_env* is a pointer to the vertex v_R (v_4 in the figure) in the inheritance graph that is the visibility region in which the lookup up "f-2" should take

[32] such as the *standard_lookup* defined in §6.2.4.

place. A different lookup function, *directly_visible_lookup* must be used here: only the fields immediately local to *R* may be referenced qualified by "R". Using the visibility classes introduced in the Pascal semantics example, a binding is directly visible if it is ***referable*** and ***known***. The function is:

```
procedure directly_visible_lookup (
    n       : Name;
    v       : IGVertex) : Set of Binding;

    return (visible (n, v, referable) ∩ visible (n, v, known))
```

The appropriate call is: *directly_visible_lookup* (*ident_to_name* (N2), v_{N_2}). This process may be continued for any depth of qualified references.

A separate visibility class ***directly_visible*** could be used for Pascal, but it is simpler to check for the conjunction of ***referable*** and ***known*** visibility than to deal with the extra edges and interaction between visibility classes.

Scope with Nested Qualified References

As discussed in §4.2.4, a scope *S* with declaration-before-use in which nested qualified references are possible results in a new binding of "S" for each visibility region in *S*. This is in contrast to a Pascal record, where the binding of the scope can be created after the last visibility region local to the scope. A prototypical inheritance graph for a scope *S* with nested qualified references is given in Figure 6.10. This example is similar to the example in Figure 6.9, but with the necessary changes to handle nested qualified references. The explanation of the visibility classes used in Figure 6.10 follows.

The semantics of qualified references in a language that requires declaration-before-use and allows nested qualified references are significantly more complicated than they would be in the same language without nested qualified references. At each step in resolving a qualified reference "$N_1.N_2. \cdots .N_n$", we must determine the proper environment – the proper visibility region – in which to resolve the reference to name "N_i". The reference to "N_1" is resolved using the standard lookup. The reference to "N_i" must be resolved locally to the scope denoted by "N_{i-1}". More precisely, the proper binding of "N_i" is one which is ***directly_visible***[33] in the last visibility region local to "N_{i-1}" prior to the point of the reference "$N_1.N_2. \cdots .N_n$". The reference may be nested several levels within the scope denoted by "N_{i-1}". Also, declarations local to "N_{i-1}" but textually following the point of reference may not be legally referenced in the form "$N_1. \cdots .N_i$" because of the declaration-before-use requirement. It is possible that "N_1" $\cdots$ "N_i" for some *i* denote scopes that contain the point of reference, while "N_{i+1}" $\cdots$ "N_n" do not.

Resolving qualified references requires the use of another visibility class, called the ***containing*** visibility class. ***containing*** edges in the figure are labeled with a "c". The ***referable*** and ***known*** visibility classes of a binding of a scope entity *S* are initially defined in-line with the other declarations at the same level, then redefined once at the same level to reflect the *ref_env* at the end of the declarations local to *S*. The ***containing*** visibility class of *S* is defined at the first visibility region local to *S*, and is redefined at each visibility region local to *S* (*i.e.*, resulting from a local declaration of *S*). The *ref_env* for the redefined binding points to the new visibility region resulting from the local declaration.

---

[33] Continuing as in Pascal, ***directly_visible*** is the same as the conjunction of ***referable*** and ***known***, and ***directly_visible*** is used instead of the conjunction of the other two visibility classes as a convenience.

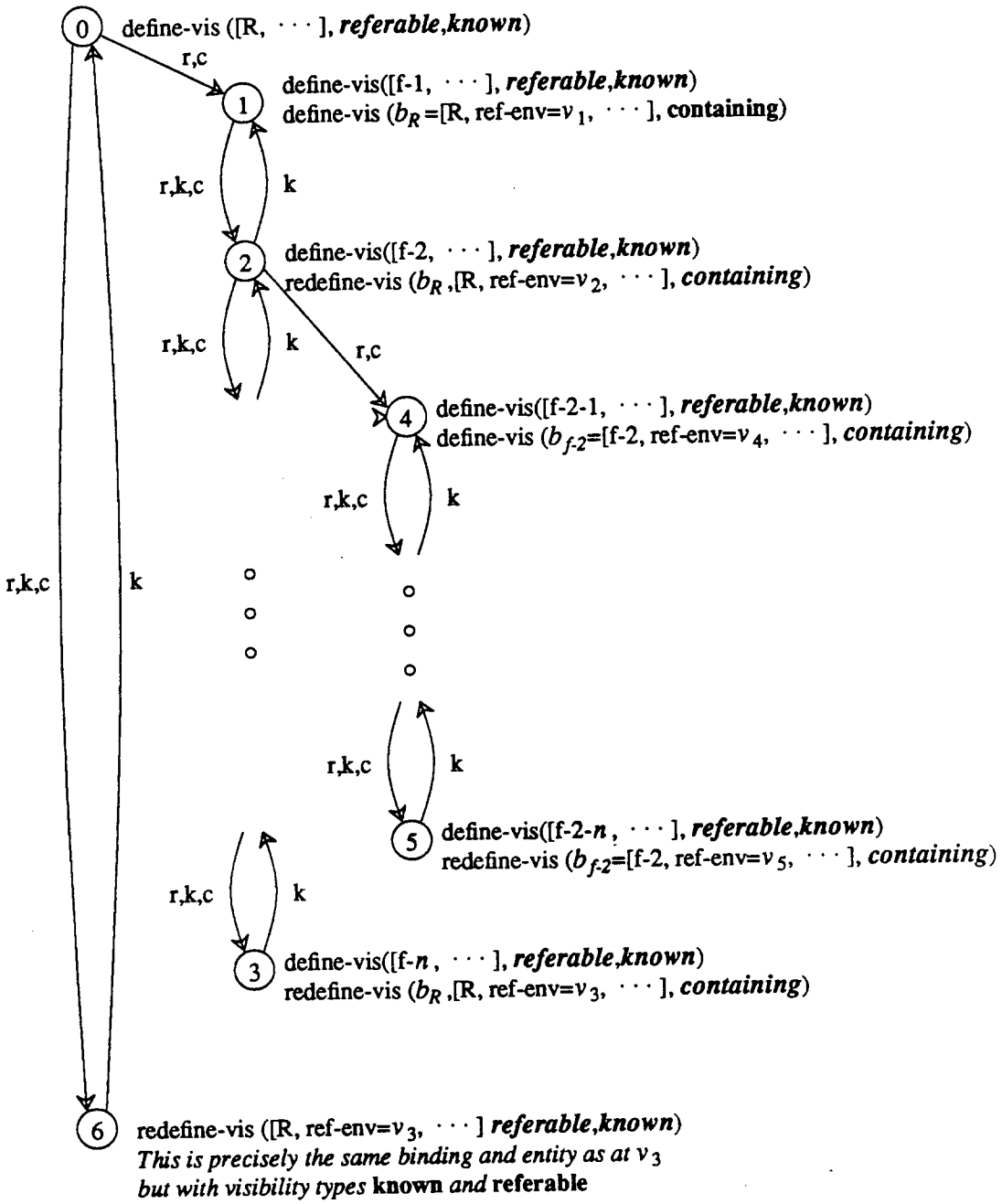


Figure 6.10: Inheritance Graph for Scope with Nested Qualified References

The clash table states that a clash between a ***containing*** binding and any other binding (including another ***containing*** binding) is to be ignored. Thus, even if ***referable*** visibility of a binding b_1 becomes shadowed by another binding b_2 that clashes with b_1, the ***containing*** visibility of b_1 is not shadowed: in any visibility region in a program, bindings for all scopes containing the visibility region are visible with visibility class ***containing***.

The ***containing*** visibility class is used to determine whether a name "N_i" in a qualified reference denotes a scope containing the point of reference, and if so, to get the appropriate binding definition or redefinition of the scope. The binding found with a ***containing*** lookup will have as the value of its *ref_env* attribute the visibility region in which the next name "N_{i+1}" of the qualified reference should be looked up using the ***directly_visible*** visibility class.

If no ***containing*** binding of "N_i" is found that corresponds to the proper scope, then the reference is not nested within "N_i". Either "N_i" is not local to "N_{i-1}", which is an error, or the point of reference follows the end of "N_i", in which case the *ref_env* attribute of the binding of "N_{i-1}" found in the previous step of resolving the qualified reference gives the proper visibility region for doing the ***directly_visible*** lookup of "N_i".

Figure 6.11 contains the definition of *resolve_qualified_ref*, and Figure 6.12 contains the definition of the auxiliary function *containing_lookup*. The *resolve_qualified_ref* function shown is for resolving qualified references in declarations in Modula-2, though its general outline is valid for any language with nested qualified references and declaration-before-use. The features specific to Modula-2 are the check requiring fields of a module referenced by qualification to be exported, and the use of *pointer_type_lookup* for a reference in a pointer type declaration. *resolve_qualified_ref* is also used in Modula-2 for simple references in declarations, which are a special case of qualified references.

Note that two lookups are performed for each name in the qualified reference. The first is either a standard lookup (for "N_1") or a ***directly_visible*** lookup, to find the correct scope or field. The second is a ***containing*** lookup, to find the correct version of the binding found in the previous step, so that we have the correct *ref_env* for the lookup of the next name.

The function *resolve_qualified_ref* is somewhat complex, but this is simply due to the complexity of the semantics of a nested qualified reference. Some of the complexity is due to additional "features" of Modula-2.

6.5.5. Defining Bindings in Arbitrary Contours

A binding can be defined in any accessible scope: a definition of the binding is associated with the vertex corresponding to the appropriate visibility region of the scope. There are two forms of accessible scopes other than the local scope at the point of creation of the binding.

The first form is a named scope. Depending on the semantics, the appropriate visibility region for the definition of visibility of the binding can be found by simply looking up the named scope, or by the more complex mechanism described in the section on nested qualified references.

The second form consists of a scope implicitly defined by its relationship to the scope causing the creation of the binding. PL/I implicit declarations are a good example of this: the binding is to be added to the global scope. In other words, the binding is to be added to the outermost scope enclosing the current scope. This result can be accomplished quite easily by defining upon entry to each scope a special name "#enclosing_scope#" that is bound to the entity denoting the enclosing scope. At any point in the program, the enclosing scope can be found by looking up the name "#enclosing_scope#". The global scope can be found by iterating this process, using the *ref_env* of each successive enclosing scope for successive lookups.

The name "#enclosing_scope#" is selected so that it cannot clash with any language- or describer-defined names. However, we want the different bindings of "#enclosing_scope#" within a program to clash with one another, so that the binding of "#enclosing_scope#" defined in a scope S will shadow the binding of "#enclosing_scope#" defined in the scope enclosing S. In a language that allows shadowing, the same visibility classes used for describer-defined bindings can be used for "#enclosing_scope#". If the language does not allow shadowing, additional visibility classes, along with appropriate clash table entries, will be needed so that bindings of "#enclosing_scope#" are shadowed properly.

```
function resolve_qualified_ref (
    N       : list of Name;
    current_vertex : IGVertex) : Binding;
    old_binding, current_binding, containing_binding : Binding;
    old_name, current_name : Name;
    containing_bindings : Set of Binding;
    is_nested : Boolean;

    old_name := head (N); N := tail (N);
    if pointer_type_decl then
        old_binding := pointer_type_lookup (old_name, current_vertex);
    else old_binding := standard_lookup (old_name, current_vertex);
    if old_binding = error_binding then
        error ("Illegal reference to '%s': not visible", name_to_ident(old_name))
        return (error_binding)
    while N ≠ null
        current_name = head (N); N := tail (N);
        if not (qualifiable_kind (old_binding.kind)) then
            error ("%s may not be qualified", old_binding.name)
            return (error_binding)

        -- find most recent redefinition of (old_binding, containing)
        containing_bindings := containing_lookup (old_binding.name, current_vertex)
        containing_binding := select b:Binding from containing_bindings
                such that b.entity.unique_id = old_binding.entity.unique_id
        is_nested := containing_binding ≠ null.
        if is_nested then
            current_binding :=
                    directly_visible_lookup (name_to_ident (current_name),
                                containing_binding.entity.ref_env)
        else current_binding := directly_visible_lookup (current_name,
                                old_binding.entity.ref_env)
        if current_binding = error_binding then
            error ("No %s declared local to %s", name_to_ident(current_name),
                    name_to_ident(old_name))
            return (error_binding)
        if old_binding.entity.kind = module_decl then
            if current_binding.exported = false then
                error ("Attempt to reference non-exported field %s of module %s qualified",
                        name_to_ident(current_name), name_to_ident(old_name))
                -- Continue processing qualified ref to find more errors, if any
        old_name := current_name; old_binding := current_binding
```

Figure 6.11: Function resolve_qualified_ref

```
function containing_lookup (
    n          : Name;
    v          : IGVertex
    pointer_type_decl : Boolean
    ) : Set of Binding;

    return (visible (n, v, containing))
```

Figure 6.12: Function containing_lookup

6.5.6. Visibility Information Independent from Source Program

Visibility information independent of any source program being processed can be represented by defining an inheritance graph or part of an inheritance graph with the visibility regions, inheritance edges, and binding definitions as desired. The visibility information resulting from a standard environment can easily be described in this manner. The first node of the inheritance graph corresponding to a user's program inherits the appropriate visibility from the last node of the subgraph corresponding to the standard environment.

6.6. Examples

This section describes how some common visibility construct features may be described using an inheritance graph.

6.6.1. Closed Scopes

A completely closed scope inherits nothing from the enclosing scope, so there is no inheritance edge from the enclosing scope to the closed scope. A partially closed scope inherits only certain bindings from the enclosing scope. In Euclid, where only pervasive bindings are inherited into closed scopes, the edge from the visibility region of the enclosing scope to the first visibility region of the closed scope is labeled with the function *restrict_pervasive_only* given in §6.2.7.

6.6.2. Named Inheritance

Named inheritance, as described in §2.6.4, allows the inheritance of visibility of bindings from a scope specified by name. Suppose we have a visibility construct "**inherit** M;" that specifies that visibility is inherited from a module with name "M", the first step is to find the providing scope M, and the vertex $v_{providing}$ corresponding to the appropriate visibility region local to M. This can be done by looking up "M" at the point of the **inherit** statement[34], and using the value of the *ref_env* attribute of M as the source of the inheritance edge for the **inherit**. Then, an inheritance edge is added from $v_{providing}$ to the vertex resulting from **inherit**. This edge is called a *dynamic edge*, because it does not depend on the static structure of the program, but rather on the visibility in some visibility region, which must be determined by (partially) evaluating the inheritance graph. The semantics of the **inherit** statement determine what visibility classes are inherited, and what restriction function is associated with the edge.

An import from a named module is an example of named inheritance. An example of an inheritance graph resulting from named inheritance is given in the next section.

[34] There is a chicken-and-egg problem apparent here, because it is necessary to resolve a reference before completing the construction of the inheritance graph. The solution to this problem will be discussed in §6.7.

6.6.3. Import and Export

The import and export constructs cause bindings to be inherited from one visibility region to another, in addition to any other inheritance. There are many forms of import/export constructs, as described in previous chapters. The first form is a form of named inheritance, and causes specific, named bindings to be inherited from the providing scope to the inheriting scope. Consider a Modula-2 import statement of the form

from M import A,B;

As described in the previous section, the providing scope *M* is found, and then the appropriate visibility region local to *M*. Finally, inheritance edges corresponding to the import statement added. Both ***referable*** and ***known*** edges are added, because the imported bindings be referenceable (thus ***referable***) in the importing scope, and they must not be redeclared in the importing scope (thus ***known***)[35].

The restriction function for the inheritance edges only returns **true** for bindings of the names listed in the import statement: "A" and "B". A ***directly_visible*** lookup of "A" and "B" must be performed at the visibility region being imported from, with appropriate error checks, to ensure that they are declared in the scope being imported from. The part of the inheritance graph corresponding to the above import statement is given in Figure 6.13.

The most common import statement is of the form

import A,B;

where "A" and "B" are imported from the scope enclosing the current scope. We must locate the proper visibility region in the enclosing scope for the source vertex of the import's inheritance edge. As in §6.5.5, a binding of "#enclosing_scope#" is defined, which has as one of its attributes (*ref _env*) the most recent visibility region of the enclosing scope. To find the correct visibility region to import from, we find the binding *b* bound to "#enclosing_scope#" and add an edge from *b.entity.ref _env* to the current visibility region. The inheritance graph corresponding to this form of import statement is given in Figure 6.14. Note that since the import is from the enclosing scope, there is the normal ***referable*** inheritance edge from the enclosing scope into the current scope in addition to the edge caused by the import.

The precise effect of an import statement on the inheritance graph depends on the language's definition of the effect of importing bindings. As discussed in sub-section "Imports and Exports" of §4.2.6, the importation of visibility of a binding may only make that binding visible (as in the ***referable*** visibility class), or it may be equivalent to creating a new declaration of the binding in the inheriting scope. The latter corresponds to the imported binding being ***known*** in the inheriting scope. Both cases can be handled by varying the inheritance from the providing scope. In the former case, only ***referable*** visibility is inherited. In the latter case, both ***referable*** and ***known*** visibility are inherited, as illustrated in Figure 6.14.

More generally, the designer of the inheritance graph description for a language must consider the possibilities of clashes between imported bindings, locally declared bindings, and ***referable*** bindings automatically inherited from the enclosing scope.

The export statement is handled similarly to the import statement, except that the providing visibility region is the current visibility region, and the inheriting visibility region is a visibility region in the inheriting scope.

---

[35] The Modula-2 reference manual [Wirth 1982] does not actually state that imported bindings may not be redeclared in the importing scope, but this seems to be the most reasonable semantics. This and other ambiguities in Modula-2 are discussed further in Chapter 7.

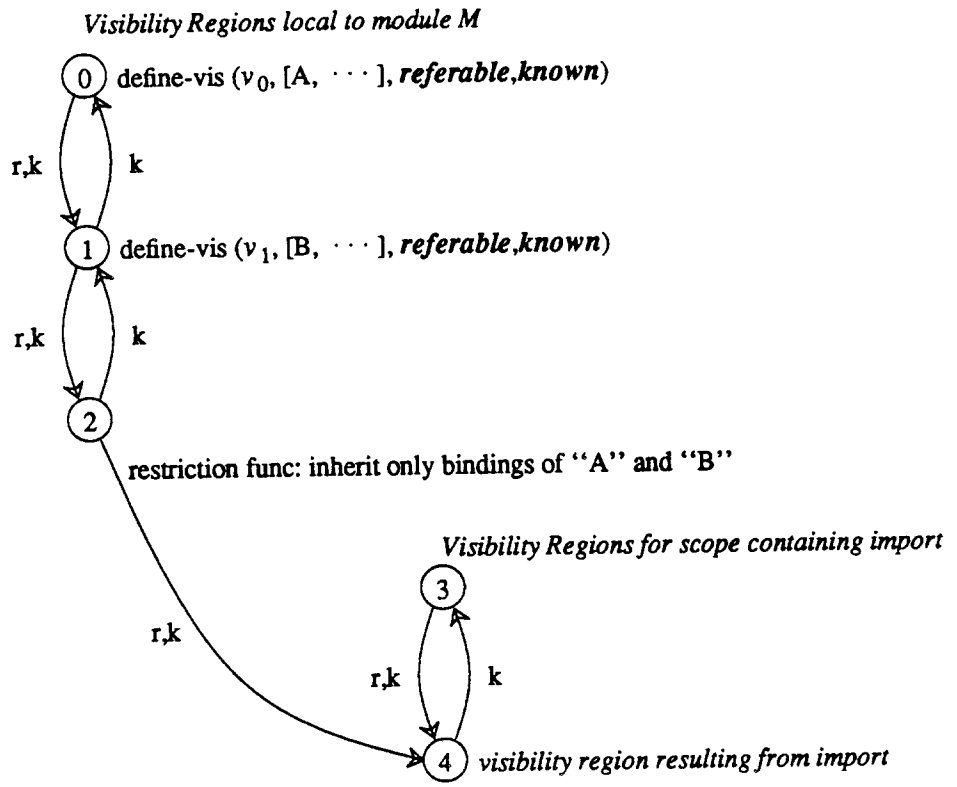


Figure 6.13: Import from Named Module

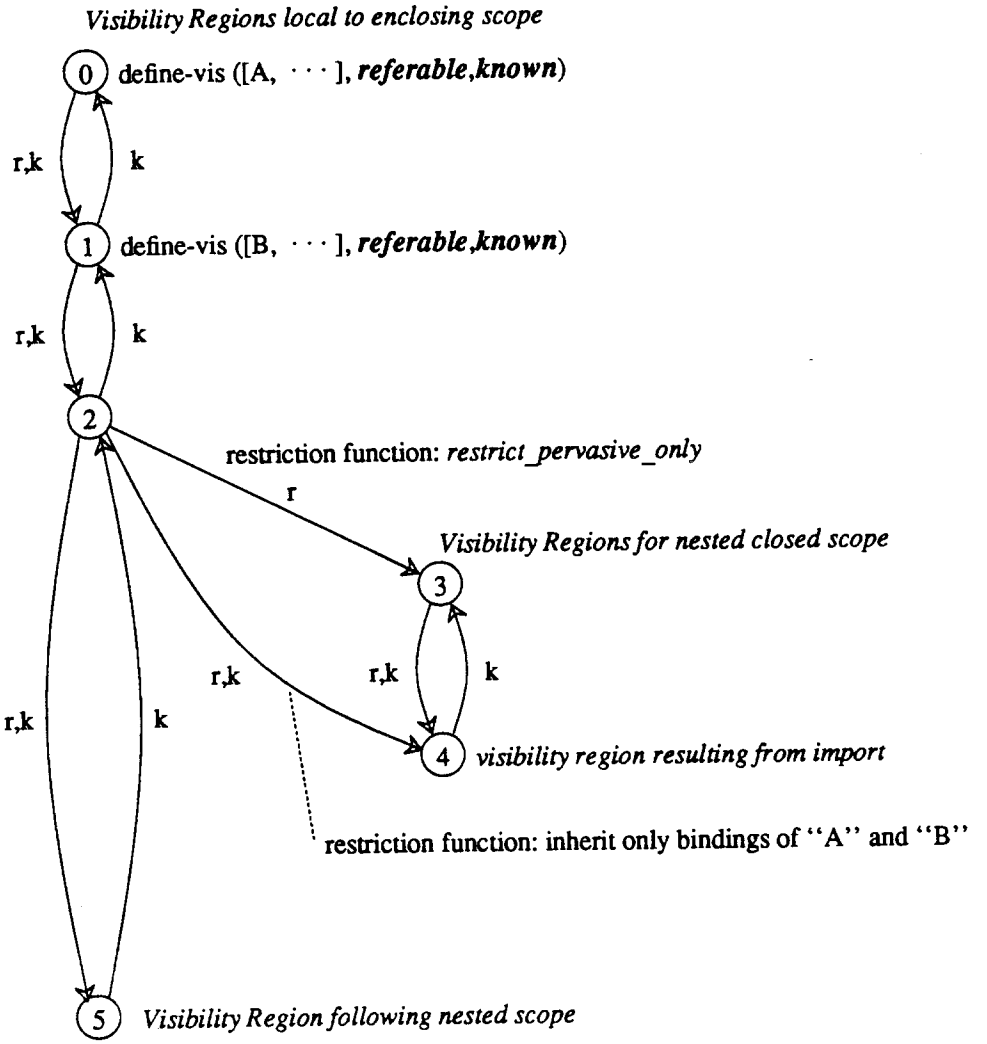


Figure 6.14: Import from Enclosing Scope

The other major form of import is the bulk import, as described in §2.6.8. This is discussed in the following section on opening of scopes.

6.6.4. Opening of Scopes

The Ada **use** is a very complex form of scope opening and is discussed separately in §6.6.9. This section describes how the Pascal **with** statement is represented in an inheritance graph. The Pascal statement

with R **begin** *statement-list* **end**

creates a new scope in which the fields of record *R* are directly visible and shadow any clashing bindings visible at the point of the **with** statement. In the terminology used thus far for Pascal, the bindings declared local to *R* must be ***referable*** and ***known*** in the scope created by the with statement.

We begin by creating a new visibility region $vr_{\textbf{with}}$ corresponding to the new scope, with a ***referable*** edge from the visibility region prior to the **with** statement to $vr_{\textbf{with}}$. We then find the binding b_R of R, which must have a *ref_env* attribute containing a pointer to the visibility region vr_{last} corresponding to its last field. ***referable*** and ***known*** edges are added from vr_{last} to $vr_{\textbf{with}}$, with the following restriction function on the ***referable*** edge:

```
function restrict_locals_only (
    b       : Binding) : Boolean;

    return (b.entity.declaring_record = b_R.entity);
```

The *declaring_record* attribute of each field of a record points to the entity of that record. This restriction is necessary because bindings defined outside the record are ***referable*** inside the record, and thus would be inherited into $vr_{\textbf{with}}$ in the absence of the restriction. No restriction function is needed on the ***known*** edge, because only the bindings defined local to R are ***known*** within R.

6.6.5. Wolf's Provide and Request Operations

Wolf's provide and request operations (described in §2.6.7) provide more precise control over visibility than any of the other visibility control features described. A binding b is visible in a visibility region vr only if b provides access to vr and vr requests access from b. Normally, a scope requests access to a binding, which is equivalent to a request from all visibility regions in the scope.

An inheritance graph edge from vr_1 to vr_2 corresponds to a request by vr_2 for access to everything visible at vr_1, and provision of access by everything visible at vr_1 to vr_2. Explicit requisition and provision of access can be modeled in the Inheritance Graph Model as follows: Each visibility region or scope distinguishable by a **provide** statement has a distinguished vertex $v_{provided}$. An edge is added from all visibility regions providing access (with inheritance restricted to the bindings actually provided) to $v_{provided}$. This edge represents a provide operation. An edge from $v_{provided}$ to the first actual visibility region of the scope, with inheritance restricted to the bindings actually requested, represents a request operation. Thus, binding b will only be visible in scope S only if S requests access to b and b provides access to S.

6.6.6. Separate Compilation

A separate inheritance graph can be created for each separately compiled program unit, and stored for later use. If the inheritance graph ig_1 for a previously processed unit U_1 is needed for processing the current program unit U_2, the inheritance graph ig_2 for U_2 can be created by retrieving ig_1 and joining it to the newly created inheritance graph with appropriate inheritance edges, and recomputing the visible sets at all of the new vertices.

In Modula-2, the inheritance graph for a definition module is needed for processing the corresponding implementation module. Edges from the last visibility region of the definition module to the first visibility region of the implementation module are added so that declarations in the definition module have the same status in the implementation module as declarations local to the implementation module. The exact visibility classes used will be given in the complete Modula-2 example.

6.6.7. Overloading

The meaning of clashing bindings, and thus the clash function, has been defined such that overloaded bindings do not clash. Thus, overloaded bindings do not shadow one another, and an overloaded binding b of name "N" will be visible (with the appropriate visibility classes) in all visibility regions where a reference "N" can possibly refer to b. When a reference "N" occurs

in visibility region *vr*, all bindings of "N" whose visibility reaches *vr* will be found by a call to *visible*. It is then up to the overloading resolution algorithm to select the correct binding. Overloading resolution is a type-resolution issue, not a visibility issue.

6.6.8. PL/I and COBOL Structures

Resolution of qualified references for structures in PL/I and COBOL can be defined using the Inheritance Graph Model. A qualified reference $N_1.N_2.\cdots.N_{n-1}.N_n$ must be resolved from right to left because of the possibility of partially qualified references. A special visibility class ***qualified_visible*** is used, where a binding is ***qualified_visible*** in any visibility region in which it can be used as the last (perhaps only) component of a qualified reference. ***qualified_visible*** visibility of a structure field's binding is inherited from the visibility region containing the field declaration to all more deeply nested structures, and to the block containing the structure declaration.

After finding the binding b_n of "N_n" corresponding to the last component of a qualified reference $N_1.N_2.\cdots.N_{n-1}.N_n$ (using ***qualified_visible*** visibility), the reference to the preceding component is resolved in the visibility region where b_n was defined. In that visibility region, only bindings of enclosing structures are ***qualified_visible***, so a lookup of ***qualified_visible*** bindings will only find bindings that can precede a reference to *b* in a qualified reference.

This process must be repeated, for each name in a qualified reference from *n* to 1. The result of step *i* (counting down from *n*) is a set of bindings *bs* that the qualified reference "$N_i.\cdots.N_n$" can be a reference to. If *bs* becomes empty at any stage, the reference is erroneous. If *bs* contains more than one binding, then the reference is valid if it is an exact reference to one of the bindings.

6.6.9. Ada Use Clause

In all previous examples, *define_vis* calls were static, independent of any other information in an inheritance graph. In a straightforward representation of Ada **use** clauses in an inheritance graph, this is no longer so. Rather, the presence of some *define_vis* calls will depend on the visibility at some vertex. Such a *define_vis* call is called a *dynamic definition.*

The semantics of Ada **use** clauses is given in §5.2. Let us say that a binding in Ada is ***directly_visible*** in all visibility regions local to the scope containing the binding's declaration and after the point of the declaration. A binding is ***potentially_visible*** where ever it is "potentially visible", as also described in §5.2.

The effect of a **use** clause "**use** P1;" in terms of an inheritance graph can be described as follows: any binding that is ***directly_visible*** at the inheritance graph vertex v_{last} resulting from the last non-private declaration in *P1* becomes ***potentially_visible*** at the vertex v_{use} resulting from the **use** clause, and is inherited with ***potentially_visible*** visibility into all scopes nested within the scope containing the **use** clause. Because a binding is not normally ***potentially_visible*** where it is ***directly_visible***, it is not possible to simply inherit ***potentially_visible*** visibility from v_{last}. Rather, it is necessary to add a call

$$define\_vis\ (v_{use}, b, potentially\_visible)$$

for each binding *b* such that $b \in visible(b.name, v_{last}, directly\_visible)$. Precisely which bindings have ***potentially_visible*** visibility defined at v_{use} using *define_vis* calls depends on which bindings are inherited with ***directly_visible*** visibility at v_{last}, so some information about inheritance of visibility in the graph must be available before the ***potentially_visible*** *define_vis* calls can be added. This is similar to the problem of adding an edge which depends on a reference, as for named inheritance (§6.6.2).

Figure 6.15 shows a fragment of an inheritance graph resulting from a **use** statement that imports the contents of two packages:

```
use P1, P2;
```

where *P1* contains a binding of "A," and *P2* contains a binding of "B."

The clash table for Ada is[36]:

Ada Clash Table		Visibility Class for b_2		
		directly_visible	***referable***	***potentially_visible***
Vis Class for b_1	***directly_visible***	error("description error")	**shadow** b_2	–
	referable	**shadow** b_1	–[37]	–
	potentially_visible	–	–	–

The ***directly_visible*** visibility class is used to keep track of local declarations of a scope. ***directly_visible*** arcs flow in text order direction, but not from outer to inner scopes, as illustrated by edge (v_0,v_1) in Figure 6.15. ***potentially_visible*** arcs flow in the direction of text order, and also from outer to inner scopes. Inheritance of ***potentially_visible*** bindings is never halted by restriction functions or by **shadow** entries in Ada's clash table, so any ***potentially_visible*** binding is ***potentially_visible*** anywhere following its initial definition, including in nested scopes. This corresponds to the Ada rules for **use** statements that state that all non-private bindings of all packages appearing in **use** statements affecting a scope are potentially visible in that scope.

Given these basic visibility classes for Ada, it is possible to define a lookup function. Any ***referable*** bindings take precedence over any ***potentially_visible*** bindings, and if any two ***potentially_visible*** bindings b_1 and b_2 clash at the point of reference, then any ***potentially_visible*** binding with the same name as b_1[38] is not visible, and should not be returned by the lookup function. *direct_lookup*, for regular references in Ada, follows.

```
function direct_lookup (
    n       : Name;
    v       : IGVertex) : Set of Binding;

    referable_bindings,
    potentially_visible_bindings : Set of Binding;

    referable_bindings := visible (n, v, referable);
    potentially_visible_bindings := visible (n, v, potentially_visible);
    return (referable_bindings ∪
        { b | b ∈ potentially_visible_bindings
                ∧ (∀ b' ∈ referable_bindings such that b ≠ b', clash (b ,b')=false)
                ∧ (∀ b' ∈ potentially_visible_bindings such that b ≠ b', clash (b ,b')=false)
        })
```

direct_lookup is somewhat complicated, and has time complexity of $O(m \times n + n^2)$, where m is the size of *referable_bindings* and n is the size of *potentially_visible_bindings*. However,

[36] Note that I haven't attempted to do a complete description of the visibility control rules of Ada, so it is possible that other visibility classes are needed to handle other visibility control constructs.

[37] This is an error, but it is ignored because the error will also be detected by a ***directly_visible/directly_visible*** clash, and there is no reason to produce extra error messages.

[38] equivalently b_2, since only bindings with the same name clash, by definition of clash.

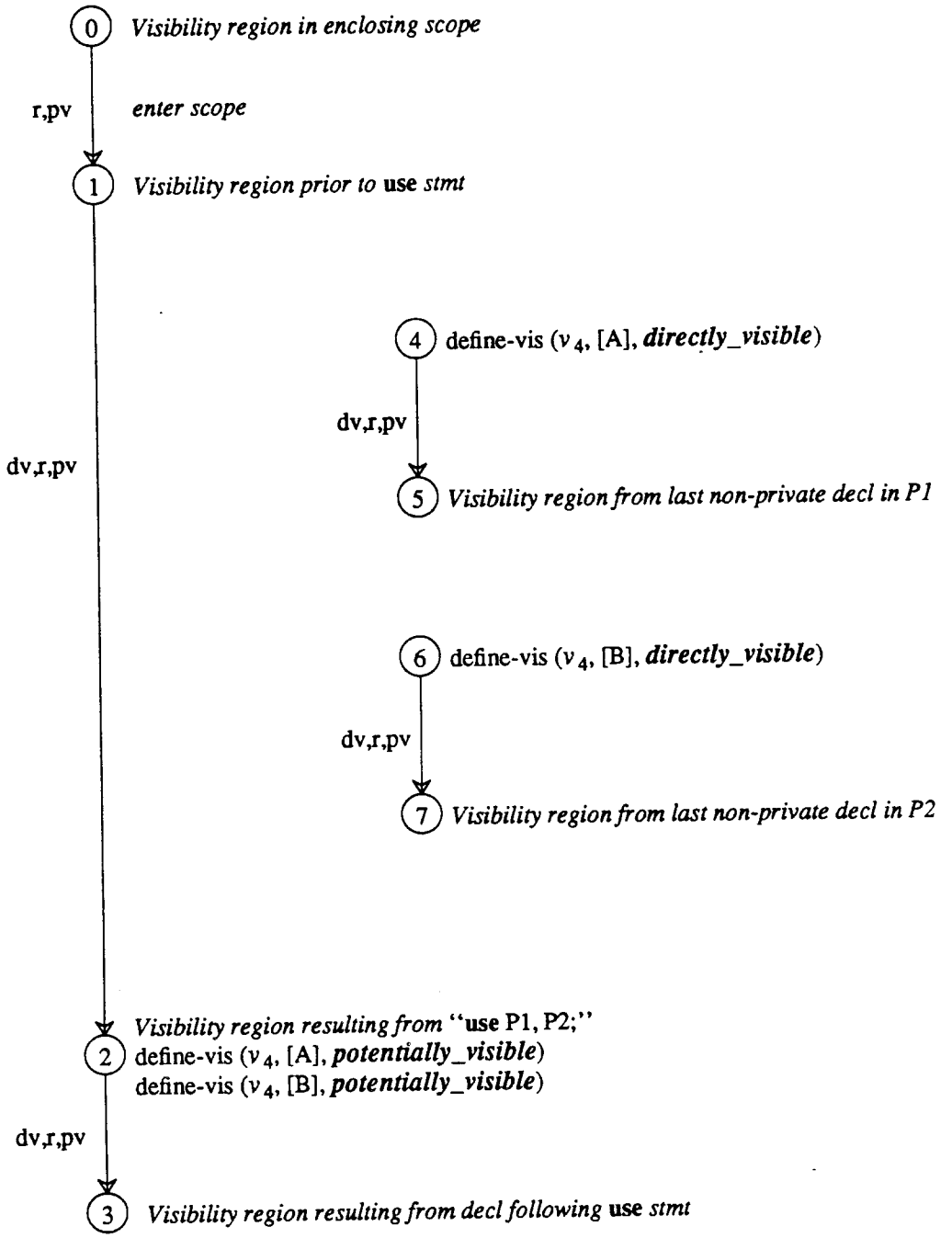


Figure 6.15: Inheritance Graph for Ada use Statement

m and n are both likely to very small (less than 2) in most cases.

6.6.10. Flavors

One requirement for a model of visibility control is the ability to select among the bindings inherited by a contour based on some ordering, as in multiple inheritance in Flavors (Multiple Inheritance sub-section of §4.2.6). This can be done by ordering the visibility classes and using the clash resolution process to shadow all bindings but the one with the lowest numbered visibility class.

The expression

(defflavor S_1 ( ) (S_2 S_3))

defines a flavor S_1 which inherits visibility from S_2 and S_3. Suppose reference "A" occurs in the body of S_1. If "A" is bound in S_1, then that binding overrides, or shadows, any binding of "A" in S_2 or S_3. If "A" is not bound in S_1, then we look for a binding of "A" visible in S_2, and finally in S_3 if no binding of "A" is visible in S_2.

This semantics of shadowing can be represtented in the Inheritance Graph Model using two visibility classes, ***referable*** and ***inherited***. A binding b is ***referable*** in the scope in which it is declared, and in all scopes in which b becomes visible (in the ordinary sense of "visible") by inheritance. A binding b is ***inherited*** in the visibility region which inherits visibility of b, whether b actually becomes ***referable*** in the inheriting scope or is shadowed by another binding of the same name. ***referable*** edges are labeled with "r." There are no ***inherited*** edges. The need for any other visibility classes is ignored for the purposes of this example.

Part of the inheritance graph corresponding to the above call to *defflavor* is illustrated in Figure 6.16, where S_2 contains a binding of "A" to an *Integer* variable, and S_3 contains a binding of "A" to a *Boolean* variable. According to the ordering rules for inheritance in Flavors, only the [*A,Int*] binding should become ***referable*** in S_1.

Calls to *define_vis* are added in three distinct stages, and the calls to *define_vis* are numbered in the graph to show these three stages:

(1) First, the definitions of ***referable*** visibility corresponding to the creation of the bindings of "A" in S_2 and S_3 are associated with v_4 and v_6, respectively.

(2) Second, ***inherited*** visibility of [*A,Int*] is defined at v_1, reflecting the inheritance from v_5 in S_2. ***inherited*** visibility of [*A,Bool*] is defined at v_2, reflecting the inheritance from v_7 in S_3.

(3) For each call "*define_vis* (*b*, *v*, ***inherited***)", if $b \in$ *visible* (*b.name*, *v*, ***inherited***) and $b \notin$ *visible* (*b.name*, *v*, ***referable***), then add a call "*define_vis* (*b*, *v*, ***referable***)".

At this point, [*A,Int*] is defined to be ***referable*** at v_1, and [*A,Bool*] is defined to be ***referable*** at v_2. Visibility of both of these bindings is propagated via the ***referable*** edges. The clash resolution process specifies that ***referable*** bindings shadow ***inherited*** bindings:

Flavors Clash Table		Visibility Class for b_2	
		referable	***inherited***
Vis Class for b_1	***referable***	error ("clashing declarations")	shadow b_2
	inherited	shadow b_1	error ("description error")

<[*A,Int*],***referable***> will be inherited at v_2, causing <[*A,Bool*],***inherited***> to be shadowed at v_2. Because the definition of ***referable*** visibility at v_2 depends on <[*A,Bool*],***inherited***> to be visible at v_2 *after* clash resolution, the *define_vis* call that defines ***referable*** visibility of [*A,Bool*] at v_2 is no longer valid, and must be removed.

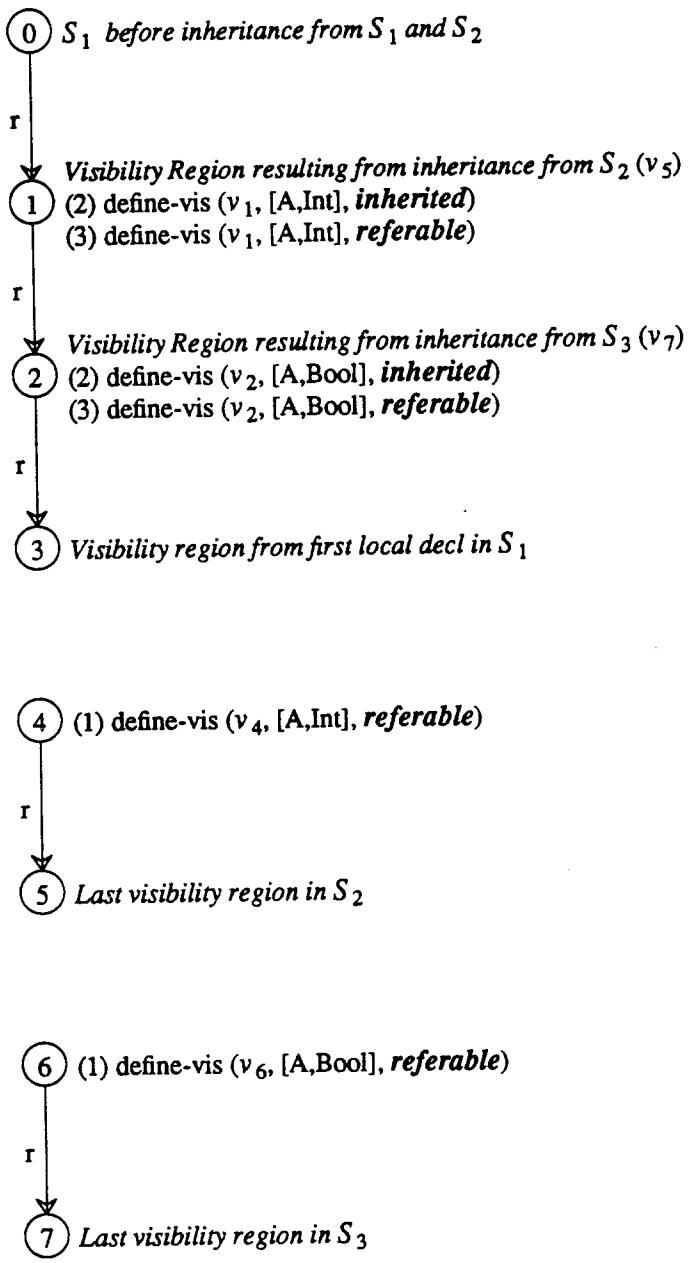


Figure 6.16: Inheritance Graph for Flavors Example

Note that ***referable*** edges only flow downward in the graph, so ***inherited*** visibility of [*A,Int*] at v_1 is not affected by the definition of ***referable*** visibility at v_2. Thus bindings inherited from a flavor named in the inheritance list in a call to *defflavor* override bindings of the same name inherited from a flavor later in the inheritance list.

After the *define_vis* call that defines ***referable*** visibility of [*A,Bool*] at v_2 is removed, the propagation of visibility and the clash resolution process must be repeated. The total number of iterations required is proportional to the number of levels of inheritance, where each flavor that

inherits visibility from other flavors represents a single level of inheritance, although a clever algorithm for evaluating the inheritance graph might be able to avoid re-evaluating the inheritance graph each time, particularly because these inheritance graphs will have no cycles.

Shadowing Based on Priorities

A simpler description of shadowing based on ordering can achieved by allowing numbered priorities on edges. The priority of an edge decreases with increasing numerical value. Priorities are a possible extension to the basic Inheritance Graph Model, although all visibility features encountered can be described using the primitives of the basic Inheritance Graph Model. Priorities just make the description of ordering simpler.

If clashing bindings (of any visibility class) are inherited via more than one incoming edge of a vertex, then the binding(s) inherited via the edge with highest (lowest numbered) priority actually becomes visible at the vertex, and all other clashing bindings are shadowed. The inheritance graph in Figure 6.17 illustrates how the importation of the bindings of scopes S_2 and S_3 into S_1 could be represented. Each edge is labeled not only with a visibility class, but with a priority.

In the figure, a "main" stream of inheritance edges has priority 1, and inheritance edges from the scopes being inherited from have priority 2. At v_1, a binding of "A" declared local to S_1 would have priority over a binding inherited from S_2. There is no binding of "A" declared in S_1, so [*A,Int*] declared in S_2 becomes visible (***referable***) at v_1. <[*A,Int*],***referable***> is inherited via the priority 1 edge ($v1$, v_2) to v_2 and thus has priority over any clashing binding inherited via the priority 2 edge (v_7, v_2) from S_3. The **prioritize** primitive is used to select among bindings based on priorities, as illustrated in the following clash table:

Flavors Clash Table (w/Priorities)		Visibility Class for b_2	
		referable	***known***
Vis Class for b_1	***referable***	**prioritize**	**shadow** b_1
	known	**shadow** b_2	error ("clashing declarations")

The ***known*** visibility class is needed in this version of the inheritance graph for Flavors to detect duplicate declarations of a name local to a scope, because it is not an error for two bindings to be ***referable*** at the same vertex in this version.

The power of edge priorities is limited: it is certainly possible to invent ordering schemes that cannot be defined using only edge priorities. However, such ordering schemes are likely to be so complex as to have questionable value in a programming language.

6.7. Building the Inheritance Graph

Previous sections have defined the inheritance graph, and the meaning of resolving references using the inheritance graph. The correspondence between various language constructs and structures in an inheritance graph has also been illustrated. However, actually building the inheritance graph corresponding to a particular program may be more difficult than designing the subgraph-schema for a particular visibility construct, because the definition of the inheritance graph for many languages will be circular. Constructs such as import and export that create dynamic edges in the inheritance graph are the source of the problem.

There are two classes of operations on an inheritance graph: *construction* operations, which build and modify the inheritance graph (*create_vertex, create_edge, define_vis*, etc.), and *reference* operations, which examine the inheritance graph (*visible*). In a language with simple visibility control rules, construction operations do not depend on reference operations. Visibility

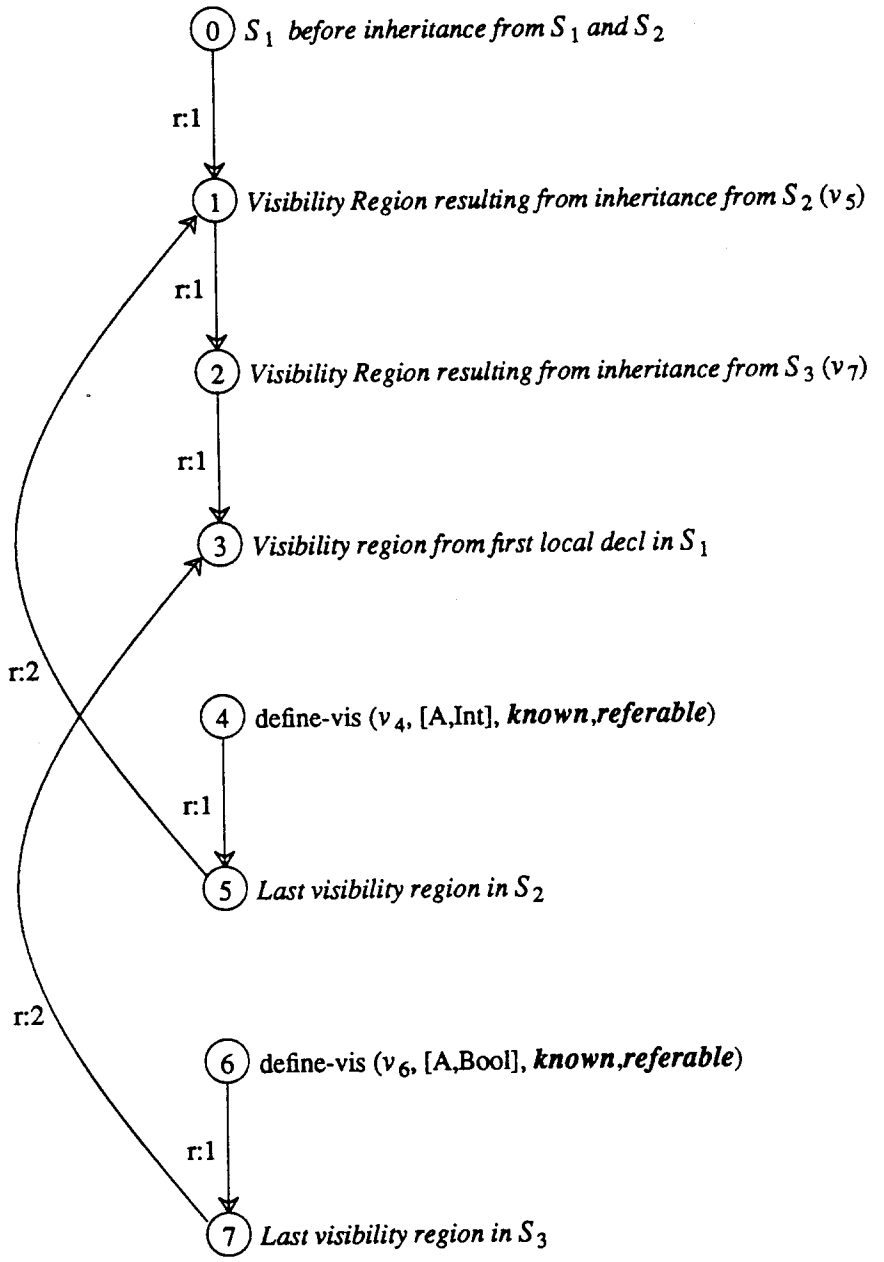


Figure 6.17: Inheritance Graph with Prioritized Inheritance in Flavors

constructs that create dynamic edges usually *do* depend on reference operations, particularly those operations that create an edge from or to a named scope (named import and export, and named inheritance), or those that create dynamic definitions. Sufficient information to compute the result of calls on *visible* must be available for the inheritance graph. If this is done by computing the visible set at each vertex in the inheritance graph, the inheritance graph must be built and each visible set computed before the necessary lookups can be done to find the appropriate vertices for the dynamic edges. Visibility must then be recomputed, because other references

may depend on inheritance of visibility due to the new edges. We call a description that results in such inheritance graphs a *cyclical inheritance graph description.*

In the general case, the process of building the inheritance graph is iterative, because each new edge may allow new references to be resolved, adding yet more edges. A new edge can also cause a previously resolved reference to be invalidated, either by making a different binding visible at the point of reference, or by causing no binding to be visible. Therefore, the iteration must continue until there is no change in the inheritance graph or in references that can affect the inheritance graph. Figure 6.18 contains an outline of an algorithm to construct an inheritance graph.

```
procedure build_inheritance_graph
    -- reference operations that construction operations depend on
    d_reference := {r:reference operation | inheritance graph depends on r}
    -- all other reference operations
    o_reference := {r:reference operation in inheritance graph | r ∉ d_reference}

    -- construction operations that depend on a reference operation
    d_construction := {c:construction operation | c depends on d_reference ∪ o_reference}
    -- all other construction operations
    o_construction := {c:construction operation in inheritance graph | c ∉ d_construction}

    all_ops = d_reference ∪ o_reference ∪ d_construction ∪ o_construction

    -- mapping from a construction op to the reference operations that it depends on
    constr_op_refs ( d_construction_i ) = { r ∈ d_reference | d_construction_i depends on r}

    for op ∈ all_ops
    |   new_value(op) := null
    for c ∈ o_construction
    |   new_value(c),errors(c) = execute(c)
    changed := true
    while changed = true
    |   changed := false
    |   evaluate inheritance graph
    |   foreach c ∈ d_construction
    |   |   foreach r ∈ constr_op_refs(c)
    |   |   |   new_value(r),errors(r) := execute(r)
    |   |   |   if new_value(r) ≠ old_value(r) then changed := true
    |   |   new_value(c),errors(c) := execute(c)
    |   |   if new_value(c) ≠ old_value(c) then changed := true
    |   old_value := new_value
    foreach r ∈ o_reference
    |   new_value(r),errors(r) := execute(r)
    all_errors := ∪ (op ∈ all_ops) errors(op)
```

Figure 6.18: Algorithm build_inheritance_graph

Each operation produces a value and a (possibly empty) set of error messages. *errors* is a tuple indexed by the operation, containing the set of errors resulting from each operation. Each time an operation is re-evaluated, the new errors replace the old errors, so only errors produced the last time an operation is executed are *final errors*, as opposed to *intermediate errors*, which result from intermediate executions of an operation and can be discarded when the operation is executed again. *all_errors* collects all of the error messages together.

new_value and *old_value* are also tuples indexed by operations, and are used to detect changes in the result values of operations. The assignment of *new_value* to *old_value* at the bottom of the **while** loop is a value copy, not a pointer assignment. A more efficient implementation using pointers is obvious. At the end of *build_inheritance_graph*, *new_value* contains the correct values (perhaps special error values) of all operations.

A possible optimization would be to evaluate only operations in *d_construction* whose references have just been successfully evaluated for the first time, re-evaluating them again after all operations in *d_construction* have been evaluated, to check for changes. This is likely to be useful in languages and programs where operations in *d_construction* depend heavily on one another (directly or indirectly), resulting in many iterations of the **while** loop. This occurs only in programs with deep nesting and multi-level imports or exports, or multi-step named inheritance.

However, most programs have no more than one level of edges resulting from named inheritance, resulting in at most three iterations. The initial iteration is needed so that references for dynamic edges can be resolved, the second iteration propagates the effects of the dynamic edges, and the third iteration is needed to check that the evaluation has stabilized. One more iteration is needed for each additional level of indirection in references for dynamic edges. For languages in which a newly resolved reference or new edge cannot change or invalidate a previously resolved reference, the last iteration can be omitted.

If the inheritance graph description includes dynamic definitions, as for Flavors, the number of iterations required will be proportional to the number of levels of inheritance. There may be several levels of inheritance, due to the style in which this inheritance mechanism is used.

6.8. Well-Definedness of an Inheritance Graph

Very few restrictions have thus far been placed on a description of an inheritance graph. It is certainly possible to create an ambiguous inheritance graph. This section describes how ambiguous inheritance graphs may arise, restrictions on inheritance graphs that preclude various kinds of ambiguities, and different classes of inheritance graphs based on these restrictions.

The first example is very simple; it is based on an actual example that occurred during the early stages of design of the inheritance graph formalism, while experimenting with ways to describe the meaning of clashing-binding errors:

6.8.1. Shadowing and Ambiguity

Assume we have the following clash table and the inheritance graph in Figure 6.19.

Clash Table		Visibility Type for b_2 ***referable***
Vis Type for b_1	***referable***	**shadow** b_1, **shadow** b_2

For the example, *clash* (b_1,b_2) = **true**. There are two valid assignments (two fixed points) to the visible sets in the inheritance graph, depending on the order of computation of the visible

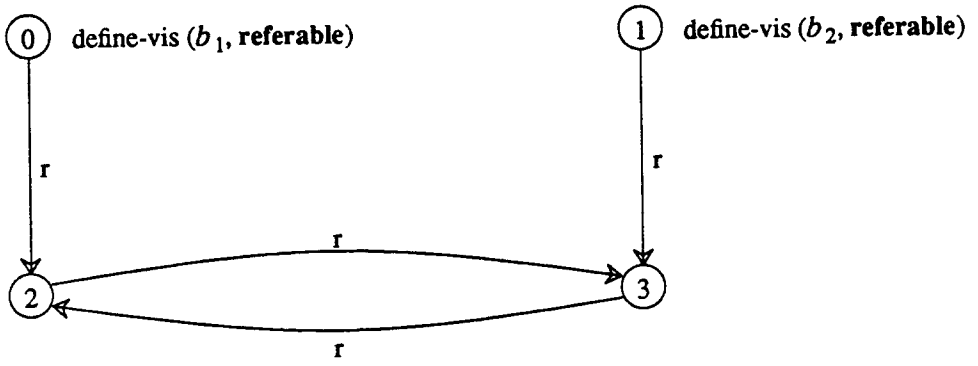


Figure 6.19: Ambiguous Inheritance Graph

sets of the graph. In both assignments, <b_1,*referable*> is visible at v_0, and <b_2,*referable*> is visible at v_1. In one valid assignment, <b_1,*referable*> is visible at v_2, while nothing is visible at v_3. In the other valid assignment, <b_2,*referable*> is visible at v_3, while nothing is visible at v_2. If the net visibility of v_2 is computed before the visibility sets at v_3, then b_1 will be inherited by v_3, resulting in a clash between b_1 and b_2. The clash table states then that both bindings are shadowed there, resulting in no further inheritance, and thus the first valid assignment. The computation of the second valid assignment is symmetric.

This ambiguity results because of the different possible orders of application of clash resolution, resulting in shadowing actions in different places.

Definition 6.1. Ordered Inheritance Graph

An inheritance graph is defined to be *ordered* if shadowing is restricted to use as an action resulting from clashes between bindings with different visibility classes, and visibility classes are strictly ordered. vc_i can only shadow vc_j, for $1 \le i < j \le n$, where n is the number of visibility classes used in the inheritance graph description. In addition, any dynamicaly created *define_vis* calls defining visibility of bindings with visibility class vc_j can only depend on bindings with visibility class vc_i, $i<j$.

If the inheritance graph is ordered, then the kind of ambiguity illustrated in Figure 6.19 cannot occur. An ordered inheritance graph IG can be split into separate graphs $IG_1, \cdots, IG_n$. Each IG_i has the same vertices as IG, but has only edges of visibility class vc_i.

Because of the ordering restriction on shadowing, shadowing in IG_j is determined solely by IG_i, $i<j$. That is, whether a particular binding/visibility-class pair <b, vc_i> whose visibility reaches a vertex v in IG_j is shadowed at v depends only on the binding/visibility-class pairs visible at v in IG_i, $i<j$. This condition not only makes the ambiguity illustrated in Figure 6.19 impossible, but it makes possible the definition of sufficient conditions for well-definedness of each IG_j. These sufficient conditions can be easily tested by the inheritance graph designer. The sufficient conditions arise from data flow analysis techniques, and will be discussed in Chapter 8.

6.8.2. Cyclical Inheritance Graph Descriptions

Let us assume that we have an inheritance graph description that results in well-defined inheritance graphs, with the exception that some inheritance graph construction operations may depend on reference operations, as described in §6.7 and §6.6.9. An ordered inheritance graph description that in addition satisfies the sufficient conditions for well-definedness mentioned at

the end of the previous section satisfies this condition. Our discussion in this section is restricted to ordered inheritance graphs. An inheritance graph description may be well-defined even though it is not ordered, although determining well-definedness for non-ordered inheritance graph descriptions is in general very difficult. Most of the results of this section can be extended to non-ordered, but well-defined inheritance graphs.

Oscillating Inheritance Graph Edges

Even if an inheritance graph is ordered, and each partition of the inheritance graph is well-defined, the meaning of the inheritance graph can be ambiguous if edges of the graph depend on references,[39] requiring iterative evaluation of the inheritance graph. This can occur if the edge-adding operation *e_op* depends on a reference *r_op* that returns a binding/visibility-class pair $<b, vc_i>$ used to determine where to add the edge, which in turn causes $<b, vc_i>$ to be shadowed at the vertex where *r_op* is executed, invalidating the edge. This is called an *edge↔ref dependency* This situation will oscillate with each iteration of the evaluation of the inheritance graph, resulting in a non-terminating iteration.

Figure 6.20 contains an inheritance graph that illustrates this situation. Attached to v_1 is the definition of binding $b_{N,1}$ with visibility class vc_i, where $b_{N,1}$ is used to denote the first binding of "N". Attached to v_5 is the definition of binding $b_{N,2}$ with visibility class vc_j, $i<j$. The meaning of the graph is as follows: *e_op* makes a call *visible*(v_3, *ident_to_name*("N"), vc_i), performing a lookup of "N" at v_3. There is a path p_1 from v_1 to v_3 along which $b_{N,1}$ is inherited, and all such paths pass through v_2. $b_{N,1}$ is found by the call to *visible*, and *e_op* adds a vc_j edge from v_5 to v_4, which is a vertex accessible via an attribute of $b_{N,1}$. There is a path p_4 from v_4 to vertex v_2 composed of vc_j edges that do not restrict inheritance of $b_{N,2}$.

For the example, the clash table specifies that a binding with visibility class vc_j shadows a clashing binding with visibility class vc_i. On the next iteration of evaluation of the inheritance graph, $b_{N,2}$ will have visibility class vc_j at v_2, and will shadow $b_{N,1}$ at v_2, resulting in $b_{N,1}$ no longer being visible at v_3. Thus, the reference *r_op* that found $b_{N,1}$ at v_3 will be unsuccessful, and the edge (v_5,v_4) created by *e_op* is no longer valid and must be removed.

Since no valid assignment to the inheritance graph has yet been found, the iteration of evaluation of the graph must continue, but the graph will oscillate: if the edge (v_5,v_4) exists, then the reference used to create the edge will be invalid after the effect of the edge on the inheritance is propagated. If the edge is removed, the reference is made valid, causing the edge to be added again on the next iteration.

The following examples illustrate how an edge↔ref dependency can occur in real programs. The example in Figure 6.21 contains two modules, P and Q: Modula-2 semantics are assumed: an imported binding is equivalent to a locally declared binding, and will result in a multiple declaration error if a clashing binding is imported or locally declared[40]. We will assume use of only two visibility classes, ***known*** and ***referable***, with the same meanings they have in Pascal.

Module P, denoted by P_1, contains a local declaration of "P" as an integer variable, denoted by P_2. Q contains two import statements. The first imports P_1. The second imports P_2 from within P_1. However, this results in two ***known*** bindings of "P" being inherited into Q. Since a ***known*** shadows a ***referable*** binding, the ***known*** binding of P_1 shadows the ***referable*** binding of P_2, and the ***known*** binding of P_2 shadows the ***referable*** binding of P_1. Because

[39] *d_construction* in Figure 6.18 contains operations that add edges to the inheritance graph.

[40] Modula-2 does not actually define this [Wirth 1982]. However, this seems to be the most reasonable interpretation of the meaning of import and export in Modula-2, and is the interpretation used by (at least some) implementations. The issue of poorly defined visibility control rules in Modula-2 is discussed in more detail in §7.1.

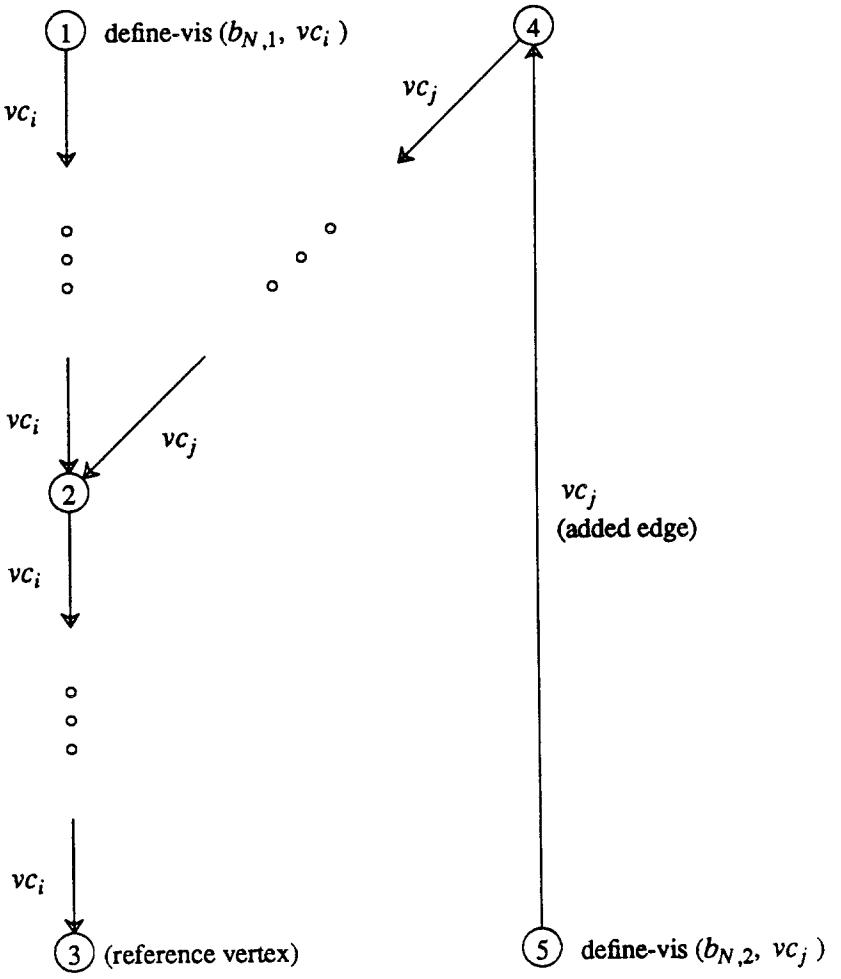


Figure 6.20: Oscillating Edge in an Inheritance Graph

```
module P      -- P1
      P : integer;      -- P2
end P;

module Q
      import P;
      from P import P;
end Q;
```

Figure 6.21: Simple Import Example (Oscillating Edge)

P_1 is no longer ***referable*** in Q, the second import statement cannot find the P to import from, so P_2 is no longer imported, removing the clash. Once the clash is gone, P_1 is visible again at the second import, and the cycle is established: the edge corresponding to the second import depends on itself, by causing the reference to P_1 to be be invalidated.

The example in Figure 6.22 is more complex, and illustrates a different cause of edge-oscillation. Only the relevant visibility regions in the example are labeled. The state after each iteration of the evaluation is illustrated in Table 6.1. The bindings visible with each visibility class at visibility regions vr_5, vr_6 and, vr_7 are given for each step. The table shows the bindings visible both before and after clash resolution are considered, in order to help the reader understand the transition from one iteration step to the next. The dynamic edges in the graph after each iteration are shown as

<(from-visibility region, to-visibility region), binding imported by edge>.

The visibility rules assumed in this example are like those of Pascal, with modules (closed scopes), user-defined pervasive bindings, and import statements added. In the example, a

```
pervasive module N   -- N1
       module M      -- M1
              N : integer;     -- N2
              vr0
       end M
       vr1
end N;
vr2
pervasive module M   -- M2
       module N      -- N3
       end M
       vr3
end M;
vr4

module S
       vr5
       from N import M;
       vr6
       from M import N;
       vr7
end S;
```

Figure 6.22: Import Example (Oscillating Edge)

	known	*referable*			non-local edges	
Step	vr_5, vr_6, vr_7	vr_5	vr_6	vr_7	from N import M	from M import N
1	–	N_1, M_2	N_1, M_2	N_1, M_2		
clash-resolved	–	N_1, M_2	N_1, M_2	N_1, M_2	<(vr_1, vr_6), M_1>	no "N" in M_2
2	M_1	N_1, M_2	N_1, M_1, M_2	N_1, M_1, M_2		
clash-resolved	M_1	N_1	N_1, M_1	N_1, M_1	<(vr_1, vr_6), M_1>	<(vr_0, vr_7), N_2>
3	M_1, N_2	N_1, M_2	N_1, M_1, M_2	N_1, N_2, M_1, M_2		
clash-resolved	M_1, N_2	–	M_1	N_2, M_1	no "N" visible	<(vr_0, vr_7), N_2>
4	N_2	N_1, M_2	N_1, M_2	N_1, N_2, M_2		
clash-resolved	N_2	M_2	M_2	N_2, M_2	no "N" visible	no "N" in M_2
5	–	N_1, M_2	N_1, M_2	N_1, M_2		
clash-resolved	–	N_1, M_2	N_1, M_2	N_1, M_2	<(vr_1, vr_6), M_1>	no "N" in M_2

Table 6.1: Evaluation States for Module R

pervasive binding is inherited into all nested scopes with ***referable*** visibility, but not ***known*** visibility. An import statement makes the imported binding ***referable*** and ***known*** in the importing scope. A ***known*** binding shadows a clashing ***referable*** binding. The actions taken in each iteration of the evaluation are as follows:

Step 1 N_1 and M_2 are ***referable*** in S, and the first import statement causes an edge to be added from within the body of N_1 (visibility region vr_1) to the visibility region vr_6 following the import, importing M_1. No binding of "N" is visible in M_2, so the second import fails due to an unresolved reference, and can't add an edge.

Step 2 M_1 is ***known*** throughout S, and ***referable*** in vr_6 and vr_7 because of the edge added in the previous step. $<M_1$, ***known***> shadows $<M_2$, ***referable***>, so M_2 is no longer ***referable*** in S. N_1 is still visible in S, so the edge resulting from the first import is still valid. N_2 is visible in M_1, so the second import succeeds on this iteration, adding an edge from vr_0 to vr_7, importing N_2.

Step 3 N_2 is now ***referable*** in vr_7, and ***known*** throughout S, shadowing visibility of N_1 in S. Since N_2 is not ***referable*** in vr_6, the edge corresponding to the first import is no longer valid, because no binding of "N" is visible there. Thus, the edge from vr_1 to vr_6 is removed. The edge corresponding to the second import is still valid.

Step 4 Because the edge importing M_1 was removed, M_1 is no longer ***known*** in S, and thus M_2 is not shadowed and is again ***referable*** in all visibility regions in R. There is still no binding of "N" visible in vr_5, so the first import still fails. As in Step 1, there is no binding of "N" visible in M_2, so the second import fails, and the corresponding edge must be removed.

Step 5 The state at this step is identical to Step 1, so a cycle has been established that will not terminate without external intervention.

The example in Figure 6.22 is clearly an illegal program, because there can be no set of visible bindings of "N" and "M" in S such that both import statements can be satisfied. The first import statement requires that M be contained within N, while the second import requires that N be contained within M.

Traditional implementations of symbol tables do not have problems of the sort illustrated in the two examples because they use a strict left-to-right implementation. Once a reference has been bound, it is never changed. This is made possible by language designers by requiring ordering of visibility constructs so that one-pass, left-to-right evaluation, with some back-patching, is sufficient. The key is in the need for back-patching. The more general Inheritance Graph Model avoids the concept of back-patching by directly representing the language features that result in back-patching, using edges that flow in the direction opposite to text order. The Pascal program fragment in Figure 6.23 illustrates the need for back-patching in Pascal.

A naive implementation of Pascal's visibility rules will bind the reference "T1" in P to the first declaration of "T1", when in fact the correct interpretation is that the reference "T1" is an illegal forward reference. This is a common bug in Pascal compilers. In a strictly left-to-right pass over the program, an illegal forward reference in this situation is not detectable. Some form of back-patching must be added to detect if a name declared in a scope was referenced earlier in the scope. Dependencies of this sort are hard to understand, and non-obvious cases are easily overlooked, as the case where a $T1$ is declared both in the same scope as $T2$ and in the enclosing scope.

The Inheritance Graph Model does not depend on strict left-to-right evaluation, but rather can be considered as a system of constraints, to be solved by any appropriate means. An oscillating evaluation occurs because satisfying one constraint causes another constraint to become unsatisfied, and vice-versa. The key in avoiding problems with oscillating inheritance graphs is

```
...
type T1 = ···; { first declaration of T1 }
procedure P( ··· );
begin
        type T2 = T1;
        type T1 = ···; { second declaration of T1 }
...
end
```

Figure 6.23: Illegal Forward Reference in Pascal

detecting when they occur (or when they *can* occur).

Ideally, we would like to be able to analyze an inheritance graph description to detect the possibility of edge↔ref dependencies. Any inheritance graph description in which this can occur can result in oscillation, and should be modified to avoid or at least detect edge↔ref dependencies. §6.8.3 will discuss methods of avoiding oscillating edges in an inheritance graph, as well as methods of detecting and halting an oscillating evaluation.

Dynamic Definitions and Ambiguity

Dynamic definitions are those created by *define_vis* calls that depend on visibility in an inheritance graph. The dynamic definitions described in §6.6.9 can easily cause ambiguities because of their power. If the inheritance graph description is ordered and not cyclical, dynamic definitions can cause no problems: IG_j can have definitions added that depend on IG_i, $i<j$, but there are no circularities to cause problems. The Ada **use** statement falls into this category: ***potentially_visible*** depends on ***directly_visible***, but not the other way around, so the set of ***potentially_visible*** definitions is well-defined.

If the inheritance graph description is cyclical, the set of dynamic definitions can depend (indirectly) on itself, possibly causing an oscillation. One way this can occur is through an interaction with the creation of dynamic edges, resulting in a situation similar to that shown in the example in Figure 6.20. Details of such an example are left for the reader.

The second situation does not involve oscillating edges, but does involve an oscillation in the set of dynamic definitions in an inheritance graph. Assuming an ordered inheritance graph IG and a fixed edge set, this can occur only if there is a dependency of a *define_vis* call defining vc_i visibility on vc_j visibility, for $i<j$, at some vertex v_k. Thus, the set of definitions in IG_i depends on IG_j. The definitions added to IG_i can affect shadowing in IG_{i+1}, ··· ,IG_j, reducing the set of bindings with visibility class vc_j visible at v_k. This can in turn reduce the set of dynamic vc_i definitions added, affecting shadowing in IG_{i+1}, ··· , IG_j, *increasing* the set of bindings with visibility class vc_j visible at v_k. This oscillation can continue, with no fixed point.

If the inheritance graph description is ordered, there are no forward dependencies of dynamic definitions (dynamic definitions dependent on later visibility classes), and the inheritance graph is not cyclical (no construction operations dependent on reference operations), then the inheritance graph description is well-defined, and each inheritance graph generated from the description will have a single valid assignment. If there are forward dependencies, then the inheritance graph description must be carefully analyzed to determine whether the inheritance graph description is well-defined. A forward dependence between visibility classes due to a dynamic definition may not actually produce a circularity in an inheritance graph produced from the description, or may produce an inheritance graph with a circularity with a well-defined least fixed point.

6.8.3. Avoiding Oscillating Inheritance Graph Evaluations

This section considers methods of avoiding oscillating inheritance graph evaluations. Only oscillating edges are considered, because dynamic edges are more common in language descriptions than dynamic definitions, and because the problems involved are very similar in the two cases.

Analyzing Inheritance Graph Descriptions

Analyzing an inheritance graph description to detect the possibility of edge $\leftrightarrow$ ref dependencies is very difficult. Suppose the inheritance graph for a language is described using an attribute grammar, as in Chapter 7. One possible approach to analysis is to compute bottom-up and/or top-down summary information at each production about potential paths in occurring in inheritance graphs described by the attribute grammar. For "local" edges, this is not difficult: methods similar to those for computing characteristic graphs apply [Farrow 1984; Farrow 1986; Knuth 1968].

However, the presence of dynamic inheritance edges throws the proverbial "monkey wrench" into the works. The location of a dynamic inheritance edge cannot be determined until evaluation time, because the location of the edge depends on program-specific information such as what bindings are declared and where bindings are shadowed. Thus, the analysis must be very pessimistic, with the result that little useful information will be obtained. The situation is even worse when multi-step addition of dynamic edges is considered.

Causes of Oscillations

The oscillation of edges in the inheritance graphs corresponding to the examples in Figure 6.21 and Figure 6.22 was due to real errors in the programs. The error in Figure 6.21 is "clashing declarations," using the definition that two bindings ***known*** in the same scope is an error. The problem that causes the oscillation is the fact that a ***known*** binding shadows any ***referable*** binding of the same name throughout the scope, so if two bindings of the same name are ***known*** in the same scope, each ***known*** binding shadows the ***referable*** visibility of the other binding with the same name. In the example in Figure 6.21, this occurs, and causes the reference depended on by the second import statement to be invalidated. Preventing this shadowing in the event of a clashing declaration error would solve the oscillation problem in this case, but I have been unable to develop a clean method of achieving the desired effect. This is a problem for future consideration.

The oscillation in the example in Figure 6.22 is due to an illegal forward reference, which can be detected in Step 3, where N_2 is ***known*** but not ***referable*** at vr_5, where the lookup for the first import is performed. The first import was successful in Step 1 only because N_1 is pervasive and automatically inherited into S, resulting in N_1 being ***referable*** at vr_5.

A hypothesis is that all oscillating inheritance graph evaluations result from program errors. It is important to distinguish between "true" errors, such as clashing declarations and illegal forward references, and "unresolved reference" errors. The latter are a normal part of an intermediate evaluation step (for example, resulting from a multi-step import), while the former are not. If the hypothesis is true (considering only "true" errors), the inheritance graph description contains checks for all visibility-related errors, and we halt evaluation after completing the iteration in which the first "true" error is detected, then all iterative inheritance graph evaluations based on that description will terminate. The edge-oscillation in both example programs can be prevented by this method. Two disadvantages to this approach exist:

(1) The hypothesis has not been proven.

(2) The onus of checking for all visibility errors is placed on the person writing the inheritance graph description. The omission of such an error check can result in an oscillating

evaluation when some user attempts to compile a program. An oscillating evaluation can be detected and halted, as will be discussed in the next section, but the user will see a "compiler error," which is undesirable. Of course, the corresponding situation in a traditional implementation where the implementor has omitted an error check must be considered: an illegal program will be compiled without complaint by the compiler, resulting in unpredictable results.

Another (unproven) hypothesis is that a reference depended on by a dynamic edge, once resolved, will never become unresolved in a legal program, resulting in the removal of the edge. All examples considered satisfy this hypothesis. This hypothesis is related to the previous hypothesis, because this hypothesis states that any removal of an edge results from a program error, even if the evaluation eventually becomes stable and halts. If we hold this hypothesis to be true, then the evaluation iteration can be halted if an edge is removed. This must be reported to the user as a compiler error.

Halting an Oscillating Evaluation

None of the methods discussed thus far are guaranteed to detect all oscillating evaluations. The writer of the inheritance graph description may simply leave out some necessary error checks, as in the illegal forward reference problem, resulting in oscillation for some programs (however unlikely it is that anyone may write such programs). A fail-safe method for detecting an oscillating evaluation is a necessity.

An evaluation of an inheritance graph oscillates if the state at iteration i is identical to the state at iteration j, $j>i+1$ (if $j=i+1$, then the iteration has stabilized and the evaluation is complete). Discovering an oscillating evaluation using this test is expensive if the entire state at iteration j must be compared to the entire state at iteration i, for $1<i<j$. This complete comparison can be avoided because the visibility in each iteration of the evaluation of the inheritance graph can be completelyf characterized by the set of edges and definitions in the inheritance graph: more precisely, the set of dynamic edges and definitions (those dependent on references). This is so because the set of other edges and the set of definitions in the inheritance graph are fixed. Since the number of dynamic edges will normally be small in comparison to the size of the inheritance graph, the cost for testing equality of evaluation states can be kept quite small. A hash function on the dynamic edges in each iteration can be used to further speed comparisons. Of course, this comparison can be very slow in the worst case (many dynamic edges in comparison to the size of the graph, with many states mapping to the same hash value), but worst-case or near-worst-case examples are extremely unlikely.

Writing Descriptions to Avoid Oscillation

Oscillating evaluations can be avoided by writing inheritance graph descriptions such that oscillations clearly can not occur. The general idea is to prevent cycles where the reference depended on by a dynamic edge can depend on the edge itself. This may be very difficult or impossible for some languages, and it is probably preferable to use other methods to prevent or halt oscillation if avoiding cycles will significantly complicate the inheritance graph description.

An important method of preventing oscillation is by adding appropriate error checks, for example for multiple declaration and forward reference, and halting when a "true" error is detected. These two error checks are specific instances of checks for interference of a binding/visibility-class pair $<b, vc_i>$ inherited via a dynamic edge e with the visibility of a pair $<b', vc_j>$ depended on by e.

Language Design to Avoid Oscillation

A language designer can help the language implementor avoid the problem of oscillation inheritance graph evaluations, while at the same time making the language easier to understand, by making edge $\leftrightarrow$ ref dependencies impossible to create. A language that requires declaration-before-use and doesn't allow exports of the form "**export** M **to** N;", while avoiding the need for back-patching, satisfies this condition. The back-patching requirement in the example in Figure 6.22 can be avoided by using declaration contour-relative shadowing, instead of scope contour-relative shadowing (§4.4). Using declaration contour-relative shadowing, N_1 will still be ***referable*** in vr_5 at Step 3 in Table 6.1, and the iteration will stabilize with a valid assignment.

Unfortunately, satisfying this condition in a language design may be overly restrictive, outlawing visibility constructs that are useful and non-problematic when applied legally. A weaker and more practical restriction is to require that any actual edge $\leftrightarrow$ ref dependency be illegal, and easily detectable by an inheritance graph description, so that the implementation can check for any such dependency and signal an error to halt the iteration.

6.9. Summary of the Inheritance Graph Model

The Inheritance Graph Model is a natural and general model of visibility control that is useful for a wide variety of purposes. It permits the direct representation of the actual structure of visibility in a program, free from implementation or other details that might detract from the expression and understanding of the fundamental visibility control features of a language. Its greatest asset is its construction from a few very general basic concepts: visibility regions, multiple visibility constructs, definitions (of visibility of binding/visibility-class pairs), inheritance, and clash resolution. All of the Inheritance Graph Model's descriptive capabilities arise from these few concepts.

The Inheritance Graph Model is very general: all visibility features considered can be described using its basic primitives.

CHAPTER 7

A Modula-2 Example

This chapter presents a description of the inheritance graph for a complete language: Modula-2 [Wirth 1982]. Modula-2 was chosen as the example for several reasons:

(1) It has fairly complex visibility rules, so describing them was expected to be a reasonable test of the descriptive power of the inheritance graph.

(2) I was not extremely familiar with Modula-2 prior to the description effort, so the experience was expected to be helpful in determining whether the Inheritance Graph Model is useful in increasing understanding of a language.

(3) Wirth's most popular earlier language, Pascal [Jensen and Wirth 1974] had a number of ambiguities [Welsh *et al.* 1977]. I hoped that describing the inheritance graph for Modula-2 would assist in uncovering any ambiguities.

An attribute-grammar-like method is used for describing the inheritance graph for Modula-2. This method was chosen only because the author is familiar with attribute grammars. However, it is important that the reader keep in mind that the choice of an attribute grammar is not very important – it is merely a means for describing the correspondence between an abstract syntax tree and an inheritance graph. The inheritance graph is applicable to almost any descriptive formalism used for programming languages.

The description is not intended for input as-is into a system for building inheritance graph-generators. Rather, it is intended as an exercise in describing the relation between programs of a real language and the corresponding inheritance graphs. For this reason, some information that would be necessary input to an inheritance graph-generator system is omitted. Other liberties are also taken in the description, as will be described in later sections.

Following the grammar describing Modula-2, §7.7 discusses the experience of developing the inheritance graph description for Modula-2, covering both the utility of the Inheritance Graph Model in describing the visibility rules of Modula-2, and the use of the attribute grammar formalism for describing inheritance graphs.

7.1. Summary of Visibility Control Rules of Modula-2

Modula-2 is an extension of Pascal in most aspects, including the visibility control rules. It has block structure, records, and **with** statements as in Pascal. A declaration local to a scope S shadows clashing bindings at the beginning of S, not at the point of declaration.

A declaration of a binding must precede any reference to that binding *in a declaration*, but unlike Pascal, a reference to a binding in a statement can precede the declaration of the binding. This distinction apparently has no real purpose other than to match the intended implementation: declarations are processed sequentially in the first pass, making declaration-before-use necessary, while references in statements are processed in a later pass, where it is easier to eliminate the declaration-before-use requirement.

As in Pascal, the declaration-before-use requirement is relaxed for references to types in pointer-type declarations, to make mutually recursive data types possible.

In addition to procedures, functions, and blocks, Modula-2 has modules. Modules are closed scopes to all bindings except bindings of standard identifiers, which are pervasive. All

other scopes are open scopes.

Modula-2 provides several forms of import and export statements for modifying the visibility of bindings across module boundaries. The import statement makes some or all of the bindings in the enclosing scope or in a named scope visible in the importing module. An export in module M_1 makes directly visible bindings visible in the scope enclosing M_1, or in a module that imports M_1. Import and export are symmetric; an exported binding can be imported by the module inheriting visibility of the binding or *vice-versa.*

Several ambiguities in the visibility control rules of Modula-2 were discovered while describing the inheritance graph. The ambiguities are presented here along with the chosen interpretations used in the inheritance graph description for Modula-2.

(1) Are two declarations of the same identifier in the same scope an error? For our description of Modula-2, this is defined to be an error.

(2) Are imported/exported bindings the same as locally declared bindings for the purpose of checking for clashing local bindings? Yes.

(3) Do imported/exported bindings have the same visibility as locally declared bindings? In particular are bindings imported into a definitions module also visible in the corresponding implementation module? Yes.

(4) Given a module M with a local declaration A, must A be exported from M for a qualified reference "M.A" nested within M to be legal? Yes, although there are strong arguments on both sides.

(5) Are the lookup rules for import/export the same as the rules for references in declarations, or in statements? The export statement only makes sense if all bindings declared local to a scope are visible, since the export precedes any declarations in a scope. The meaning of an import isn't clear: in the inheritance graph description, only prior declarations are visible.

(6) What is the scope of an enumeration constant (particularly those declared in an anonymous type nested in a record declaration)? In the inheritance graph description, it is the innermost enclosing block, procedure, function, or module.

Other points of Modula-2's visibility control rules will be discussed where relevant in the description of the inheritance graph.

7.2. Design of the Inheritance Graph for Modula-2

Four visibility classes are used to describe the visibility control rules of Modula-2:

A binding is ***known*** in visibility regions where no clashing binding may be declared. In the description, a binding is ***known*** in the visibility regions local to the declaring scope, and in any visibility region that receives the ability to reference a binding because of an import or export statement. Imported bindings are ***known*** because they are equivalent to local declarations for error detection[41]. The bindings made visible in the scope of a **with** statement are also ***known***, because they must shadow clashing bindings visible in the scope enclosing the **with** statement. ***known*** visibility flows both forward and backward in text order in the graph.

A binding is ***decl_referable*** in all visibility regions where it may be referenced in a declaration. ***decl_referable*** visibility flows forward in text order in the graph, and inward into nested scopes.

A binding is ***stmt_referable*** in all visibility regions where it may be referenced in a statement. ***stmt_referable*** visibility flows forward and backward in text order in the graph, and

[41] That is, I have chosen to treat them this way. As stated earlier, the language manual is not clear on this point.

inward into nested scopes. Both ***decl_referable*** and ***stmt_referable*** visibility classes are needed because of the declaration-before-use requirement on references in declarations, but not references in statements.

The ***containing*** visibility class is used for bindings corresponding to modules. ***containing*** visibility of a binding of module M is defined at the first visibility region local to M, and is inherited forward in text order and into all nested scopes, with no shadowing. Thus, the binding of M is ***containing*** visible at all visibility regions contained within M, even visibility regions local to scopes nested within M. The ***containing*** visibility class is used to resolve nested qualified references, which may occur within modules, but not in records. The method for handling nested qualified references was described in the Nested Qualified References sub-section of §6.5.4, along with the appropriate subgraph schema.

7.3. Clash Function and Clash Table for Modula-2

The clash function for Modula-2 simply returns **true** for any pair of bindings, since *clash* is called with a pair of bindings only if their *name* fields are the same. and there is no overloading in Modula-2.

The clash table for Modula-2 is:

Modula-2 Clash Table		Visibility Class for b_2			
		known	***decl_referable***	***stmt_referable***	***containing***
Vis Class for b_1	***known***	error("clashing decls")	**shadow** b_2	**shadow** b_1	–
	decl_referable	**shadow** b_1	error("descrip. error")	error("descrip. error")	–
	stmt_referable	**shadow** b_1	error("descrip. error")	error("descrip. error")	–
	containing	–	–	–	–

The important points of the clash table are that ***known*** shadows ***decl_referable*** and ***stmt_referable*** bindings, and nothing shadows ***containing*** bindings. The entries filled with "descrip. error" should only be activated if there is an error in the description, and are included as an extra check on the correctness of the description.

Several lookup functions are required for use in different contexts. They are given in §7.6.1.

7.4. Subgraph Schema for Modula-2

All of the visibility control constructs in Modula-2 are very similar to the examples given in Chapter 6. Instead of repeating those schemata with minor modifications, the differences between Modula-2 and those examples will be described at relevant points in the inheritance graph description itself.

7.5. Introduction to the Inheritance Graph Description

7.5.1. The Attribute Grammar-Like Formalism

The inheritance graph description looks like an attribute grammar, and borrows many features from attribute grammars. The discussion assumes the reader is somewhat familiar with attribute grammars. For an introduction to attribute grammars, see [Farrow 1983], [Waite and Goos 1984], or [Aho *et al.* 1986]. Like a true attribute grammar, the description uses inherited and synthesized attributes[42] associated with nodes of the tree corresponding to a program, where the values of inherited attributes are computed from above the attribute's node in the tree, and the

[42] It is important not to confuse attributes of tree nodes in the attribute grammar and attributes of entities and bindings. Which is meant should be clear from the context of a use of the word "attribute".

values of synthesized attributes are computed from below. Semantic functions associated with each production define the values of synthesized attributes on the LHS of the production, and the values of inherited attributes on the RHS of the production.

However, the inheritance graph description is neither a complete nor valid attribute grammar. It is not complete because some (hopefully) obvious details are omitted, such as complete definitions of all attributes for all tree nodes. The description builds only as much type information as is needed for name resolution. A complete language description would include more type information, as required for type checking. Some simple functions are just described instead of being written in full.

The description is not a valid attribute grammar because some operations cause side-effects, which are banned in a true attribute grammar, and also because there are circular dependencies in the description. These circularities arise because of the ordering problem discussed in §6.7.

The side-effects result from the calls on *create_edge*, *create_vertex*, *define_vis*, and *redefine_vis*. This problem could be eliminated by adding additional attributes and dependencies, such that:

(1) A single, synthesized "IG" attribute depends (directly or indirectly) on all inheritance graph construction operations, and whose evaluation involves re-evaluating the inheritance graph.

(2) All reference operations depend on the "IG" attribute, so that all construction operations must be executed before any reference operations.

The circularity problem could then be handled by the iterative algorithm described in §6.7, with additional ordering resulting from the attribute dependencies.

7.5.2. Notation

Common attribute grammar notation is used for the most part, though not entirely. Multiple assignment and conditional assignment are used.

Two types of extensions to attribute grammars are used to shorten and simplify the grammar. The first is the INCLUDING operator, where an occurrence of

INCLUDING *attribute-name*

occurring in an expression returns the value of the first *attribute-name* attribute found by traversing up the parse tree from the point of reference to the root. This is used for attributes that are defined once for each scope, and used in many subtrees of the scope. The INCLUDING operator was introduced in the GAG system [Kastens *et al.* 1981].

The other extensions are based on constructs proposed in [Jüllig and DeRemer 1984] for describing attribute flow in regular right-part attribute grammars. The Modula-2 description uses a different notation, however. The notation is needed for productions of the form:

```
definition_seq
        :       definition+
&SEMANTICS
        definition'FIRST.first_vr = definition_seq.first_vr;
        create_edge (definition'I.last_vr, definition'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (definition'I+1.first_vr, definition'I.last_vr,
                {known, stmt_referable}, rf_null);
        definition_seq.last_vr = definition'LAST.last_vr;
        ;
```

Productions with regular right parts (such as this one) are not allowed in conventional attribute grammars, because there is an instance of each attribute of *definition* for each *definition* in a *definition_seq*, and each attribute instance may receive a different value. Conventional attribute grammars avoid this problem by disallowing regular right parts – forcing sequences to be written as recursions, which tend to be long and tedious in an attribute grammar. The extensions (of which some are illustrated in the above example) are:

FIRST
: *node'FIRST.attr* denotes the *attr* attribute of the first *node* in the sequence.

LAST
: *node'LAST.attr* denotes the *attr* attribute of the last *node* in the sequence.

I and I+1
: Semantic functions containing references of the form *node'I.attr* and *node'I+1.attr* are (in effect) replicated for $1 \le I \le n-1$, where n is the length of the sequence. This permits an element of the sequence to depend on either neighbor.

ALL
: Semantic functions containing references of the form *node'ALL.attr* and *node'I+1.attr* are (in effect) replicated for $1 \le I \le n$. This allows all attributes in a sequence to receive the same value.

CONCATENATE
: A reference of the form *node'CONCATENATE.attr* on the RHS of a semantic function has as its result a list whose elements are the values of *attr* for each element of the regular right part.

7.6. The Inheritance Graph Description

7.6.1. Types and Functions Used in the Inheritance Graph Description

The type *Name* will be used to represent names. The standard techniques for implementing a string table [Aho *et al.* 1986, p. 431] can be used to translate objects of type *Ident* to type *Name* and vice-versa. The functions *ident_to_name* and *name_to_ident* are used for this purpose.

The describer-defined types for bindings and entities follow: The choice of whether an attribute should be part of a binding or an entity depends on whether different bindings of an entity can have different attribute values. The question is moot if only one binding of each entity can exist.

```
Binding = ^BindingRecord;
Entity = ^EntityRecord;

BindingRecord =
  record
      name : Name;
      entity : Entity;
      exported : Boolean; -- true if binding is exported qualified
  end ;

EntityRecord =
  record
      name    : Name; -- for mapping from entity to identifier
      pervasive : Boolean; -- standard bindings are pervasive
      readOnly : Boolean; -- for bindings exported readonly
      declaringScope : Entity; -- Module or scope containing a decl
      kind : EntityKind;        -- module_decl, procedure_decl, etc.
      ··· other semantic information ···
  end ;
```

Pointers are used for bindings and entities so that equality of bindings can be defined to be equality of pointers. "@" is used as the address operator. In an implementation, sequential indices would most likely be used instead of pointers so that sets of bindings could be more easily represented.

The types *IGVertex* and *IGEdge* represent vertices and edges in the inheritance graph, and are presumed to be predefined.

Procedures *define_vis* and *redefine_vis*, and function *visible* are as defined in Chapter 6. Function *create_vertex* creates and returns a new inheritance graph vertex. Function *create_edge* creates, as a side effect, an edge from v_1 to v_2 for each visibility class $vis\text{–}class_i$ in *visTypeSet*, with visibility class $vis\text{–}class_i$ and restriction function *restrictionFunction*.

In the grammar, all procedures, for example *define_vis*, that normally take a single visibility class as an argument will be called with a set of visibility classes. This is done in order to avoid a lot of redundancy. Each of these calls can be considered to be a macro call that expands to a separate call for each visibility class, with the condition that all other parameters to the macro call are only evaluated once.

```
function create_vertex : IGVertex;

procedure create_edge (
      v1, v2 : IGVertex ;
      visType: set of VisibilityType;
      restrictionFunction: function (Binding) : Boolean);
```

Several lookup functions are needed for use in different contexts: *statement_lookup* is used for references occurring in statements. *restrict_unique* checks that a unique binding is returned by the call on *visible* in *statement_lookup*. If not, *restrict_unique* produces an appropriate error message and returns *error_binding*, which is a standard error value used to avoid much error checking in other places. *restrict_unique* is as defined in §6.2.4.

```
function statement_lookup (
    name    : Name,
    v       : IGVertex) : Binding;

    bindings : Set of Binding;

    bindings := (visible (name, v, stmt_referable));
    return restrict_unique (bindings, name, v);
```

declaration_lookup is used for references occurring in declarations. An illegal forward reference to a binding occurs if the binding is ***known*** but not ***decl_referable***.

```
function declaration_lookup (
    name    : Name,
    v       : IGVertex) : Binding;

    bindings : Set of Binding;

    bindings := visible (name, v, decl_referable);
    num_bindings = size (bindings);
    if num_bindings = 0 then
        known_bindings := visible (name, v, known);
        if size (known_bindings) > 0 then
            error (v, "Illegal forward reference to '%s'" name_to_ident(name),
        else
            error (v, "'%s' referenced but not visible", name_to_ident(name));
        return (error_binding);
    else if num_bindings = 1 then
        return (select b:Binding from bindings);
    else
        error (v, "More than one declaration of '%s' visible", name_to_ident(name));
        return (error_binding);
    return restrict_unique (bindings, name, v);
```

The lookup function for references occurring in pointer type definitions is *pointer_type_lookup*. A binding *b* is found by *pointer_type_lookup* if it is either ***decl_referable*** (regular visibility for declarations) or it is ***known***. A binding is ***known*** throughout its local scope, so bindings declared later in the same scope will be found, as is appropriate in Modula-2.

```
function pointer_type_lookup (
    name    : Name,
    v       : IGVertex) : Binding;

    bindings : Set of Binding;

    bindings := visible (name, v, known) ∪ visible (name, v, decl_referable);
    return restrict_unique (bindings, name, v);
```

check_import_visible is used to verify that bindings being imported actually exist. This must be done in addition to creating the edge along which visibility of the bindings can be inherited. There is no need to check whether the imported binding clashes with a local declaration, because clash resolution will detect this.

```
procedure check_import_visible (
    name_list       : list of Name;
    from_vr,        -- visibility region being imported from
    import_vr : IGVertex    -- visibility region at import statement
    );
    b : Binding;

    foreach name:Name ∈ name_list do
        -- An error message will be produced at the providing visibility region if the lookup fails
        b := declaration_lookup (name, v)
        if b = error_binding then
            error ("Imported binding %s not visible", name_to_ident(name))
```

check_export_visible is used to check that bindings being exported are ***known*** in the exporting scope.

```
procedure check_export_visible (
    name_list       : list of Name;
    from_vr                 : IGVertex  -- visibility region being exported from
    );
    b : Binding;

    foreach name:Name ∈ name_list do
        b := restrict_unique (visible (name, from_vr, known))
        if b = error_binding then
            error ("Exported binding %s not directly visible", name_to_ident(name))
```

Several restriction functions are used on inheritance edges in the Modula-2 description. *rf_null* is the default restriction function. It places no restriction on inheritance, returning **true** for all bindings. *rf_closed_scope* is used on the edge connecting the visibility region prior to the declaration of a module to the first visibility region local to the module. Only pervasive bindings are inherited by a closed scope.

```
function rf_closed_scope (
    b : Binding
    ) : Boolean;

    return (b.entity.pervasive);
```

Function *rf_local_decls* creates and returns a closure that returns **true** only for bindings that are local to the scope represented by the binding passed to *rf_local_decls*. A closure is needed because the restriction function depends not only on the class of the visibility construct, but also on the specifics of the instance of the visibility construct (*i.e.*, which bindings are declared local to the scope in question). *rf_local_decls* is used for the inheritance edge from a definition module to the corresponding implementation module, and for the edge from the last field of a record to the scope of a **with** statement opening the record.

```
function rf_local_decls (
    declaring_scope : Binding
    ) function (Binding) : Boolean;

    function rf_local_decls_closure (b : Binding) : Boolean;
        return (b.declaring_scope = declaring_scope)

    return (rf_local_decls_closure);
```

redefine_declaring_scope is called for each declaration, and is used to redefine the *ref_env* attribute of the scope containing the declaration. This is needed when the declaring scope is a module, because of the possibility of nested qualified references.

```
procedure redefine_declaring_scope (vr: IGVertex, binding: Binding);
    new_binding = @BindingRecord(binding.name,
            @EntityRecord(binding.name,
                    pervasive = binding.pervasive,
                    readOnly = binding.readOnly,
                    declaring_scope = binding.declaring_scope,
                    kind = binding.kind,
                    ref_env = vr,
                    descriptor = binding.descriptor,
                    ...))
    redefine_vis (vr, binding, new_binding, containing);
```

Errors are handled by calls to *error*. As discussed in §6.4, errors must be associated with some object (a binding, entity, or inheritance graph vertex) in order to avoid duplicate error messages, and so that errors can be printed in meaningful places. However, this extra argument to calls on *error* will be omitted, because it doesn't add much to the understanding of the inheritance graph description.

7.6.2. Attributes of Tree Nodes

in_vr is the visibility region in which references should be resolved, and is inherited by all tree nodes where references may occur.

```
in_vr    : inh : IGVertex;
```

first_vr and *last_vr* are used to keep track of the first and last visibility regions resulting from a subtree rooted at a particular node: for example the first and last visibility regions resulting from a list of declarations. They are used primarily to allow the vertices corresponding to different parts of a production to be joined by edges. For example, in a list of declarations, each declaration has its own *first_vr* and *last_vr*, which are connected together in a chain with inheritance edges. In some cases where a production may produce an empty list, a dummy vertex vr_{dummy} is created so that *first_vr* and *last_vr* have valid values, and the creation of edges higher in the tree can ignore the possibility of empty sequences. This can actually result in a single visibility region corresponding to more than one vertex in the inheritance graph, but the meaning of references is unaffected, so we can ignore the extra vertices.

```
first_vr: synth : IGVertex;
last_vr  : synth : IGVertex;
```

declaring_scope is defined at each scope that can contain local declarations. Instead of propagating it step-by-step through the production, it is accessed where needed by the "INCLUDING" construct, which finds the closest occurrence of a *declaring_scope* attribute

above the reference in the attributed tree.

```
declaring_scope : Entity;
```

following _vr in a module_declaration is the visibility region following the visibility region where visibility of the module is defined. It is used as the target for visibility resulting from an export statement.

```
following_vr : IGVertex;  -- following visibility region in enclosing scope
```

enclosing _decl _vr is a, and is set by any declaration immediately local to a scope. It is used by enumeration constants as their definition vertex.

```
enclosing_decl_vr : IGVertex;
```

7.6.3. The Grammar

The attribute grammar describing the inheritance graph for Modula-2 follows. Explanatory paragraphs are interspersed in the grammar. A discussion of the development of the inheritance graph description of Modula-2 follows in §7.7.

```
-- Modula-2 grammar
-- context-free grammar (c) Copyright Robert A. Ballance,
-- Jacob Butcher, Michael L. Van De Vanter 1987.
-- (Work in progress). All rights reserved.
-- Modifications, annotations, and semantic actions by Phillip Garrison

Initialization of global constants

enclosing_scope_name: Name = ident_to_name ("#enclosing_scope#");
```

The *in _vr* for a compilation unit is the standard environment, which must be provided by the person writing the language system. The standard environment is simply a visibility region at which all the standard bindings are visible, just as if the standard environment enclosed the compilation unit.

```
compilation_unit
        :       definition_module
&SEMANTICS
        definition_module.in_vr = compilation_unit.in_vr;
        compilation_unit.first_vr = definition_module.first_vr;
        compilation_unit.last_vr = definition_modulelast_vr;
        |       program_module
&SEMANTICS
        program_module.in_vr = compilation_unit.in_vr;
        compilation_unit.first_vr = program_module.first_vr;
        compilation_unit.last_vr = program_module.last_vr;
        |       implementation_module
&SEMANTICS
        implementation_module.in_vr = compilation_unit.in_vr;
        compilation_unit.first_vr = implementation_module.first_vr;
        compilation_unit.last_vr = implementation_module.last_vr;
        ;

program_module
        :       "MODULE" id opt_priority ";" import_seq block id "."
```

```
&SEMANTICS
        DECL_VR: IGVertex = create_vertex();
        program_module.first_vr = DECL_VR;
        program_module.last_vr = DECL_VR;

        NAME:Name = ident_to_name(id<1>);
        program_module.name = NAME;
        MODULE_BINDING: Binding =
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = null_Binding,
                        kind = program_module_decl,
        -- unique_id used by redefine_vis to check entity equality
                        unique_id = GetUniqueName(),
                        ref_env = block.last_vr;
                        ... )),
        define_vis (DECL_VR, MODULE_BINDING,
                {known, stmt_referable, decl_referable})

        -- first visibility region local to module, a local temporary
        FIRST_LOCAL_VR:IGVertex = create_vertex();
        opt_priority.in_vr = FIRST_LOCAL_VR;

        -- define containing visibility
        define_vis (FIRST_LOCAL_VR, MODULE_BINDING, {containing})
        -- edge corresponding to scope entry
        create_edge (DECL_VR, FIRST_LOCAL_VR,
                {stmt_referable, decl_referable}, rf_closed_scope);
        create_edge (DECL_VR, FIRST_LOCAL_VR,
                {containing}, rf_null);
        create_edge (FIRST_LOCAL_VR, import_seq.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (import_seq.first_vr, FIRST_LOCAL_VR,
                {known, stmt_referable}, rf_null);
        create_edge (import_seq.last_vr, block.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (block.first_vr, import_seq.last_vr,
                {known, stmt_referable}, rf_null);
        block.in_vr = import_seq.last_vr;

        -- define #enclosing_scope# for use by import
        define_vis (FIRST_LOCAL_VR,
                @BindingRecord(enclosing_scope_name,
                        @EntityRecord(enclosing_scope_name,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = MODULE_BINDING,
                        kind = special_decl,
                        ref_env = program_module.in_vr ... )),
```

```
                {known, stmt_referable, decl_referable})
        block.declaring_scope = MODULE_BINDING;
        ;

opt_priority
        :       "[" const_expr "]"
&SEMANTICS
        const_expr.in_vr = opt_priority.in_vr;
        |       -- empty
        ;

implementation_module
        :       "IMPLEMENTATION" "MODULE" id opt_priority ";"
                import_seq block id "."
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        implementation_module.first_vr,
        implementation_module.last_vr = DUMMY_VR;

        NAME:Name = ident_to_name(id<1>);
        -- add inheritance from last vis region in definitions module
        -- to first vis region in body of implementation module
        PROVIDING_SCOPE:Binding =
                statement_lookup (NAME, implementation_module.in_vr);
        -- rf_local_decls returns a restriction function that
        -- permits inheritance only of local bindings of the scope entity
        -- passed to it as its argument
        create_edge (PROVIDING_SCOPE.entity.ref_env, FIRST_LOCAL_VR,
                {known, containing, stmt_referable, decl_referable},
                rf_local_decls(PROVIDING_SCOPE));

        -- first visibility region local to module
        FIRST_LOCAL_VR: IGVertex = create_vertex();
        opt_priority.in_vr = FIRST_LOCAL_VR;

        -- edge corresponding to scope entry
        create_edge (import_seq.first_vr, FIRST_LOCAL_VR,
                {known, stmt_referable}, rf_null);
        create_edge (import_seq.last_vr, block.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (block.first_vr, import_seq.last_vr,
                {known, stmt_referable}, rf_null);
        block.in_vr = import_seq.last_vr;
        -- define #enclosing_scope# for use by import
        define_vis (FIRST_LOCAL_VR,
                @BindingRecord(enclosing_scope_name,
                        @EntityRecord(enclosing_scope_name,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = MODULE_BINDING,
                        kind = special_decl,
```

```
                    ref_env = program_module.in_vr ... )),
                {known, stmt_referable, decl_referable})

        block.declaring_scope = PROVIDING_SCOPE;
        ;

definition_module
        :       "DEFINITION" "MODULE" id ";"
                import_seq definition_seq "END" id "."
&SEMANTICS
        DECL_VR: IGVertex = create_vertex();
        definition_module.first_vr = DECL_VR;
        definition_module.last_vr = DECL_VR;

        NAME:Name = ident_to_name(id<1>);
        definition_module.name = NAME;
        MODULE_BINDING: Binding =
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = null_Binding,
                        kind = definition_module_decl,
        -- unique_id used by redefine_vis to check entity equality
                        unique_id = GetUniqueName(),
                        ref_env = definition_seq.last_vr;
                        ... )),
        define_vis (DECL_VR, MODULE_BINDING,
                {known, stmt_referable, decl_referable})

        -- first visibility region local to module
        FIRST_LOCAL_VR:IGVertex = create_vertex();
        -- define containing visibility
        define_vis (FIRST_LOCAL_VR, MODULE_BINDING, {containing})

        -- edge corresponding to scope entry
        create_edge (DECL_VR, FIRST_LOCAL_VR,
                {stmt_referable, decl_referable}, rf_closed_scope);
        create_edge (DECL_VR, FIRST_LOCAL_VR,
                {containing}, rf_null);
        create_edge (FIRST_LOCAL_VR, import_seq.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (import_seq.first_vr, FIRST_LOCAL_VR,
                {known, stmt_referable}, rf_null);
        create_edge (import_seq.last_vr, definition_seq.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (definition_seq.first_vr, import_seq.last_vr,
                {known, stmt_referable}, rf_null);
        definition_seq.in_vr = import_seq.last_vr;
        definition_seq.declaring_scope = MODULE_BINDING;
```

```
        -- define #enclosing_scope#
        define_vis (FIRST_LOCAL_VR,
                @BindingRecord(enclosing_scope_name,
                        @EntityRecord(enclosing_scope_name,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = MODULE_BINDING,
                        kind = special_decl,
                        ref_env = definition_module.in_vr ... )),
                {known, stmt_referable, decl_referable})
        ;

import_seq
        :       import+
&SEMANTICS
        import'FIRST.in_vr = import_seq.in_vr;
        -- link vis regions created by each import
        create_edge (import'I.last_vr, import'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (import'I+1.first_vr, import'I.last_vr,
                {known, stmt_referable}, rf_null);
        import_seq.first_vr = import'LAST.first_vr;
        import_seq.last_vr = import'LAST.last_vr;
        |       -- empty
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        import_seq.first_vr, import_seq.last_vr = DUMMY_VR;
        ;

-- lookup must either always return a valid binding (with a real ref_env)
-- or we must check for errors
import  :       "FROM" id "IMPORT" id_list ";"
&SEMANTICS
        ENCLOSING_SCOPE:Binding =
                statement_lookup (enclosing_scope_name, import.in_vr);
        PROVIDING_SCOPE:Binding =
                statement_lookup (ident_to_name(id), ENCLOSING_SCOPE.entity.ref_env);
        import.first_vr = import.last_vr = create_vertex();
-- check that imported bindings exist
        check_import_visible (id_list.name_list,
                PROVIDING_SCOPE.entity.ref_env, import.first_vr);
-- rf_name_list is a function that takes a list of names as an argument
-- and returns a restriction function that returns true only for bindings
-- with one of the listed names
        create_edge (PROVIDING_SCOPE.entity.ref_env, import.in_vr,
                {known, stmt_referable, decl_referable}, rf_name_list(id_list.name_list));
        |       "IMPORT" id_list ";"
&SEMANTICS
        PROVIDING_SCOPE:Binding =
                statement_lookup (enclosing_scope_name, import.in_vr);
        import.first_vr = import.last_vr = create_vertex();
```

```
-- check that imported bindings exist
        check_import_visible (id_list, PROVIDING_SCOPE.entity.ref_env);
        create_edge (PROVIDING_SCOPE.entity.ref_env, import.in_vr,
                {known, stmt_referable, decl_referable}, rf_name_list(id_list.name_list));
        ;

export  :       "EXPORT" id_list ";"
&SEMANTICS
        -- check that exported bindings exist
        check_export_visible (id_list.name_list, export.in_vr);
        create_edge (export.in_vr, export.following_vr,
                {known, stmt_referable, decl_referable}, rf_name_list(id_list.name_list));
        |       "EXPORT" "QUALIFIED" id_list ";"
&SEMANTICS
-- just set "exported" field to true for all bindings referenced
-- in id_list: not shown.
        ;

id_list
        :       id+{","}
&SEMANTICS
        IDENT_LIST: IdentList = id'CONCATENATE;
        -- convert identifiers in list to names
        id_list.name_list = ident_list_to_name_list (IDENT_LIST);
        ;

definition_seq
        :       definition+
&SEMANTICS
        definition'FIRST.first_vr = definition_seq.first_vr;
        create_edge (definition'I.last_vr, definition'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (definition'I+1.first_vr, definition'I.last_vr,
                {known, stmt_referable}, rf_null);
        definition_seq.last_vr = definition'LAST.last_vr;
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        definition_seq.first_vr, definition_seq.last_vr = DUMMY_VR;
        ;

definition
        :       "CONST" constant_declaration_seq
&SEMANTICS
        constant_declaration_seq.in_vr = definition.in_vr;
        definition.first_vr = constant_declaration_seq.first_vr;
        definition.last_vr = constant_declaration_seq.last_vr;
        |       "TYPE" type_definition_seq
&SEMANTICS
        type_declaration_seq.in_vr = definition.in_vr;
        definition.first_vr = type_definition_seq.first_vr;
        definition.last_vr = type_definition_seq.last_vr;
```

```
        |       "VAR" var_declaration_seq
 &SEMANTICS
        var_declaration_seq.in_vr = definition.in_vr;
        definition.first_vr = var_declaration_seq.first_vr;
        definition.last_vr = var_declaration_seq.last_vr;
        |       procedure_heading
 &SEMANTICS
        procedure_heading.in_vr = definition.in_vr;
        definition.first_vr = procedure_heading.first_vr;
        definition.last_vr = procedure_heading.last_vr;
        ;

 constant_declaration_seq
        :       constant_declaration+
 &SEMANTICS
        constant_declaration'FIRST.in_vr = constant_declaration_seq.in_vr;
        create_edge (constant_declaration'I.last_vr, constant_declaration'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (constant_declaration'I+1.first_vr, constant_declaration'I.last_vr,
                {known, stmt_referable}, rf_null);
        constant_declaration_seq.last_vr = constant_declaration'LAST.last_vr;
        |       -- empty
 &SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        constant_declaration_seq.first_vr, constant_declaration_seq.last_vr = DUMMY_VR;
        ;

 type_definition_seq
        :       type_definition+
 &SEMANTICS
        type_definition'FIRST.in_vr = type_definition_seq.in_vr;
        create_edge (type_definition'I.last_vr, type_definition'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (type_definition'I+1.first_vr, type_definition'I.last_vr,
                {known, stmt_referable}, rf_null);
        type_definition_seq.last_vr = type_definition'LAST.last_vr;
 &SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        type_definition_seq.first_vr, type_definition_seq.last_vr = DUMMY_VR;
        ;

 var_declaration_seq
        :       var_declaration+
 &SEMANTICS
        var_declaration'FIRST.in_vr = var_declaration_seq.in_vr;
        create_edge (var_declaration'I.last_vr, var_declaration'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (var_declaration'I+1.first_vr, var_declaration'I.last_vr,
                {known, stmt_referable}, rf_null);
        var_declaration_seq.last_vr = var_declaration'LAST.last_vr;
 &SEMANTICS
```

```
        DUMMY_VR: IGVertex = create_vertex();
        var_declaration_seq.first_vr, var_declaration_seq.last_vr = DUMMY_VR;
        ;

declaration
        :       "CONST" constant_declaration_seq
&SEMANTICS
        constant_declaration_seq.in_vr = declaration.in_vr;
        declaration.first_vr = constant_declaration_seq.first_vr;
        declaration.last_vr = constant_declaration_seq.last_vr;
        |       "TYPE" type_declaration_seq
&SEMANTICS
        type_declaration_seq.in_vr = declaration.in_vr;
        declaration.first_vr = type_declaration_seq.first_vr;
        declaration.last_vr = type_declaration_seq.last_vr;
        |       "VAR" var_declaration_seq
&SEMANTICS
        var_declaration_seq.in_vr = declaration.in_vr;
        declaration.first_vr = var_declaration_seq.first_vr;
        declaration.last_vr = var_declaration_seq.last_vr;
        |       procedure_declaration
&SEMANTICS
        procedure_declaration.in_vr = declaration.in_vr;
        declaration.first_vr = procedure_declaration.first_vr;
        declaration.last_vr = procedure_declaration.last_vr;
        |       module_declaration
&SEMANTICS
        module_declaration.in_vr = declaration.in_vr;
        declaration.first_vr = module_declaration_seq.first_vr;
        declaration.last_vr = module_declaration.last_vr;
        module_declaration.following_vr = declaration.following_vr;
        ;

type_declaration_seq
        :       type_declaration+
&SEMANTICS
        type_declaration'FIRST.in_vr = type_declaration_seq.in_vr;
        -- link vis regions created by each type_declaration
        create_edge (type_declaration'I.last_vr, type_declaration'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (type_declaration'I+1.first_vr, type_declaration'I.last_vr,
                {known, stmt_referable}, rf_null);
        type_declaration_seq.first_vr = type_declaration'LAST.first_vr;
        type_declaration_seq.last_vr = type_declaration'LAST.last_vr;
        |       -- empty
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        type_declaration_seq.first_vr, type_declaration_seq.last_vr = DUMMY_VR;
        ;

constant_declaration
```

```
        :       id "=" const_expr
&SEMANTICS
        LOCAL_VR: IGVertex = create_vertex();
        constant_declaration.first_vr = LOCAL_VR;
        constant_declaration.last_vr = LOCAL_VR;
        const_expr.in_vr = constant_declaration.in_vr;
        NAME:Name = ident_to_name(id);
        define_vis (LOCAL_VR,
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = constant_decl,
                        ... )),
                {known, stmt_referable, decl_referable})

        redefine_declaring_scope (LOCAL_VR, constant_declaration.declaring_scope);
        ;
```

A *type_declaration* can be a completion of an opaque type declared in the corresponding definition module. If such an opaque type exists, we must redefine it instead of defining an entirely new binding.

```
type_declaration
        :       id "=" type
&SEMANTICS
        LOCAL_VR: IGVertex = create_vertex();
        type_declaration.first_vr = LOCAL_VR;
        type_declaration.last_vr = LOCAL_VR;
        type.enclosing_decl_vr = LOCAL_VR;
        type.in_vr = type_declaration.in_vr;
        NAME:Name = ident_to_name(id);
-- use known visibility class, because we want only bindings declared
-- in the definition module, not ones that are just visible
-- (decl_referable or stmt_referable)
        old_binding:Binding = visible (NAME, LOCAL_VR, known);
        if old_binding = error_binding then
                define_vis (LOCAL_VR,
                        @BindingRecord(NAME,
                                @EntityRecord(NAME,
                                pervasive=false,
                                readOnly=false,
                                unique_id = GetUniqueName();
                                declaring_scope = INCLUDING declaring_scope,
                                kind = type_decl,
                                type_def = type.type_def;
                                ... )),
                        {known, stmt_referable, decl_referable})
        else
                redefine_vis (LOCAL_VR,
```

```
                        old_binding,
                        @BindingRecord(NAME,
                                @EntityRecord(NAME,
                                pervasive=false,
                                readOnly=false,
                                unique_id = old_binding.unique_id;
                                declaring_scope = old_binding.declaring_scope,
                                kind = type_decl,
                                type_def = type.type_def;
                                ... )),
                        {known, stmt_referable, decl_referable})
        endif;
        redefine_declaring_scope (LOCAL_VR, type_declaration.declaring_scope);
        ;

type_definition
        :       type_declaration
&SEMANTICS
        type_definition.first_vr = type_declaration.first_vr;
        type_definition.last_vr = type_declaration.last_vr;
        |       id
-- opaque type definition: identifier only, no type information
&SEMANTICS
        LOCAL_VR: IGVertex = create_vertex();
        type_definition.first_vr = LOCAL_VR;
        type_definition.last_vr = LOCAL_VR;
        type.enclosing_decl_vr = LOCAL_VR;
        NAME:Name = ident_to_name(id);
        define_vis (LOCAL_VR,
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = opaque_type_decl,
                        type_def = null;
                        ... )),
                {known, stmt_referable, decl_referable})
        redefine_declaring_scope (LOCAL_VR, type_definition.declaring_scope);
        ;

var_declaration
        :       decl_id_list ":" type
&SEMANTICS
        decl_id_list.in_vr = var_declaration.in_vr;
        decl_id_list.kind = var_decl;
        decl_id_list.type_def = type.type_def;
        var_declaration.first_vr = decl_id_list.first_vr;
        var_declaration.last_vr = decl_id_list.last_vr;
        type.enclosing_decl_vr = decl_id_list.last_vr;
        type.in_vr = var_declaration.in_vr;
```

```
        -- descriptor for "type" will be created by type checking semantics
        -- for "type" and propagated down through tree where a descriptor
        -- for each variable can be created
        ;

decl_id_list
        :       decl_id+{","}
&SEMANTICS
        decl_id.type_def = decl_id_list.type_def;
        decl_id.kind = decl_id_list.kind;
        decl_id'FIRST.first_vr = decl_id_list.first_vr;
        create_edge (decl_id'I.last_vr, decl_id'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (decl_id'I+1.first_vr, decl_id'I.last_vr,
                {known, stmt_referable}, rf_null);
        decl_id_list.last_vr = decl_id'LAST.last_vr;
        ;

decl_id
        : id
&SEMANTICS
        LOCAL_VR: IGVertex = create_vertex();
        decl_id.first_vr = LOCAL_VR;
        decl_id.last_vr = LOCAL_VR;
        NAME:Name = ident_to_name(id);
        define_vis (LOCAL_VR,
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = decl_id.kind,
                        type_def = decl_id.type_def;
                        ... )),
                {known, stmt_referable, decl_referable})
        redefine_declaring_scope (LOCAL_VR, decl_id.declaring_scope);
        ;

procedure_declaration
        :       procedure_heading block id ";"
&SEMANTICS
        procedure_heading.in_vr = procedure_declaration.in_vr;
        LOCAL_VR: IGVertex = create_vertex();
        procedure_declaration.first_vr = procedure_heading.first_vr;
        procedure_declaration.last_vr = procedure_heading.last_vr;
        create_edge (procedure_heading.last_heading_vr, block.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (block.first_vr, procedure_heading.last_heading_vr,
                {known, stmt_referable,}, rf_null);
        block.declaring_scope = procedure_heading.procedure_binding;
        ;
```

```
procedure_heading
        :       "PROCEDURE" id opt_formal_parms ";"
&SEMANTICS
        DECL_VR: IGVertex = create_vertex();
        procedure_declaration.first_vr = DECL_VR;
        procedure_declaration.last_vr = DECL_VR;

        NAME:Name = ident_to_name(id);
        procedure_heading.procedure_binding : Binding =
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = procedure_decl,
                        ... )),
        define_vis (DECL_VR, procedure_heading.procedure_binding,
                {known, stmt_referable, decl_referable})

        -- first visibility region local to procedure
        FIRST_LOCAL_VR: IGVertex = create_vertex();
        -- edge corresponding to scope entry
        create_edge (DECL_VR, FIRST_LOCAL_VR,
                {containing, stmt_referable, decl_referable}, rf_null);
        -- last_heading_vr is the last visibility region in the
        -- formal parameters section  (or FIRST_LOCAL_VR if no formal parms),
        -- from which the block in procedure_declaration inherits visibility
        procedure_heading.last_heading_vr =
                if opt_formal_parms.exists then
                        opt_formal_params.last_vr
                else
                        FIRST_LOCAL_VR
                elsif;
        if opt_formal_parms.exists then
                create_edge (FIRST_LOCAL_VR, opt_formal_parms.first_vr,
                        {known, containing, stmt_referable, decl_referable}, rf_null);
                create_edge (opt_formal_parms.first_vr, FIRST_LOCAL_VR,
                        {known, stmt_referable}, rf_null);
        endif;
        define_vis (FIRST_LOCAL_VR,
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = special_decl,,
                        ref_env = block.last_vr;
                        ... )),
                {known, stmt_referable, decl_referable})
        opt_formal_params.in_vr = FIRST_LOCAL_VR;
        redefine_declaring_scope (DECL_VR, procedure_heading.declaring_scope);
```

```
        opt_formal_parms.declaring_scope = procedure_heading.procedure_binding;
        ;

opt_formal_parms
        :       "(" formal_parm_seq ")" opt_return_type
&SEMANTICS
        opt_formal_parms.first_vr, opt_formal_parms.last_vr,
        formal_parm_seq.in_vr = opt_formal_parms.in_vr;
        |       -- empty
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        opt_formal_parms.first_vr, opt_formal_parms.last_vr = DUMMY_VR;
        ;

formal_parm_seq
        :       fp_section+{";"}
&SEMANTICS
        fp_section'FIRST.first_vr = formal_parm_seq.first_vr;
        create_edge (fp_section'I.last_vr, fp_section'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (fp_section'I+1.first_vr, fp_section'I.last_vr,
                {known, stmt_referable}, rf_null);
        formal_parm_seq.last_vr = fp_section'LAST.last_vr;

        fp_section'FIRST.in_vr = formal_parm_seq.in_vr;
        fp_section'I+1.in_vr = fp_section'I.last_vr;
        |       -- empty
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        formal_parm_seq.first_vr, formal_parm_seq.last_vr = DUMMY_VR;
        ;

fp_section
        :       opt_var decl_id_list ":" formal_type
&SEMANTICS
        fp_section.first_vr = decl_id_list.first_vr;
        fp_section.last_vr = decl_id_list.last_vr;
        formal_type.in_vr = fp_section.in_vr;
        -- descriptor for "formal_type" will be created by type checking semantics
        -- for "formal_type" and propagated down through tree where a descriptor
        -- for each variable can be created
        ;

opt_var
        :       ["VAR"]
        ;

module_declaration
        :       "MODULE" id opt_priority ";" import_seq opt_export block id ";"
&SEMANTICS
        DECL_VR: IGVertex = create_vertex();
```

```
module_declaration.first_vr = DECL_VR;
module_declaration.last_vr = DECL_VR;

NAME:Name = ident_to_name(id);
MODULE_BINDING: Binding =
        @BindingRecord(NAME,
                @EntityRecord(NAME,
                pervasive=false,
                readOnly=false,
                declaring_scope = INCLUDING declaring_scope,
                kind = module_decl,
-- unique_id used by redefine_vis to check entity equality
                unique_id = GetUniqueName(),
                ref_env = block.last_vr;
                ... )),
define_vis (DECL_VR, MODULE_BINDING,
        {known, stmt_referable, decl_referable})

-- first visibility region local to module
FIRST_LOCAL_VR: IGVertex = create_vertex();
-- define containing visibility
define_vis (FIRST_LOCAL_VR, MODULE_BINDING, {containing});
-- edge corresponding to scope entry
create_edge (DECL_VR, FIRST_LOCAL_VR,
        {stmt_referable, decl_referable}, rf_closed_scope);
create_edge (DECL_VR, FIRST_LOCAL_VR,
        {containing}, rf_null);
-- connect other visibility regions
create_edge (FIRST_LOCAL_VR, import_seq.first_vr,
        {known, containing, stmt_referable, decl_referable}, rf_null);
create_edge (import_seq.first_vr, FIRST_LOCAL_VR,
        {known, stmt_referable}, rf_null);
create_edge (import_seq.last_vr, block.first_vr,
        {known, containing, stmt_referable, decl_referable}, rf_null);
create_edge (block.first_vr, import_seq.last_vr,
        {known, stmt_referable}, rf_null);
block.in_vr = import_seq.last_vr;

-- define #enclosing_scope#
define_vis (FIRST_LOCAL_VR,
        @BindingRecord(enclosing_scope_name,
                @EntityRecord(enclosing_scope_name,
                pervasive=false,
                readOnly=false,
                declaring_scope = MODULE_BINDING,
                kind = special_decl,
                ref_env = module_declaration.in_vr ... )),
        {known, stmt_referable, decl_referable})
-- export can see all declarations, and we want to export both
-- stmt_referable and decl_referable visibility classes, and decl_referable is only
-- inherited forward, so the in_vr for the export is the last
```

```
        -- visibility region of the module
        opt_export.following_vr = module_declaration.following_vr;
        opt_export.in_vr = block.last_vr;
        redefine_declaring_scope (DECL_VR, module_declaration.declaring_scope);
        block.declaring_scope = MODULE_BINDING;
        ;

opt_export
        :       export
&SEMANTICS
        export.in_vr = opt_export.in_vr;
        export.following_vr = opt_export.following_vr;
        |       -- empty
        ;
```

The *pointer_type_decl* attribute is on *type*, *pointer_type*, *simple_type* and *qual_id* nodes, and is used to determine when a name reference is in a pointer-type declaration. If so, then forward references are legal, and *pointer_type_lookup* is used in qual_id to resolve the reference.

```
type
        :       simple_type
&SEMANTICS
        simple_type.in_vr = type.in_vr;
        type.type_def = simple_type.type_def;
        simple_type.pointer_type = type.pointer_type;
        |       array_type
        array_type.in_vr = type.in_vr;
        type.type_def = array_type.type_def;
        |       record_type
        record_type.in_vr = type.in_vr;
        type.type_def = record_type.type_def;
        |       set_type
        set_type.in_vr = type.in_vr;
        type.type_def = set_type.type_def;
        |       pointer_type
        pointer_type.in_vr = type.in_vr;
        type.type_def = pointer_type.type_def;
        |       procedure_type
        procedure_type.in_vr = type.in_vr;
        type.type_def = procedure_type.type_def;
        ;
```

The important synthesized attribute *type_def* is the entity describing the type in the type declaration. Where the process of finding/producing this entity involves referencing existing ones, this is shown. Other aspects of the process are not relevant to visibility control issues

```
simple_type
        :       qual_id
&SEMANTICS
        qual_id.in_vr = simple_type.in_vr;
        simple_type.type_def = qual_id.binding.entity;
```

```
        qual_id.pointer_type = simple_type.pointer_type;
        |       enumeration
&SEMANTICS
        enumeration.in_vr = simple_type.in_vr;
        -- type_def attribute for enumeration not relevant to visibility
        |       subrange_type
&SEMANTICS
        subrange_type.in_vr = simple_type.in_vr;
        -- type_def attribute for subrange_type not relevant to visibility
        ;

qual_id
        :       id+{"."}
&SEMANTICS
        IDENT_LIST: IdentList = id'CONCATENATE;
        -- convert identifiers in list to names
        NAME_LIST: NameList = ident_list_to_name_list (IDENT_LIST);
        -- resolve reference
        -- resolve_qualified_ref is as defined in Chapter 6
        qual_id.binding = resolve_qualified_ref (
                NAME_LIST, qual_id.in_vr, qual_id.pointer_type_decl);
        ;

enumeration
        :       "(" enum_id_list ")"
        ;

enum_id_list
        :       enum_id+{","}
        ;

enum_id
        : id
&SEMANTICS
        NAME:Name = ident_to_name(id);
        define_vis (INCLUDING enclosing_decl_vr,
                {known, stmt_referable, decl_referable})
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = enum_constant,
                        ... )),
                {known, stmt_referable, decl_referable})
        ;


subrange_type
        :       opt_qual_id "[" const_expr ".." const_expr "]"
&SEMANTICS
```

```
        opt_qual_id.in_vr = const_expr<1>.in_vr = const_expr<2>.in_vr
                = subrange_type.in_vr;
        ;

opt_qual_id
        :       qual_id
        -- lookup of qual_id done in qual_id production: what is done with
        -- binding found is not important to us here
&SEMANTICS
        qual_id.in_vr = opt_qual_id.in_vr;
        |       -- empty
        ;

array_type
        :       "ARRAY" simple_type_list "OF" type
&SEMANTICS
        simple_type_list.in_vr = type.in_vr = array_type.in_vr;
        type.pointer_type_decl = false;
        -- type_def attribute for array_type not relevant to visibility
        ;

simple_type_list
        :       simple_type+{","}
&SEMANTICS
        simple_type.in_vr = simple_type_list.in_vr;
        ;

record_type
        :       "RECORD" field_list "END"
&SEMANTICS
        -- first visibility region local to record
        FIRST_LOCAL_VR: IGVertex = create_vertex();
        -- edge corresponding to scope entry
        create_edge (record_type.in_vr, FIRST_LOCAL_VR,
                {containing, stmt_referable, decl_referable}, rf_null);
        field_list.in_vr = FIRST_LOCAL_VR;
        record_type.type_def =
                @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = record_type,
                        ref_env = field_list.last_vr;
                ... ),
        ;
```

The production for *field_list* assumes that each *opt_field* produces at least one visibility region. To make this assumption valid, *DUMMY_VR* is created in the production for *opt_field* if *opt_field* $\rightarrow \varepsilon$. The assumption is needed because a lack of sufficient notation to link together only the non-empty *opt_fields*. The result is the production of some extra inheritance graph vertices, so some visibility regions may correspond to more than one vertex. However, the extra

vertices don't change the semantics of any references.

```
field_list
        :       opt_field+{";"}
&SEMANTICS
        opt_field'FIRST.in_vr = field_list.in_vr;
        -- link vis regions created by each opt_field
        create_edge (opt_field'I.last_vr, opt_field'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (opt_field'I+1.first_vr, opt_field'I.last_vr,
                {known, stmt_referable}, rf_null);
        field_list.first_vr = opt_field'LAST.first_vr;
        field_list.last_vr = opt_field'LAST.last_vr;
        ;

opt_field
        :       field
&SEMANTICS
        field.in_vr = opt_field.in_vr;
        opt_field.first_vr = field.first_vr;
        opt_field.last_vr = field.last_vr;
        |       -- empty
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        opt_field.first_vr, opt_field.last_vr = DUMMY_VR;
        ;

field
        :       decl_id_list ":" type
&SEMANTICS
        field.first_vr = decl_id_list.first_vr;
        field.last_vr = decl_id_list.last_vr;
        type.in_vr = field.in_vr;
        type.pointer_type_decl = false;

        decl_id_list.type_def = type.type_def;
        decl_id_list.kind = var_decl;
        |       "CASE" ":" qual_id "OF" variant_list opt_else_fields "END"
&SEMANTICS
        qual_id.in_vr = field.in_vr;
        variant_list.in_vr = field.in_vr;
        opt_else_fields.in_vr = field.in_vr;

        -- join together visibility regions
        field.first_vr = variant_list.first_vr;
        if opt_else_fields.exists then
                field.last_vr = opt_else_fields.last_vr;
                create_edge (variant_list.last_vr, opt_else_fields.first_vr,
                        {known, containing, stmt_referable, decl_referable}, rf_null);
                create_edge (opt_else_fields.first_vr, variant_list.last_vr,
                        {known, stmt_referable}, rf_null);
        else
```

```
                field.last_vr = variant_list.last_vr;
        endif;
        |       "CASE" id ":" qual_id "OF" variant_list opt_else_fields "END"
&SEMANTICS
        qual_id.in_vr = field.in_vr;
        variant_list.in_vr = field.in_vr;
        opt_else_fields.in_vr = field.in_vr;

        LOCAL_VR: IGVertex = create_vertex();
        NAME:Name = ident_to_name(id);
        define_vis (LOCAL_VR,
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
                        pervasive=false,
                        readOnly=false,
                        declaring_scope = INCLUDING declaring_scope,
                        kind = field_decl,
                        ... )),
                {known, stmt_referable, decl_referable})

        -- join together visibility regions
        field.first_vr = LOCAL_VR;
        create_edge (LOCAL_VR.last_vr, variant_list.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (variant_list.first_vr, LOCAL_VR.last_vr,
                {known, stmt_referable}, rf_null);
        if opt_else_fields.exists then
                field.last_vr = opt_else_fields.last_vr;
                create_edge (variant_list.last_vr, opt_else_fields.first_vr,
                        {known, containing, stmt_referable, decl_referable}, rf_null);
                create_edge (opt_else_fields.first_vr, variant_list.last_vr,
                        {known, stmt_referable}, rf_null);
        else
                field.last_vr = variant_list.last_vr;
        endif;
        ;

variant_list
        :       opt_variant+{"|"}
&SEMANTICS
        opt_variant'FIRST.in_vr = variant_list.in_vr;
        -- link vis regions created by each opt_variant
        create_edge (opt_variant'I.last_vr, opt_variant'I+1.first_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (opt_variant'I+1.first_vr, opt_variant'I.last_vr,
                {known, stmt_referable}, rf_null);
        variant_list.first_vr = opt_variant'LAST.first_vr;
        variant_list.last_vr = opt_variant'LAST.last_vr;
        ;

opt_variant
```

```
        :       case_label_list ":" field_list
&SEMANTICS
        field_list.in_vr = opt_variant.in_vr;
        case_label_list.in_vr = opt_variant.in_vr;
        opt_variant.first_vr = field_list.first_vr;
        opt_variant.last_vr = field_list.last_vr;
        |       -- empty
&SEMANTICS
        opt_variant.first_vr, opt_variant.last_vr = DUMMY_VR;
        ;

case_label_list
        :       case_label+{","}
&SEMANTICS
        case_label'ALL.in_vr = case_label_list.in_vr;
        ;

case_label
        :       const_expr
&SEMANTICS
        const_expr.in_vr = case_label.in_vr;
        |       const_expr ".." const_expr
&SEMANTICS
        const_expr<1>.in_vr = case_label.in_vr;
        const_expr<2>.in_vr = case_label.in_vr;
        ;

opt_else_fields
        :       "ELSE" field_list
&SEMANTICS
        field_list.in_vr = opt_else_fields.in_vr;
        opt_else_fields.first_vr = field_list.first_vr;
        opt_else_fields.last_vr = field_list.last_vr;
        opt_else_fields.exists = true;
        |       -- empty
-- This illustrates the alternative to creating a dummy visibility region
-- for an empty list.  It is more complicated, because it requires an
-- extra attribute, and a check whether the field is empty or not.
&SEMANTICS
        opt_else_fields.first_vr, opt_else_fields.last_vr = null;
        opt_else_fields.exists = false;
        ;

set_type
        :       "SET" "OF" simple_type
&SEMANTICS
        simple_type.in_vr = set_type.in_vr;
        simple_type.pointer_type_decl = false;
        ;

pointer_type
```

```
        :       "POINTER" "TO" type
&SEMANTICS
        type.in_vr = pointer_type.in_vr;
        type.pointer_type_decl = true;
        ;

procedure_type
        :       "PROCEDURE" opt_formal_types
&SEMANTICS
        opt_formal_types.in_vr = procedure_type.in_vr;
        ;

opt_formal_types
        :       ["(" formal_var_type_seq ")" opt_return_type]
&SEMANTICS
        formal_var_type_seq.in_vr = opt_formal_types.in_vr;
        opt_return_type.in_vr = opt_formal_types.in_vr;
        ;

formal_var_type_seq
        :       formal_var_type*{";"}
&SEMANTICS
        formal_var_type'ALL.in_vr = formal_var_type_seq.in_vr;
        ;

formal_var_type
        :       formal_type
&SEMANTICS
        formal_type.in_vr = formal_var_type.in_vr;
        |       "VAR" formal_type
&SEMANTICS
        formal_type.in_vr = formal_var_type.in_vr;
        ;

formal_type
        :       qual_id
&SEMANTICS
        qual_id.in_vr = formal_type.in_vr;
        |       "ARRAY" "OF" qual_id
&SEMANTICS
        qual_id.in_vr = formal_type.in_vr;
        ;

opt_return_type
        :       [":" qual_id]
&SEMANTICS
        qual_id.in_vr = opt_return_type.in_vr;
        ;

expr
        :       expr "=" expr
```

```
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "#" expr             => ne(expr<1>, "#", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "<" expr             => lt(expr<1>, "<", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "<=" expr            => le(expr<1>, "<=", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr ">" expr             => gt(expr<1>, ">", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr ">=" expr            => ge(expr<1>, ">=", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "<>" expr            => ne(expr<1>, "<>", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "IN" expr            => in(expr<1>, "IN", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       "+" expr                  => u_plus("+", expr)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        |       "-" expr          => u_minus("-", expr)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        |       expr "+" expr             => plus(expr<1>, "+", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "-" expr             => minus(expr<1>, "-", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "OR" expr            => or(expr<1>, "OR", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "*" expr             => star(expr<1>, "*", expr<2>)
&SEMANTICS
```

```
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "/" expr              => slash(expr<1>, "/", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "DIV" expr                  => div(expr<1>, "DIV", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "MOD" expr                  => mod(expr<1>, "MOD", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       expr "AND" expr                  => and(expr<1>, "AND", expr<2>)
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        expr<3>.in_vr = expr<1>.in_vr;
        |       integer
        |       real
        |       string
        |       set
&SEMANTICS
        set.in_vr = expr.in_vr;
        |       designator
&SEMANTICS
        designator.in_vr = expr.in_vr;
        |       designator actual_parms
&SEMANTICS
        designator.in_vr = expr.in_vr;
        actual_parms.in_vr = expr.in_vr;
        |       "NOT" expr
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        |       "(" expr ")"
&SEMANTICS
        expr<2>.in_vr = expr<1>.in_vr;
        ;

set
        :       designator "{" element_list "}"
&SEMANTICS
        designator.in_vr = set.in_vr;
        element_list.in_vr = set.in_vr;
        |       "{" element_list "}"
&SEMANTICS
        element_list.in_vr = set.in_vr;
        ;

element_list
        :       element+{","}
```

```
&SEMANTICS
        element'ALL.in_vr = element_list.in_vr;
        ;

element
        :       expr
&SEMANTICS
        expr.in_vr = element.in_vr;
        |       expr ".." expr
&SEMANTICS
        expr<1>.in_vr = element.in_vr;
        expr<2>.in_vr = element.in_vr;
        ;
```

A ''designator'' as defined by Modula-2 is a reference in an expression, perhaps qualified, and perhaps containing array indexing and pointer indirection. The steps in resolving a qualified reference are much simpler than a reference in a declaration context, because all declarations in enclosing scopes are visible, not just those preceding the reference. Thus, after the initial lookup, successive lookups are done in the ref_env of the preceding binding, which is the last visibility region local to the scope corresponding to that binding.

Finding the proper entities is only part of the process of determining the access path to a particular storage location. Including the effects of array indexing and pointer indirection is also necessary, but not part of the visibility control problem, as defined in this dissertation.

```
designator
        :       id des_seq
&SEMANTICS
        des_seq.in_vr = designator.in_vr;
        des_seq.prev_binding =
                statement_lookup (ident_to_name(id),designator.in_vr);
        designator.binding = des_seq.binding;
        ;

des_seq
        :       des des_seq
&SEMANTICS
        des.in_vr, des_seq<2>.in_vr = des_seq<1>.in_vr;
        des.prev_binding = des_seq<1>.prev_binding;
        des_seq<1>.binding = des_seq<2>.binding;
        |       -- empty
        des_seq.binding = des_seq.prev_binding;
        ;

des
        :       "." id
&SEMANTICS
        des.binding =
                if qualifiable_kind (des.prev_binding.entity.kind) then
                        statement_lookup (ident_to_name(id),
                                des.prev_binding.entity.ref_env);
                else
                        error_binding;
```

```
                endif;
        if (des.prev_binding.entity.kind = module_decl) &&
                (not des.binding.exported) then
         error ("Attempt to reference non-exported declaration of module by qualification")
        endif
        |       "[" expr_list "]"
&SEMANTICS
        expr_list.in_vr = des.in_vr;
        |       "^"
        ;

actual_parms
        :       "(" opt_expr_list ")"
&SEMANTICS
        opt_expr_list.in_vr = actual_parms.in_vr;
        ;

opt_expr_list
        :       expr_list
&SEMANTICS
        expr_list.in_vr = opt_expr_list.in_vr;
        |       -- empty
        ;

expr_list
        :       expr+{","}
&SEMANTICS
        expr'ALL.in_vr = expr_list.in_vr;
        ;

block
        :       decl_seq "END"
&SEMANTICS
        decl_seq.in_vr = block.in_vr;
        block.first_vr = decl_seq.first_vr;
        block.last_vr = decl_seq.last_vr;
        |       decl_seq "BEGIN" stmt_list "END"
&SEMANTICS
        decl_seq.in_vr = block.in_vr;
        block.first_vr = decl_seq.first_vr;
        stmt_list.in_vr =  block.last_vr = decl_seq.last_vr;
        ;

decl_seq
        :       declaration+
&SEMANTICS
        declaration'FIRST.in_vr = decl_seq.in_vr;
        create_edge (declaration'I.last_vr, declaration'I+1.last_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (declaration'I+1.last_vr, declaration'I.last_vr,
                {known, stmt_referable}, rf_null);
```

```
        declaration'I.following_vr = declaration'I+1.first_vr;
        -- add an extra vr at the end of the declaration list so that there
        -- is always a vr following a module_declaration (for exports)
        FOLLOWING_VR : IGVertex = create_vertex();
        declaration'LAST.following_vr = FOLLOWING_VR;

        create_edge (declaration'LAST.last_vr, following_vr,
                {known, containing, stmt_referable, decl_referable}, rf_null);
        create_edge (following_vr, declaration'LAST.last_vr,
                {known, stmt_referable}, rf_null);
        decl_seq.last_vr = following_vr;
        declaration'ALL.declaring_scope = INCLUDING declaring_scope;
        |       -- empty
&SEMANTICS
        DUMMY_VR: IGVertex = create_vertex();
        decl_seq.first_vr, decl_seq.last_vr = DUMMY_VR;
        ;


-- statements cannot affect the referencing environment of their enclosing
-- context, so no first_vr or last_vr is needed.
stmt_list
        :       opt_stmt+{";"}
&SEMANTICS
        opt_stmt'ALL.in_vr = stmt_list.in_vr;
        ;

opt_stmt
        :       stmt
&SEMANTICS
        stmt.opt_stmt.in_vr = opt_stmt.in_vr;
        |       -- empty
        ;

stmt
        :       assignment
&SEMANTICS
        assignment.in_vr = stmt.in_vr;
        |       proc_call
&SEMANTICS
        proc_call.in_vr = stmt.in_vr;
        |       if_stmt
&SEMANTICS
        if_stmt.in_vr = stmt.in_vr;
        |       case_stmt
&SEMANTICS
        case_stmt.in_vr = stmt.in_vr;
        |       while_stmt
&SEMANTICS
        while_stmt.in_vr = stmt.in_vr;
        |       repeat_stmt
```

```
&SEMANTICS
        repeat_stmt.in_vr = stmt.in_vr;
        |       loop_stmt
&SEMANTICS
        loop_stmt.in_vr = stmt.in_vr;
        |       for_stmt
&SEMANTICS
        for_stmt.in_vr = stmt.in_vr;
        |       with_stmt
&SEMANTICS
        with_stmt.in_vr = stmt.in_vr;
        |       "EXIT"
        |       "RETURN"
        |       "RETURN" expr
&SEMANTICS
        expr.in_vr = stmt.in_vr;
        ;

assignment
        :       designator ":=" expr
&SEMANTICS
        designator.in_vr, expr.in_vr = assignment.in_vr;
        ;

proc_call
        :       designator
&SEMANTICS
        designator.in_vr = proc_call.in_vr;
        |       designator actual_parms
&SEMANTICS
        designator.in_vr = actual_parms.in_vr = proc_call.in_vr;
        ;

if_stmt
        :       "IF" expr "THEN" stmt_list elsif_seq opt_else "END"
&SEMANTICS
        expr.in_vr = stmt_list.in_vr = elsif_seq.in_vr = opt_else.in_vr = if_stmt.in_vr;
        ;

elsif_seq
        :       elsif*
&SEMANTICS
        elsif'ALL.in_vr = elsif_seq.in_vr;
        ;

elsif
        :       "ELSIF" expr "THEN" stmt_list
&SEMANTICS
        expr.in_vr = stmt_list.in_vr = elsif.in_vr;
        ;
```

```
opt_else
        :       "ELSE" stmt_list
&SEMANTICS
        stmt_list.in_vr = opt_else.in_vr;
        |       -- empty
        ;

case_stmt
        :       "CASE" expr "OF" case_list opt_else "END"
&SEMANTICS
        expr.in_vr = case_list.in_vr = opt_else.in_vr = case_stmt.in_vr;
        ;

case_list
        :       opt_case+{"|"}
&SEMANTICS
        opt_case'ALL.in_vr = case_list.in_vr;
        ;

opt_case
        :       [case_label_list ":" stmt_list]
&SEMANTICS
        case_label_list.in_vr = stmt_list.in_vr = opt_case.in_vr;
        ;

while_stmt
        :       "WHILE" expr "DO" stmt_list "END"
&SEMANTICS
        expr.in_vr = stmt_list.in_vr = while_stmt.in_vr;
        ;

repeat_stmt
        :       "REPEAT" stmt_list "UNTIL" expr
&SEMANTICS
        expr.in_vr = stmt_list.in_vr = repeat_stmt.in_vr;
        ;
```

A **for** statement creates a new scope which declares the control variable, and inherits visibility from the last visibility region preceding the **for** .

```
for_stmt
        :       "FOR" id ":=" expr "TO" expr opt_by "DO" stmt_list "END"
&SEMANTICS
        expr<1>.in_vr = for_stmt.in_vr;
        -- create binding of control variable local for for statement
        LOCAL_VR: IGVertex = create_vertex();
        create_edge (for_stmt.in_vr, LOCAL_VR,
                {containing, stmt_referable, decl_referable}, rf_null);
        NAME:Name = ident_to_name(id);
        define_vis (LOCAL_VR,
                @BindingRecord(NAME,
                        @EntityRecord(NAME,
```

```
                    pervasive=false,
                    readOnly=false,
                    declaring_scope = null,
                    kind = control_var_decl,
                    ... )),
                {known, stmt_referable, decl_referable})

        expr<2>.in_vr = opt_by.in_vr = stmt_list.in_vr = LOCAL_VR;
        ;

opt_by
        :       "BY" const_expr
&SEMANTICS
        const_expr.in_vr = opt_by.in_vr;
        ;

loop_stmt
        :       "LOOP" stmt_list "END"
&SEMANTICS
        stmt_list.in_vr = loop_stmt.in_vr;
        ;
```

The **with** statement in Modula-2 creates a new open scope in which all bindings local to the opened record are directly visible. This is achieved by creating a new scope (a new vertex inheriting all but ***known*** visibility) and adding edges from the last visibility region local to the record being opened. Inheritance is restricted to the fields local to that record. ***known*** visibility is inherited, so any binding inherited from the opened record will shadow any clashing binding visible outside the **with** statement.

```
with_stmt
        :       "WITH" designator "DO" stmt_list "END"
&SEMANTICS
        LOCAL_VR: IGVertex = create_vertex();
        create_edge (for_stmt.in_vr, LOCAL_VR,
                {containing, stmt_referable, decl_referable}, rf_null);
        -- new scope inherits all visibility of bindings local to
        -- named record
        create_edge (designator.binding.ref_env, LOCAL_VR,
                {known, stmt_referable, decl_referable},
                rf_local_decls(designator.binding));
        stmt_list.in_vr = LOCAL_VR;

        ;

const_expr
        :       expr
&SEMANTICS
        expr.in_vr = const_expr.in_vr;
        ;
```

7.7. Modula-2 Description: Summary and Conclusions

The Attribute Grammar Descriptive Method

The experience of writing the inheritance graph description was mixed, giving both positive and negative results. The description is quite long, and writing it was tedious at times. However, I feel that both of these problems are primarily due to the attribute grammar-like descriptive method used. This is a common problem with attribute grammars, because transferring information from one part of the tree to another may involve many intermediate steps, each copying the information along from one node to the next. Default copy rules are often used to shorten the description, but they were intentionally avoided in order to avoid confusion. Default copy rules help eliminate semantic functions that carry little meaning, thus making the important parts of the description stand out. However, they make the absence of an important semantic function less obvious – an attribute grammar evaluator generator will happily fill in a copy rule in its place, which can result in a hard-to-detect bug.

Using default copy rules would shorten the description significantly. Many of the copy rules are needed just to propagate the proper visibility region for references down through expressions. However, the description would still be quite long. Part of this is still be due to the attribute grammar formalism. Another method of shortening the description and making it more readable would be the use of Ganzinger's technique of parameterizing compiler modules by splitting a language description into several sub-grammars [Ganzinger 1981]. Each sub-grammar contains only the semantics *and* syntax for a part of the language (*e.g.*, visibility control and related parts of the language), and a bijection exists between each sub-grammar and the full grammar. Most copy rules can be eliminated from the description because productions irrelevant to the purpose of the sub-grammar can be abstracted out. Also, the semantic functions for each sub-grammar are concerned only with one area of language semantics, avoiding the problem of intermixing different parts of the semantics of a language in each production of an attribute grammar.

Another approach to modularizing attribute grammars is described by Watt [Watt 1985]. Each part of the semantics of a language is described using a separate attribute grammar, but unlike Ganzinger's approach, each partition operates on the same attribute grammar. Also, the attribute grammar is based on the abstract syntax of the language instead of the concrete syntax. A special **propagate** operator is used to eliminate the many copy rules otherwise required by the use of the complete grammar.

Some descriptive method other than attribute grammars might be better for defining the translation from programs to inheritance graphs. A graph-based formalism would be a good candidate.

Complexity of the Description

The inheritance graph description for Modula-2 is more complex than the descriptions of individual visibility constructs given in Chapter 6. Four visibility classes were required for the description. This is mostly because the visibility control rules for Modula-2's language constructs are more complex than for the other contructs considered in the examples[43]. The first problem is the use of different visibility rules depending on whether a reference occurs in a statement or in a declaration. This necessitates the use of two separate visibility classes, ***stmt_referable*** and ***decl_referable***, for "normal" references. This is a good example of how *not* to let implementation considerations affect language design.

[43] An exception is the Ada **use** statement.

The second problem arises from the interaction of declaration-before-use and nested qualified references. Nested qualified references are very useful, and disallowing them while allowing other qualified references would tend to destroy the orthogonality of a language design. When nested qualified references are mixed with a declaration-before-use requirement, however, the semantics of the reference become much more complex, as previously illustrated. and in the Modula-2 description in this chapter. The problems of declaration-before-use will be discussed in more depth in Chapter 9.

Most of the description of the visibility control rules was not very difficult. Even features which are traditionally problematic, such as distinguishing properly between a (legal or illegal) forward reference and a reference to a binding in an enclosing scope, were easy to describe with the use of multiple visibility types. Many compilers of both Pascal and Modula-2 do not handle this correctly.

The only difficult feature to describe was nested qualified references, which are unarguably complex, so any description of them is likely to be complex. The greatest problem in writing the description was the tedium of going from a basic design of the inheritance graph for Modula-2 to the complete description. This was mostly due to the length of the attribute grammar description, as discussed in the previous section.

Ambiguities in Modula-2

Describing the inheritance graph for Modula-2 was very useful in aiding understanding of its visibility control rules, and in discovering gaps in the definition of the rules [Wirth 1982]. The process of developing an appropriate schema for each construct forced a clear understanding of the construct, while permitting a local description of the construct. Most of the ambiguities described in §7.1 in this chapter were discovered in this process.

CHAPTER 8

Evaluating the Inheritance Graph

Chapter 6 defined the meaning of the inheritance graph in terms of a least fixed point assignment to a graph, where the value at each vertex was the visible set for that vertex. Another view considers the inheritance graph as a constraint system, with the restriction, definition/redefinition, and clash resolution functions comprising the constraints.

All that is required is enough information to evaluate the *visible* primitive at any vertex, and to check for errors defined in the Clash Table. Computing the visible set for every vertex makes *visible* very simple, but it is possible to place more of the burden of definition on *visible*, as described in §5.3, where the What Visible and the Where Visible approaches to defining the visibility control rules of languages are discussed. In the Inheritance Graph Model of visibility control, this distinction has been abstracted out, but the distinction is still useful at the implementation level.

None of the definitions of the meaning of an inheritance graph provide us with a satisfactory method of finding the least fixed point assignment to the graph, or discovering if there is no fixed point assignment (an erroneous inheritance graph) or more than one fixed point, of which none is least (an ambiguous inheritance graph). This chapter explores several approaches to assuring that *visible* can be computed. The chapter presents methods for finding a least fixed point assignment, and describes sufficient conditions on an inheritance graph description to assure that any fixed point found is the least fixed point. As defined, an inheritance graph for which no valid assignment exists usually represents an incorrect program, although an inheritance graph for which the assignment oscillates usually indicates a poorly defined inheritance graph description.

An evaluation method based on the first model of visibility control presented, the Search Model, is presented first. More powerful and general techniques will follow.

The set of all bindings defined in the inheritance graph is available, and is denoted by *Bindings*. This set is easily maintained as a side-effect of *define_vis* and *redefine_vis* calls.

8.1. Search Model Evaluation of the Inheritance Graph

The Search Model has been used both to describe (as in [Wirth 1982]) and to implement (as in [Blower 1984]) the visibility control rules of languages. It is almost certainly the *mental* model used by some programmers to understand the use and availability of names in their programs. It seems obligatory to consider use of the Search Model as an approach to defining the meaning of *visible*.

The basics of the Search Model are as defined in Chapter 2. The search is initiated by a call to *visible*, with a vertex, a name, and a visibility class as arguments. A reference is resolved by searching from the point of the reference (the vertex) to find a definition matching the reference. In the inheritance graph, the search proceeds in the direction opposite that of the inheritance edges, following edges of the visibility class given in the call to *visible*. The restriction function on each edge traversed is used to augment the search predicate for the reference, as described in §5.2. The search terminates (for some languages) when a *define_vis* or *redefine_vis* operation is encountered that defines a matching binding with the visibility class specified in the call to *visible*.

Several characteristics of the inheritance graph complicate the search:

(1) The inheritance graph may contain cycles. The search must avoid looping, performing a depth-first or breadth-first search (for example). Usually, the search strategy doesn't matter, although in the old version of Flavors [Weinreb and Moon 1981], the search order is important. Due to the nature of inheritance, there is no advantage to following cycles.

(2) Branches may occur in the search. If overloading is allowed, all visible bindings of the referenced name must be found so that the overloading resolution algorithm may be applied. If no overloading is allowed, searching of all branches may be necessary to find any clashing bindings that may signify an error.

(3) Clash resolution must occur at each node. Suppose that a binding/visibility-class pair $<b, vc_i>$ is defined at vertex v_{b_def}, is referenced at v_{b_ref}, and is visible (inherited) at v_{b_ref} along a path p [44] from v_{b_def} to v_{b_ref} in the absence of any clash resolution. That is, the above-described search finds the definition of b, and b satisfies the search predicate accumulated by the search. According to the definition of the inheritance graph, if another pair $<b', vc_j>$ is inherited at some vertex $v_{b'}$ on path p and *clash*(b,b'), then clash resolution must take place, which may shadow $<b, vc_i>$ or $<b', vc_j>$ or signal an error.

(4) The procedure associated with the creation of dynamic definitions (§6.6.9) can depend on the set of all bindings visible at one or more vertices. One approach to computing this set of bindings is to perform a search lookup for every binding in *Bindings* at each vertex the procedure depends on, but this computation requires $O(|Bindings|)$ searches for every vertex that a procedure creating dynamic definitions depends on. Alternatively, a search could be implemented that finds *all* inherited bindings, maintaining the search predicate during the search, and adding each binding satisfying the search predicate for which a *define_vis* or *redefine_vis* operation is encountered. The length of this search is the same as a search for a single binding, assuming all branches must always be searched. The issues of clash resolution and error detection described in (3) must be resolved for this search also.

The clash resolution and error detection problems described in (3) are the most difficult ones.

Clash Resolution with the Search Model

For a call *visible*(*v*:*IGVertex*, *name*:*Name*, vc_i:*VisibilityType*), clash resolution can be handled by finding all binding/visibility-class pairs $<b, vc_j>$ on the search path (subject to the search predicate) whether or not $i=j$. The search process describes a search tree through the inheritance graph. After visiting all children of a vertex v' in this tree, the search process has collected the set $vs_{v',name}$ of all binding/visibility-class pairs inherited by v' subject to the search predicate and matching *name*. The clash resolution process is invoked on the set $vs_{v',name}$, possibly resulting in errors, possibly shadowing some elements of $vs_{v',name}$ resulting in a new set of binding/visibility-class pairs which are inherited at the parent of v' in the search tree.

In an inheritance graph with no restrictions on the clash table, the result computed by this bottom-up clash resolution process depends on the fact that cycles are eliminated by breaking the search when a cycle is detected (when a vertex on the path from the root to the current vertex is encountered again). If the rule required instead that each cycle be traversed once, then a different result could be computed. In the example in §6.8 (Figure 6.19), this is true: for a reference at v_2, the search order v_0,v_2,v_3,v_1 results in a clash in the bottom-up computation at v_2, resulting in neither b_1 nor b_2 being visible at v_2. The search order v_0,v_2,v_3,v_2,v_1 results in a clash in the bottom-up computation at v_3, shadowing b_1 and b_2 at v_3. At the next (and last) step in the bottom-up traversal (at v_2) b_1 alone is visible and is the result of the search. This ambiguity

[44] There may be another path from v_{b_def}, but it can be considered separately.

resulting from the precise definition of the search tree is the product of an inherently ambiguous inheritance graph.

If there are no dynamic definitions, and the inheritance graph is ordered (§6.8.1), then the search and clash resolution process can be proven correct and unambiguous by induction on the search tree: *i.e.*, it produces the correct set of binding/visibility-class pairs at vertex v. The traversal of cycles cannot affect the result of the search. A full proof is omitted, but a sketch of the important points follows:

- All paths of inheritance are represented in the search tree, except those containing cycles.
- Traversing a cycle cannot affect the result of a search, because any binding/visibility-class pair found by traversing a cycle must by necessity have been found by a shorter path in the search tree. The order in which bindings are found does not matter, because shadowing is governed by a strict order – the clashing binding/visibility-class pair with the lowest-numbered visibility class will not be shadowed regardless of where on the search path it is discovered.
- The induction involves computing the visible bindings of the referenced name at the leaves, and then for successive levels higher in the search tree.

Error Handling with the Search Model

Detecting all errors is more difficult. An error resulting from a clash table entry is independent of any references. This is one of the greatest weaknesses of the Search Model. According to the definition of the inheritance graph, a pair of clashing binding/visibility-class pairs visible at a vertex must be subjected to the clash resolution process specified by the clash table, with one possible result being an error message. The brute force approach is to find all bindings visible at each vertex by the above described search, followed by clash resolution to discover errors. This is clearly unacceptably inefficient. Some optimizations of this process are possible, but I have been unable to develop an acceptable method for finding all errors in an otherwise unrestricted ordered inheritance graph using the Search Model.

If some restrictions are placed on how the inheritance graph designer writes the description and checks for errors, efficient error checking is possible. Illegal clashing bindings are the primary source of errors. Suppose $<b, vc_i>$ is defined at vertex v. If the description is written such that all bindings $<b', vc_j>$ satisfying:

(1) $clash(b, b') = \textbf{true}$, and

(2) The result of clash resolution of $<b, vc_i>$ and $<b', vc_j>$ is a clashing binding error,

are visible at v, then the check for illegal clashing bindings can be attached to v or to the definition of b itself, eliminating the error entry from the clash table. For the sequence of declarations local to a scope, as in Pascal and Modula-2, the condition for this transformation is easily satisfied.

However, the inheritance graph resulting from the presence of import and export statements in Modula-2, as described in Chapter 7, does not satisfy this condition. Two bindings b_1 and b_2 of the same name declared in different modules and explicitly imported into the same module M violate the condition: both b_1 and b_2 are ***known*** in M, which is an error, but neither b_1 nor b_2 is ***known*** at the definition vertex of the other (in the absence of other import statements). This and other similar situations can handled by attaching a check for ***known*** clashing bindings to the vertex resulting from the import statement. This case is not difficult, but more general cases are difficult to handle. If a scope-opening style import makes the bindings from the imported module ***known*** in the importing module, then such a check is necessary for all of the bindings inherited due to the import, but the set of bindings made visible by the import is not easily determined at the point of the import statement. This situation arises at any vertex that is at the confluence of two or more inheritance edges with visibility classes that can result in clashing-binding errors. In

the worst case, it is necessary to locate all bindings visible with one of the incoming visibility classes at the vertex.

The only other kind of error in the clash table I have found reason to use is a "description error". The description error entries in the clash table are intended as redundant checks, and if the inheritance graph description is as intended, will never be invoked. If one is willing to eliminate these redundant checks, and write the inheritance graph description such that checks for illegal clashing declarations can be attached to vertices where definitions occur, the error-checking function of the clash table, and thus the problem of checking for errors at every vertex, can be eliminated.

Handling of Priorities Using the Search Model

The possible extension to the Inheritance Graph Model allowing prioritized clash resolution can be handled easily in the Search Model. When doing bottom-up clash resolution, if two clashing bindings are inherited, and if the clash table entry for the two bindings and their associated visibility classes is **prioritize**, then the binding inherited via the edge with the lowest numbered priority value is selected (if there is only one such binding), and all other clashing bindings are shadowed.

Summary of Search Model Evaluation Method

For fairly simple inheritance graphs, the Search Model can be used without much difficulty. Error-detection is the hardest problem, with only a very inefficient general solution (known to us at this point, at least). By placing most of the burden of error detection on the inheritance graph designer, requiring placement of error checks at appropriate vertices, this inefficiency can be avoided for a large class of languages. For some language features, error checking seems to be unavoidably inefficient using the Search Model.

The problem of error detection seems to be an example of an inherent weakness of the Search Model: handling some visibility constructs requires information from a large area of a program. Collecting this information separately for each vertex where it is needed is inevitably inefficient. The Visible Set Model avoids this problem, because the information at each vertex is defined in terms of information at adjacent vertices, and information can be collected for the entire graph at once. The Search Model is useful in cases where information flow is simpler. Its advantages are its simplicity and ease of understanding (for simple visibility rules) and its low storage requirements.

8.2. Evaluation Using Data Flow Analysis

Computing the set of visible binding/visibility-class pairs at each vertex in an ordered inheritance graph turns out to map very easily to a standard data flow problem: that of finding reaching definitions [Hecht 1977]. This is a set union problem for which bit vectors can be used. A bit vector in this problem represents a subset of *Bindings*. Standard data flow analysis (DFA) techniques can be used to find an assignment, and DFA results can be used to guarantee a least fixed-point assignment.

8.2.1. Formulation of Data Flow Problem

An ordered inheritance graph IG can be "partitioned" into separate graphs $IG_1, \cdots, IG_n$, where n is the number of visibility classes used in the inheritance graph description, and IG_j has only visibility class vc_j edges. Each IG_j has the same vertex set as IG. Each IG_j forms a separate data flow problem to be solved, and can depend on the solutions to IG_i, $1 \le i < j$. The uniqueness results for DFA do not hold if the inheritance graph is not ordered, because the basis for the flow problem (KILL and DEF) is then not constant. This problem will be described more

formally later.

The mapping of a graph IG_i into a graph suitable for DFA[45] is as follows:

(1) Each vertex in IG_i is split into three vertices, corresponding to the three stages of visibility defined within an inheritance graph vertex (as explained in §6.2.1): inherited visibility, intermediate visibility, and net visibility. The three vertices correspond to the three rectangles on the left side of Figure 8.1.

(2) Each edge of this graph is labeled with the function(s) affecting the inheritance of visibility via that edge. An edge corresponding to a normal inheritance edge in the inheritance graph is labeled with the restriction function for that edge. The edge connecting the "inherited visibility" and "intermediate visibility" vertices is labeled with the *define_vis* and *redefine_vis* calls associated with the original inheritance graph vertex. Finally, the edge connecting the "intermediate visibility" and "net visibility" vertices is labeled with the clash resolution function, *clash-resolve*.

(3) The resulting graph is transformed into the corresponding *edge graph* [Bondy and Murty 1976]. In simple, imprecise terms, there is a vertex in the edge graph corresponding to each edge in the original graph, and there is an edge in the edge graph corresponding to each pair of edges adjacent to the same vertex in the original graph. More precisely, there is a vertex v_i in the edge graph corresponding to each edge e_i in the original graph, and two vertices v_1 and v_2 in the edge graph are connected by a directed edge (v_1,v_2) if and only if

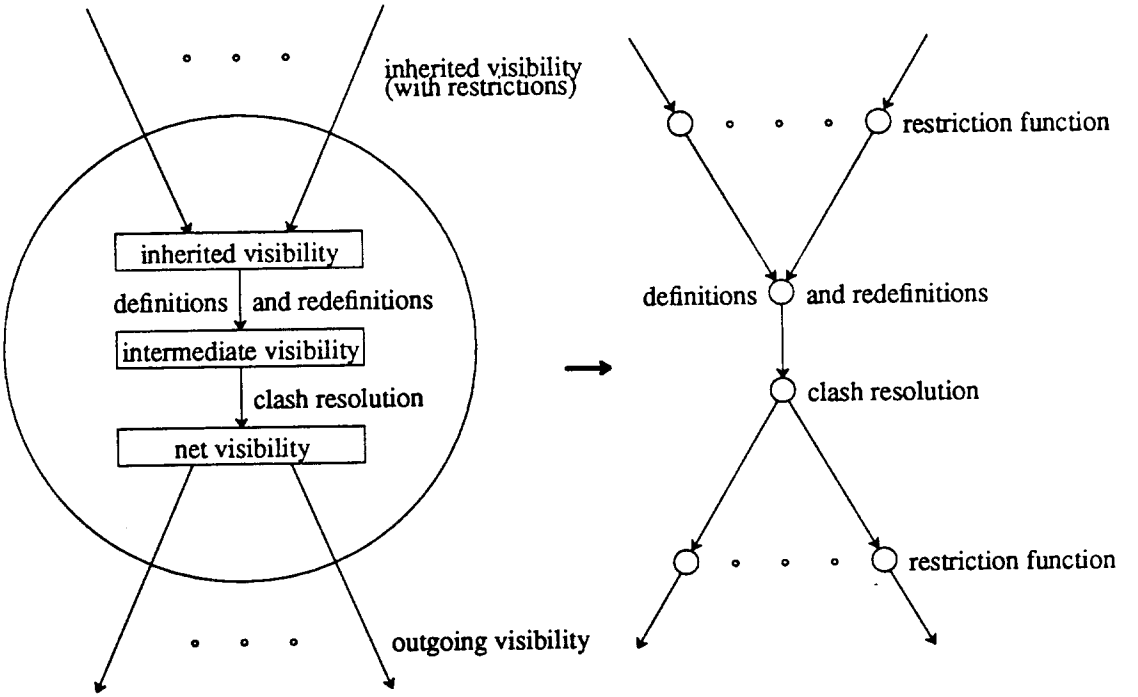


Figure 8.1: Inheritance Graph to Flow Graph Transformation

[45] Conventional DFA. Some algorithms, such as Graham-Wegman path compression [Graham and Wegman 1976; Wegman 1981] operate on edges. Then, the transformation is different (and simpler).

edges e_1 and e_2 were adjacent to the same vertex v in the original graph, with e_1 going into v, and e_2 going out of v. The transformation is illustrated in Figure 8.1.

(4) This graph will not be connected if IG_i is not connected, and thus is not a flow graph. This happens quite often. The IG_{known} subgraph for a Pascal program consists of many components – one component for each declaration scope. A dummy start vertex v_s can be created with an edge or edges to each component of the graph[46]. Vertices from each component must be selected for these edges such that each vertex in each component is reachable from v_s. The resulting flow graph is called FG_i.

KILL and DEF for each vertex in FG_i must be defined. Kills are generated by restriction functions, *redefine_vis* calls, and **shadow** entries in the clash table. Defs are generated by *define_vis* and *redefine_vis* calls. There are three kinds of vertices in FG_i:

- Vertices corresponding to inheritance edges between vertices in IG_i. Each such vertex has only one incoming edge and one outgoing edge.
- Vertices corresponding to the definition/redefinition step in an inheritance graph vertex. Each such vertex may have many incoming edges, but only one outgoing edge.
- Vertices corresponding to the clash-resolution step in an inheritance graph vertex. Each such vertex may have only one incoming edge, but many outgoing edges.

The following sections define KILL and DEF for each of these vertices.

Vertices Corresponding to Inheritance Edges

The restriction function for edge e_i in IG_i is rf_i. A restriction function restricts inheritance along an edge. The corresponding vertex in FG_i is v_i.

rf_i: *Binding* $\rightarrow$ **Boolean**

$\text{KILL}(v_i) \equiv \{ b \in Bindings \mid rf_i(b)=\textbf{false} \}$

$\text{DEF}(v_i) \equiv \varnothing$

Definition/Redefinition Vertices

$\text{KILL}(v_i) \equiv \{ b_{old} \mid \exists\, redefine\_vis(v_i, b_{old}, b) \}$

$\text{DEF}(v_i) \equiv \{ b \mid \exists\, define\_vis(v_i, b) \vee redefine\_vis(v_i, b_{old}, b) \}$

Notes:

If $\exists\, define\_vis(x, b_i)$, $redefine\_vis(y, b_j, b_k)$ such that $x=y \wedge (b_i=b_j \vee b_j=b_k)$ (*i.e.*, operations at the same node kill and def the same binding), then the meaning depends on order of operations. This is not allowed, because it makes DFA more difficult in some cases, as will be described later. In any event, it doesn't seem to make sense to both def and kill a binding at one node.

[46] Some formulations of DFA also require a common exit vertex. An alternative to making the entire graph a flow graph is to do DFA on each component separately, but we must still ensure that each component is a flow graph.

Clash Resolution Vertices

The precondition to $clash(b_i,b_j)$ is $b_i.name=b_j.name$. The precondition to $clash\_resolve(<b_i, vc_k>, <b_j, vc_l>)$ is $clash(b_i,b_j)$.

$$\text{DEF}(v_i, vc_k) \equiv \varnothing$$

$$\text{KILL}(v_i, vc_l) \equiv \{ b \in Bindings \mid \exists b' \in vis\_top_k(v_i),\ 1 \le k < l,$$
$$s.t.\ clash\_resolve(<b, vc_j>, <b', vc_i>) = \textbf{shadow}\ b \}$$

If shadowing between bindings with the same visibility class is allowed (the inheritance graph definition is not ordered), the definition of KILL is:

$$\text{KILL}(v_i, vc_l) \equiv \{ b \in Bindings \mid \exists b' \in vis\_top_k(v_i),\ 1 \le k \le l,$$
$$s.t.\ clash\_resolve(<b, vc_j>, <b', vc_i>) = \textbf{shadow}\ b \}$$

In this case, ambiguous inheritance graph definitions are very easy to create. The fixed point found (if any) depends on the order of propagation, because KILL depends on *vis_top*.

If priorities are allowed, then the definition of KILL is:

$$\text{KILL}(v_i, vc_l) \equiv$$
$$\{ <b,p> \in vc_k \mid (\exists <b', p'> \in vis\_top_k(v_i), 1 \le k \le l,$$
$$s.t.\ clash\_resolve(<b, vc_j>, <b', vc_i>) = \textbf{shadow}\ b)$$
$$\vee (\exists <b', p'> \in vis\_top_l(v_i),$$
$$s.t.\ clash\_resolve(<b, vc_j>, <b', vc_i>) = \textbf{prioritize} \wedge p' > p) \}$$

Data Flow Equations

Inheritance-edge vertices v_i: labeled with rf_i

$$vis\_top(v_i) = vis\_bot(y) \ \textbf{where}\ y \in PRED(v_i), \text{ and } y \text{ is unique}$$

$$vis\_bot(v_i) = vis\_top(v_i) - \text{KILL}(v_i)$$

Definition/redefinition vertices v_i: labeled with one or more *define_vis* or *redefine_vis* operations.

$$vis\_top(v_i) = \bigcup_{v_j \in PRED(v_i)} vis\_bot(v_j)$$

$$vis\_bot(v_i) = (vis\_top(v_i) - \text{KILL}(v_i)) \cup \text{DEF}(v_i)$$

Clash-resolution vertices v_i:

$vis\_top(v_i) = vis\_bot(y)$ **where** $y \in PRED(v_i)$, and y is unique

$vis\_bot(v_i) = vis\_top(v_i) - \text{KILL}(v_i)$

VIS (v, vc_i) is defined to be the set of binding/visibility-class pairs $<b', vc_i>, i<j$, visible at v. VIS $(v, vc_i) = vis\_bot(v)$ for IG_i, given the preceding data flow equations.

8.2.2. Computation of KILL and DEF for Each Node

Definition/redefinition functions operate on specific bindings, so the DEF bit vectors are easy to compute.

However, straightforward computation of bit vectors for KILL would be very inefficient, requiring $O(|Bindings| \times r)$ steps, where r is the number of nodes in the flow graph. This involves applying the clash resolution function and every restriction function to every binding $b \in Bindings$ to compute the bit vector corresponding to that function. This section describes more efficient ways of computing KILL for clash resolution and restriction functions.

Errors and the effects of other function calls in the clash table at v may depend on VIS (v, vc_j), so they must be done after DFA is completed.

Clash Resolution

Clash resolution must be performed at every clash resolution vertex in the flow graph. A different KILL bit vector must be computed for every such vertex and every visibility class, because the binding/visibility-class pairs $<b, vc_j>$ shadowed at vertex v because of clash resolution depend on VIS $(v, vc_i), i<j$. A brute-force approach would do pairwise tests of all bindings $b_1 \in Bindings$ and $b_2 \in$ VIS (v, vc_i) for shadowing, resulting in $O(|Bindings|^2 \times n)$ complexity to compute KILL for all clash resolution vertices, where n is the number of vertices in IG,

However, the restriction that only bindings with the same name clash can be used to reduce the expected complexity significantly. Of course, if all bindings have the same name, this doesn't help. Usually, however, relatively few names are used multiple times in a program. It is easy to track all bindings with the same name by implementing bindings with an abstract data type, and keeping track of all bindings created with the same name.

The following auxiliary functions are defined to track the auxiliary information needed to compute KILL efficiently for clash resolution:

$same\_name\_set(n:Name) \equiv \{ b \in Bindings \mid b.name=n \}$

$PCB \equiv potentially\_clashing\_bindings \equiv \{ b \in Bindings \mid size(same\_name\_set(b.name))>1 \}$

$NCB \equiv non\_clashing\_bindings \equiv Bindings - potentially\_clashing\_bindings$

The bindings in *PCB* can be grouped together at the beginning of the bit-vector representing *Bindings*. Also, bindings b, b' such that $b, b' \in same\_name\_set(b.name)$ can be grouped together. For small groups of bindings with the same name, their bits will be together in the same byte or word, which can make later bit set operations asymptotically more efficient. More generally, each set can be represented as a (*offset*, *bit_vector*, *length*) triple, where *offset* is an offset into the *Bindings* bit vector, and *bit_vector* has length *length* (in bytes or words, depending on the granularity of bit-set operations).

$clash\_set(b) \equiv \{ b' \in same\_name\_set(b.name) \mid clash(b,b') = \text{true} \}$

The value *clash_set* for a binding will also often be representable in a single byte or word. Given a set of bindings *bs*, the set of bindings that clash with bindings in *bs* can be computed by or'ing together *clash_set*(*b*) for all $b \in bs$. *clash_set* and the other auxiliary functions and sets can be computed once, before doing the DFA, because the set of bindings is fixed for a given program. Cyclical inheritance graph descriptions can add new edges, but not new bindings. Dynamic definitions can add new definitions of existing bindings. In an incremental environment, it must be possible to update these sets, which requires somewhat more sophisticated set operations.

Figure 8.2 contains an algorithm to compute KILL for all visibility classes and all vertices of inheritance graph *IG*.

The step at L1 requires $|potentially\_clashing\_bindings|$ search steps to find the bindings in *pcb*, plus $|pcb|$ union operations (often restricted to a single word or byte) to compute $\text{KILL}(v, vc_j)$. The step at L2 requires $|potentially\_clashing\_bindings|$ search steps to find the bindings in *pcb*, plus $n(n-1)/2$ calls on *f* for each *clash_set* partition of *potentially_clashing_bindings*, where *n* is the size of the *clash_set* partition. Each call on *f* may result in single-bit additions to $\text{KILL}(v, vc_j)$.

Inheritance Restriction Functions

A restriction function *rf* kills bindings that fail to satisfy a predicate specified by *rf*. Unlike KILL for clash resolution, which tends to be different for every vertex, there are usually only a few distinct restriction functions in an inheritance graph description, as witnessed by the Modula-2 example. The restriction function *rf_closed_scope* in the Modula-2 example is the

```
procedure compute_clash_kill
    for j ← 1 to the number of visibility classes
        ∀ v in IG such that v is a clash-resolution vertex
            DEF(v, vc_j) ← ∅
            KILL(v, vc_j) ← ∅
            for i ← 1 to (j−1)
                if clash−table(vc_i, vc_j) = shadow b_2 then
                    Let pcb = { b ∈ VIS(v, vc_i) | b ∈ potentially_clashing_bindings }
                    Preceding step is a zero-time operation, since pcb is a
                    contiguous group in VIS(v, vc_i)
                    L1: KILL(v, vc_j) ← KILL(v, vc_j) ∪ ( ∪ clash_set(b) )
                                                        b ∈ pcb
                elseif clash_table[vc_i, vc_j] is a function call f(b_1, b_2) then
                    Let pcb = { b ∈ VIS(v, vc_i) | b ∈ potentially_clashing_bindings }
                    L2: ∀ b_1 ∈ pcb, b_2 ∈ clash_set(b_1.name), b_1 ≠ b_2
                        actions ← f(b_1, b_2)
                        if shadow b_2 ∈ actions then
                            KILL(v, vc_j) ← KILL(v, vc_j) + b_2
                        Discard all other actions (including error messages)
                else error action: handled after computation of VIS(v, vc_j), for all vertices,
                        or empty clash table entry.
```

Figure 8.2: Compute KILL for Clash Resolution Vertices

same for all edges reflecting the default inheritance into a module. KILL$_{rf_closed_scope}$ can be computed once for an inheritance graph, and attached to every inheritance edge labeled with *rf_closed_scope*. Computing KILL$_{rf_closed_scope}$ requires iterating over *Bindings*, testing the *pervasive* attribute of each binding.

Some restriction functions depend on the instance of the production that created the edge, such as *rf_local_decls* and *rf_name_list* in the Modula-2 description, and thus a separate KILL vector must be computed for each edge labeled with one of these restriction functions. In the Modula-2 description, these are the functions that are computed as closures dependent on some program tree.

KILL for functions computed by *rf_name_list* depends only on testing equality of names of bindings. The computation can be optimized easily to kill only bindings with names from the list in the restriction function. The easiest way to do this would be to provide *rf_name_list* as a primitive, with KILL for a closure created with *rf_name_list* defined to be contain exactly those bindings with names in the *name_list* argument to *rf_name_list*.

Functions such as those computed by *rf_local_decls* depend on an attribute of a binding other than the name, and thus require testing all bindings.

8.2.3. Conventional Data Flow and the Inheritance Flow Graph

There are several differences between the graphs derived from inheritance graphs and program flow graphs, resulting from the fundamentally different language concepts they represent. The first difference is that inheritance graphs may consist of separate components, as discussed earlier.

The second difference is that a graph corresponding in structure to a deeply nested while-loop [Hecht 1977, p. 96] is common in inheritance flow graphs. This subgraph will occur frequently in any inheritance graph with edges of a single visibility class flowing in both directions in the graph. This bidirectional flow commonly occurs in languages like Pascal, which has declaration-before-use and requires that bindings be shadowed at the beginning of each scope. The inheritance graph subgraph corresponding to a list of declarations in Pascal is shown on the left side of Figure 8.3, with the corresponding flow graph shown on the right side. The number of nodes in the subgraph corresponds to the number of declarations local to a scope, which can be quite large.

This presence of this subgraph makes using certain DFA techniques undesirable because this subgraph tends to bring out their worst-case behavior. Interval analysis [Allen and Cocke 1976] requires one step in the derived sequence for every vertex in such a graph (less one). The complexity of interval analysis is $O(kr)$ bit-vector steps, for r much greater than k [Hecht 1977, p. 158], where r is the number of edges in the flow graph and k is the length of the derived sequence. Since $r \propto k$ in the nested-while subgraph, the complexity of interval analysis in this situation is $O(r^2)$.

Iterative algorithms will also tend to show worst-case behavior on this subgraph. For example, the standard round-robin algorithm requires $d+1$ iterations, where d is the number of back arcs in a path from a definition to a use. d for the while-loop subgraph is the number of nodes in the graph less one, resulting in a net complexity of $O(dr)$ bit-vector steps [Hecht 1977, p. 142], or (r^2), since $r \propto d$ for this subgraph.

The third difference is the likely presence of irreducible flow graphs. The inheritance graph in Figure 8.4 illustrates how adjacent imports can produce an irreducible graph (the expansion to a flow graph is omitted, but the expansion preserves irreducibility).

Thus, either a more efficient algorithm that handles irreducible graphs must be found, or node-splitting must be used.

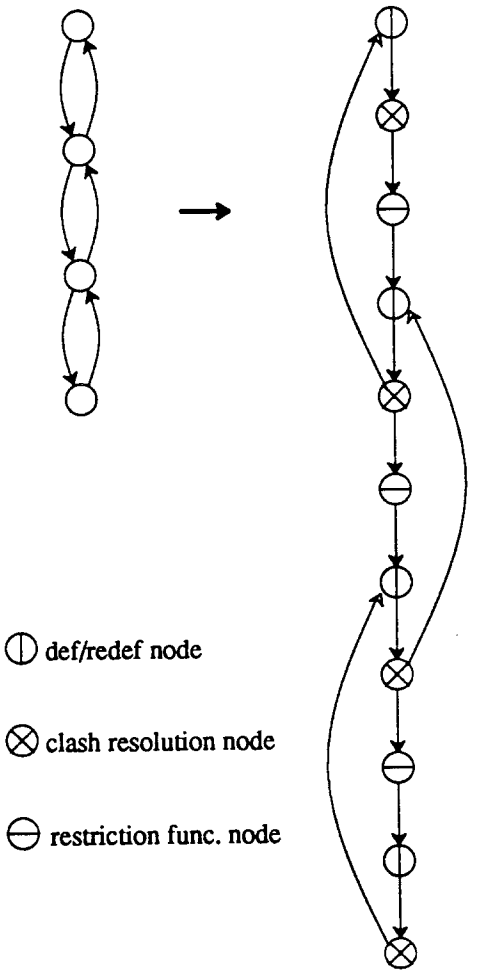


Figure 8.3: Nested While-Loop Graph

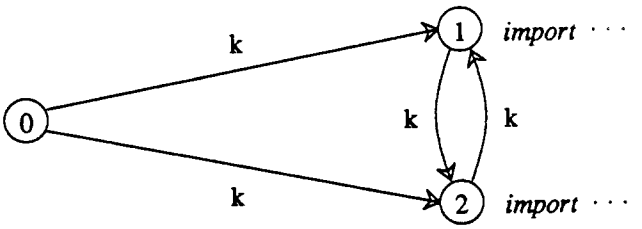


Figure 8.4: Irreducible Inheritance Graph

8.2.4. Graham-Wegman Global DFA

Graham and Wegman [Graham and Wegman 1976; Wegman 1981] developed an algorithm based on *path compression* which has a worst-case bound of $O(e \log e)$ function operations

on a graph with e edges. They give another complexity analysis that demonstrates that the number of function operations is proportional to e plus the number of non-structured exits from loops, resulting in linear performance for programs with only single-entry, single-exit control structures. The nested-while subgraph corresponds to nested single-entry, single-exit while loops, and thus can be analyzed using the Graham-Wegman algorithm in linear time. Inheritance graphs can contain subgraphs corresponding to multiple-exit control structures, but they are uncommon, and usually only affect the complexity of the DFA algorithm in an additive, not multiplicative manner.

In [Wegman 1981], Wegman describes an extension of the algorithm to handle irreducible graphs using node-splitting. This increases the worst-case bound to n^3 for a graph with n nodes, but analysis of the most common examples of irreducible subgraphs in inheritance graphs suggests that only a small added cost is due to each occurrence of the irreducible subgraph.

In Graham-Wegman DFA (GWDFA), the functions that affect the flow of information through the graph are attached to edges, so step (2) (in §8.2.1) in the conversion of IG_i into the corresponding flow graph FG_i is omitted. The nodes of FG_i correspond to the three stages of an inheritance graph vertex. The edges of FG_i correspond to inheritance edges, definitions/redefinitions, and the clash resolution process.

The fundamental concept of GWDFA is the stepwise reduction of the flow graph to a single node using three transformations: T_1', T_2', and T_3'. Each edge e is labeled with a function $f_e \in F$ representing the effect of traversing e on the information of interest. Each transformation $T_i'(G)$ produces a graph G' with a labeling F' such that the f_e for any e deleted in the transformation is represented by combining the f's for two or more edges in G into a single f' on an edge in G'. This will be clarified by means of a complete example.

Suitability of GWDFA

Suppose that X is the finite set of facts under consideration. In an inheritance graph partition IG_i, X is the set of all bindings, *Bindings*. GWDFA requires that the set F of functions be *fast*. Quoting from Graham and Wegman [Graham and Wegman 1976, p. 175], a function f is fast if $\forall X_1 \subseteq X, f(f(X_1)) \subseteq f(X_1) \cup X_1$[47]. A set of functions F is an *information propagation space* if (1) F is closed under composition and intersection, and (2) F is monotonic. A set of functions F is *fast* if (1) F is an information propagation space, and (2) all $f \in F$ are fast functions.

The set of functions F_{IG_i} composed of inheritance restriction functions, definition/redefinition functions, and clash resolution functions for IG_i (in an ordered inheritance graph) is fast:

(1) F_{IG_i} is clearly closed under intersection and composition.

(2) F_{IG_i} is monotonic, because each $f \in F_{IG_i}$ simply adds or deletes bindings, or both. If KILL for clash resolution for visibility class vc_i is allowed to depend on IG_i, then F_{IG_i} may not be monotonic. Thus, clash resolution for IG_i is allowed to depend only on previous partitions of IG.

(3) All $f \in F_{IG_i}$ are fast. For all $f \in F_{IG_i}, X_i \in X, f(f(X_i)) = f(X_i)$, so the requirement for f to be fast reduces to: $\forall X_1 \in X, f(X_1) \subseteq f(X_1) \cup X_1$, which is a tautology.

---

[47] Note that we substitute $\cup$ for $\cap$ and reverse the direction of the inequality, because the "reaching definitions" problem is a set-union problem, and Graham and Wegman's exposition assumed a set-intersection problem.

Therefore, $IP_{IG_i} = (IG_i, F_{IG_i}, Bindings, M)$, where M is a mapping from nodes to edges, is an *information propagation problem*, according to Graham and Wegman's definition, and is amenable to application of GWDFA. The goal is the computation of VIS for each node in the graph (corresponding to *vis_bot* in the earlier formulation as a conventional DFA problem).

GWDFA finds an *acceptable assignment AA*, which is a safe assignment, and is at least as good as a maximal (minimal in the case of set-union problems) fixed point. For the problem of computing VIS, this is the least fixed point.

Composition and Union of Functions

KILL and DEF corresponding to each function are as defined in §8.2.1, although they are associated with edges instead of vertices. The function associated with edge $e=(u,v)$ is $f_e(\text{VIS}(u)) = \text{VIS}(v) = (\text{VIS}(u) - \text{KILL}(e)) \cup \text{DEF}(e))$. The separation of each $f \in F_{IG_i}$ into a KILL part and a DEF part makes composition and union of functions easy. More precisely, suppose $e_1=(u,v)$ and $e_2=(v,w)$, and $\forall Y \subseteq X$,

$$f_{e_1}(Y) = (Y - \text{KILL}(e1)) \cup \text{DEF}(e1)$$

$$f_{e_2}(Y) = (Y - \text{KILL}(e2)) \cup \text{DEF}(e2)$$

then (derivation omitted, see [Graham and Wegman 1976])

$$f_{e_2}(f_{e_1}(Y)) = (Y - ((\text{KILL}(e1) - \text{DEF}(e2)) \cup \text{KILL}(e2))) \cup (\text{DEF}(e1) - \text{KILL}(e2)) \cup \text{DEF}(e2)$$

and

$$\text{DEF}(e_2 \circ e_1) = (\text{DEF}(e1) - \text{KILL}(e2)) \cup \text{DEF}(e2)$$

$$\text{KILL}(e_2 \circ e_1) = (\text{KILL}(e1) - \text{DEF}(e2)) \cup \text{KILL}(e2)$$

KILL(e) and DEF(e) are required to be disjoint for each edge e in IG_i, in order that the composition of KILL and DEF can be defined as shown.

For union of functions[48], we have:

$$f_{e_1}(Y) \cup f_{e_2}(Y) = (Y - (\text{KILL}(e1) \cap \text{KILL}(e2))) \cup (\text{DEF}(e1) \cup \text{DEF}(e2))$$

$$\text{DEF}(e_2 \cup e_1) = \text{DEF}(e1) \cup \text{DEF}(e2)$$

$$\text{KILL}(e_2 \cup e_1) = \text{KILL}(e1) \cap \text{KILL}(e2)$$

Reduction and Expansion of the Graph

Given these equations, it is possible to define the computations that must accompany each transformation. The reduction transformations and associated computations are shown in Figure 8.5. Edge numberings are kept consistent across transformations where possible. In applying a transformation T_i' to a graph G, producing G', edge e in G is labeled with f_e, while edge e in G' is labeled with f_e'. The canonical definition of f' for an affected edge is given (in terms of unions and compositions of f's), along with the value of KILL and DEF for the affected edge.

[48] This differs from Graham and Wegman's presentation, because they present a set-intersection problem.

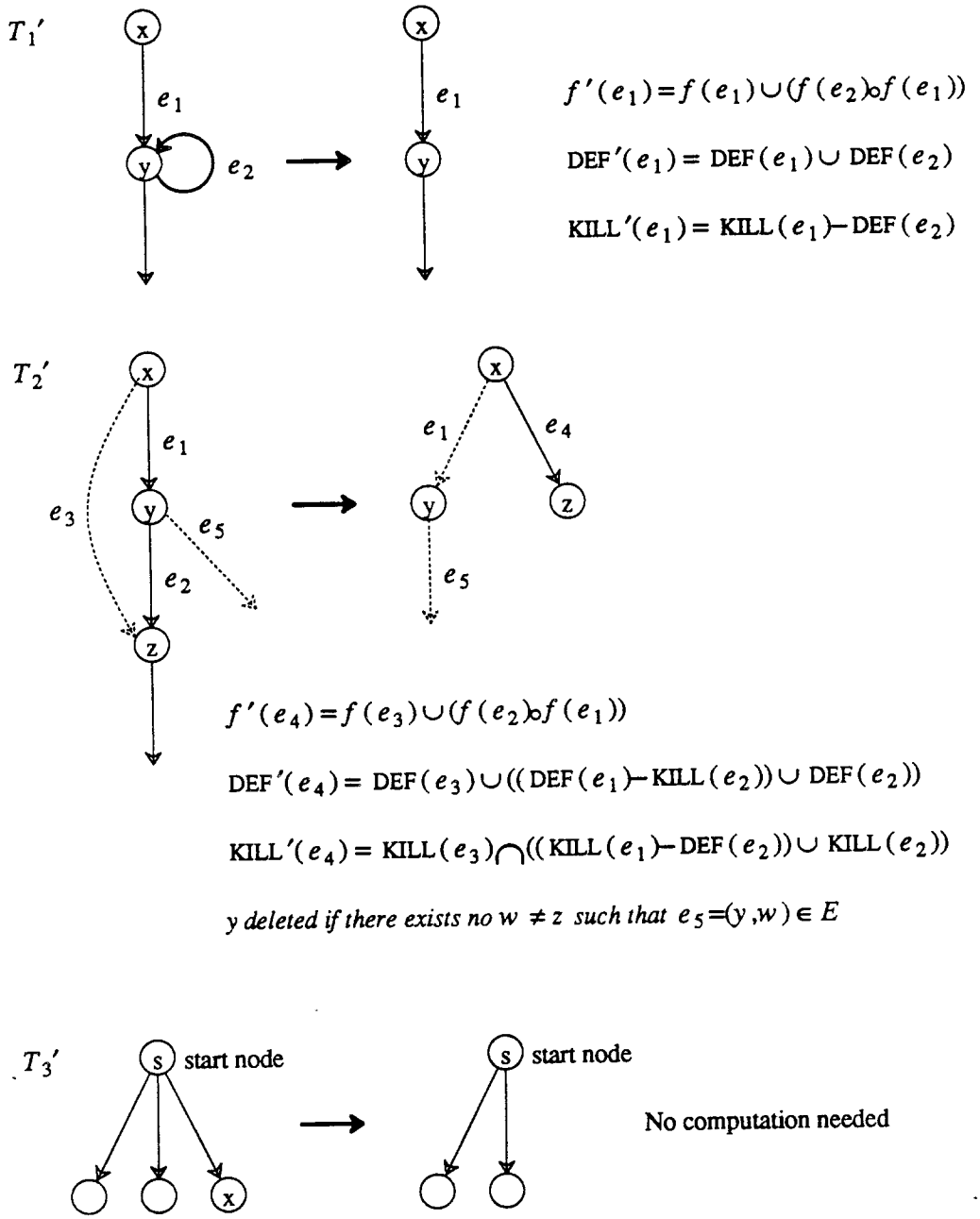
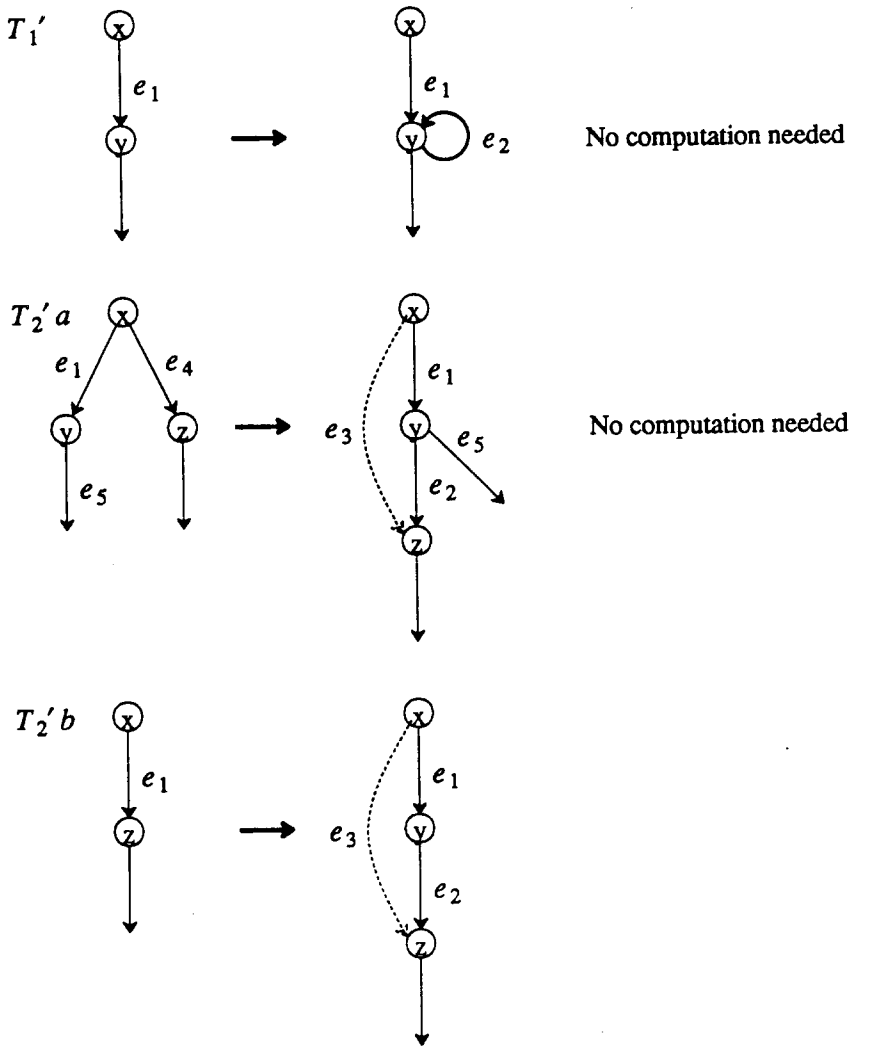


Figure 8.5: Reduction Computations for Graham-Wegman DFA

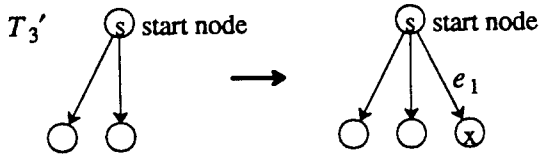
The derivations of KILL and DEF are omitted, but consist of substituting the definitions of composition and union of KILL and DEF given above into the definition of f', and simplifying.

T_1' removes a self-loop from a node v with a unique predecessor (other than itself). T_2' collapses the path x, y, z. For a nonexistent edge e, $KILL(e) = \mathbf{1} = Bindings$, and $DEF(e) = \mathbf{0} = \emptyset$, so that the absence of e_3 does not affect the computation in T_2'. If e_3 is absent in an actual graph reduction, the corresponding union or intersection operation is omitted. If y

has no successors other than z, then y is deleted by the transformation. T_3' simply deletes nodes whose only predecessor is the start node of the flow graph.



$$\text{VIS}(y) = f_{e_1}(\text{VIS}(x)) = (\text{VIS}(x) - \text{KILL}(e_1)) \cup \text{DEF}(e_1)$$



$$\text{VIS}(x) = f_{e_1}(\text{VIS}(s)) = (\text{VIS}(s) - \text{KILL}(e_1)) \cup \text{DEF}(e_1)$$

Figure 8.6: Expansion Computations for Graham-Wegman DFA

Each time the reduction deletes a node y, y has a single predecessor x, and $f_{(x,y)}$ defines VIS (y) solely in terms of VIS (x). In order to compute VIS for each node in the original graph IG_i, the graph is expanded from the start node to the original graph, applying inverse transformations in reverse reduction order. Each time a node y is added with predecessor x, VIS (y) is computed using the $f_{(x,y)}$ on (x,y) when y was deleted from the graph. Figure 8.6 shows the reverse transformations and the associated computations. It is not actually necessary to expand the graph – rather, one need just execute the expansion computations in the proper order.

In summary, Graham-Wegman DFA provides an efficient method of computing VIS for F_{IG_i}. It runs in near-linear time, where other DFA algorithms will exhibit $O(r^2)$ behavior on the potentially large subgraphs corresponding to declaration lists in languages with declaration-before-use. GWDFA also can handle irreducible graphs with node-splitting, though the details of that transformation are omitted. It also allows the use of bit vectors for efficient computations of union and intersection. The length of each bit vector will be equal to the number of elements of *Bindings*.

8.2.5. Error and Other Actions in the Clash Table

After VIS (v, vc_i) has been computed for all vertices v and all visibility classes vc_i, actions in the clash table other than *shadow* must be executed. These actions are either error calls, or function calls $f(b_1, b_2)$, where bindings b_1 and b_2 are being clash-resolved. The algorithm for processing these actions is:

```
procedure clash_actions
    actions ← ∅
    for j ← 1 to number of visibility classes
        ∀ v in IG such that v is a net-visibility vertex
        for i ← 1 to j−1
            ∀ b1 ∈ VIS(v, vc_i) such that b1 ∈ potentially_clashing_bindings
                ∀ b2 ∈ VIS(v, vc_i) such that b2 ∈ clash_set(b1.name)
                    actions ← actions ∪ clash_table[vc_i, vc_j](b1,b2)
```

If a clash error is defined to be associated with the clashing bindings (giving the error a place to be displayed), errors can be distinguished from other functions, with a more efficient implementation:

```
if clash−table(vc_i, vc_j) = error(...) then
    Let pcb = { b ∈ VIS(v, vc_i) | b ∈ potentially_clashing_bindings }
    Preceding step is a zero-time operation, since pcb is a contiguous group in VIS(v, vc_i)
    error_actions ← select_error_bindings( ∪ (clash_set(b) ∩ pcb), error(...))
                                           b ∈ pcb
    actions ← actions ∪ error_actions
elseif some other function ...
```

select_error_bindings produces errors (using the error argument) for all bindings in its binding-set argument.

8.2.6. Cyclical Inheritance Graph Descriptions

A cyclical inheritance graph description, in which construction operations depend on reference operations (*e.g.*, an import from a named scope), requires that an inheritance graph be evaluated repeatedly, adding the effects of any newly successful construction operations at each iteration, until the iteration stabilizes.

When using DFA to evaluate an inheritance graph, this involves repeating the DFA for each partition of the graph for each iteration. However, the changes to the graph are often small, and

incremental DFA could greatly decrease the amount of work needed for the second and later iterations. Incremental analysis is even more important for an implementation of a language-oriented editor, where declarations may be added or deleted, and the structure of the program changed.

Several methods for doing incremental DFA have been proposed [Pollock and Soffa 1987; Ryder and Carroll 1987; Ryder and Paull 1987; Zadeck 1984]. Methods based on interval analysis [Ryder and Carroll 1987; Ryder and Paull 1987] and iterative methods [Pollock and Soffa 1987] are not applicable because of the nested-while subgraph.

Zadeck's algorithm, called the *Partitioned Variable Technique* (PVT), is only applicable to a limited set of data flow problems, but it is applicable to the problem of computing VIS. PVT partitions the problem into independent problems called *clusters*, which are solved separately. For the VIS computation problem, a cluster consists of all bindings with the same name. Since bindings with different names do not clash, the clusters are clearly independent. The algorithm is linear in the number of edges in the graph for each cluster affected by a change. For many changes, the area visited by the algorithm is only a small part of the graph. Adding or deleting a definition only affects a single cluster, while adding or deleting a vertex or edge can affect many clusters.

GWDFA bears some similarities to interval analysis, for which Ryder, Carroll, and Paull give an incremental algorithm. It might be possible to apply the same methods to GWDFA. However, this possibility has not been analyzed thoroughly, and thus is an area for further research.

8.2.7. Efficiency Considerations

One problem with the application of DFA as described is that, although the number of bit-vector operations is near-linear in the number of edges, it is O(|*Bindings* |× *number of visibility regions*) in space. A Pascal program with 1000 declarations would require about 32K 32-bit words for storing visible sets for each visibility class, and a roughly proportionate number of bit-set operations.

However, if programs are modular, in reasonably sized compilation units, then the situation is much better. Only those bindings declared local to the compilation unit M being processed, plus those that can be made directly visible to M, must be included in the bit vectors used in the data flow analysis. Those bindings whose names appear in an import statement in M or which are directly visible in a scope that is opened by M fall into this category. Of course, how successful this strategy is will depend heavily on the language and specific programs being compiled. Implementation experience is necessary to evaluate exactly what implementation strategies are best.

8.3. Non-Automatic Implementation

Instead of automatically generating an inheritance graph and evaluating it using DFA or some other mechanism, it may be expedient to use the inheritance graph formalism as a guide to creating a manual implementation for a particular language. This section presents one such method.

A simple Search Model implementation is easy for languages with fairly simple visibility control rules. The principles outlined in §8.1 for automatic generation of an evaluator for an inheritance graph can be applied directly to a hand-written implementation. If the language being implemented has declaration-before-use, then optimization of the Inheritance Graph Model may be necessary to get an acceptably efficient implementation, because strict application of the Inheritance Graph Model will result in one vertex per visibility construct, and thus one search step per visibility construct. The search can be optimized by collecting the declarations local to a

scope into a hash table, and maintaining explicit source positions for the purpose of determining the relative order among declarations, and between declarations and references. Then, a search requires only one search step per scope on the search path, though maintaining the information on ordering of declarations and references in a scope is more complex than before.

If the language doesn't have declaration-before-use, then we need not worry about ordering information within a scope, and keeping all declarations local to a scope in a hash table is easy.

Data collected on a large sample of Pascal programs (a total of 133,500 source lines, ranging from 2500 to 31000 lines, covering a wide variety of applications) indicates that an average of only 2.5 search steps would be needed for a search in such an implementation.

8.4. Summary of Evaluation Methods

Two methods of automatically generating an inheritance graph evaluator from an inheritance graph description were presented: one based on the Search Model, and one using data flow analysis. An efficient implementation of the Search Model version is possible only if some restrictions are placed on how clash resolution is used. Using the clash resolution process to detect errors is very expensive using the Search Model. However, the Search Model can easily handle prioritized clash resolution.

Data flow analysis provides an elegant solution to the problem of finding a solution to each partition of an ordered inheritance graph. If the clash resolution functions, etc., meet some mild restrictions, then data flow analysis can be used to find a least fixed point assignment in near-linear time in the number of edges. However, this implementation may be fairly slow and space-consuming for large monolithic programs.

The implementation effort for data flow analysis is non-trivial. However, if data flow analysis is going to be used for program optimization, then the implementation of the data flow analysis can be shared, so the incremental implementation effort for evaluating the inheritance graph using DFA will be small.

The Inheritance Graph Model can also be useful as a guide to non-automatic implementation.

CHAPTER 9

Discussion and Conclusions

This chapter ties together the various concepts discussed in the preceding chapters, and also elaborates on some other ideas that were given short shrift in earlier chapters, and deserve more attention. Applications of the Inheritance Graph Model are discussed first. Next follows a comparison of the Inheritance Graph Model and related work. Some language design issues raised in this research are then discussed. Finally, this chapter contains a discussion of topics for future research, and a summary of the important points of the dissertation.

9.1. Applications of the Inheritance Graph Model

9.1.1. Language Study, Understanding, Comparison, and Design

Perhaps the important application of the Inheritance Graph Model is in the understanding and comparison of the visibility control rules of programming languages, using inheritance graph descriptions. An inheritance graph description in this discussion does not necessarily mean a complete, formal description as given for Modula-2 in Chapter 7. Rather, definitions of the visibility classes, the inheritance graph schemata for the visibility constructs, the clash function and table, and the describer-defined lookup functions are probably the most appropriate level of definition for this purpose. A language designer or implementor can create the inheritance graph description for a language, and use the description to gain a better grasp of the subtleties of the visibility control rules. The inheritance graph description helps to expose the complexities of visibility control rules: there seems to be a roughly proportional increase in the complexity of an inheritance graph description with the complexity of the visibility control rules being described. Of course, this comparison is mostly subjective, where we measure the complexity of an inheritance graph description by the number of visibility classes and the complexity of the subgraph schema corresponding to the visibility constructs of the language. The measure of complexity of the visibility rules of a language is based on the number of different visibility constructs and the difficulty of understanding each visibility construct and the interactions between the visibility constructs. Examples of visibility control features discussed in some detail in this dissertation that illustrate this point include:

Scope Contour-Relative Shadowing and Declaration-Before-Use
: This combination of features (found in Pascal and many other languages) requires the use of reverse-text-order ***known*** edges to represent the fact that a binding shadows clashing bindings at the beginning of a declaring scope, instead of just forward edges. These back-edges correspond to the fact that back-patching is needed in one-pass name resolution.

Nested Qualified References with Declaration-Before-Use
: This is a very complex feature to describe, because the meaning of such a reference depends on its relative position within each of the enclosing scopes. Its description requires the additional visibility class ***containing***.

Modula-2 References in Statements/Declarations
: In Modula-2, references in declarations require declaration-before-use, while references in statements do not. The distinction between the two kinds of references results in two visibility classes, ***decl_referable*** and ***stmt_referable***, being used instead of the single ***referable*** visibility class used in Pascal.

Since increased complexity of the inheritance graph description corresponds to increased complexity of the visibility control rules it represents (assuming reasonable care and skill on the part of the person creating the description), and thus increased difficulty of understanding the visibility control rules, the language designer can use the inheritance graph description to experiment with different versions of visibility control rules until a satisfactory balance of complexity and expressive power is found. Of course, one design goal should be minimum complexity for the desired expressive power of the visibility control rules.

The inheritance graph descriptions for different languages can be used to compare the visibility control rules of the languages. The comparison will be easier if the descriptions are written using the same style as much as possible, but this is true for any specification method. Such factors as the number of visibility classes, the meaning of each visibility class, the connectivity of each subgraph schema, and the clash function and table are all important points of consideration in such a comparison.

A language implementor can benefit from construction and study of the inheritance graph description for the language under consideration, because the description can aid in clarifying the exact meaning of each visibility construct.

9.1.2. Implementation Using the Inheritance Graph Model

As described in Chapter 8, an implementation of the visibility control rules of a language can be generated automatically from a complete inheritance graph description for the language, similar to the description given for Modula-2. The inheritance graph for a particular program can be generated from the description, and the data flow analysis techniques described in §8.2 can be used to find the least fixed-point assignment to the inheritance graph.

There are three significant disadvantages to using this approach. The first is that the implementation effort is significant: Graham-Wegman data flow analysis must be implemented, including node-splitting. Support for all of the auxiliary tasks must also be provided, including maintaining information about the assignment computed at each iteration so that stabilization of an iterative evaluation can be detected, as well as an oscillating evaluation. The error handler must be written such that only errors on the final iteration are kept and reported.

This is a much larger implementation effort than even a fairly complex language-specific symbol-table manager, but the effort is not prohibitive, considering that the usual compiler-compiler argument applies: the automated system can be used for many different languages.

The second disadvantage is the lack of analytical tools sufficient to ensure that a cyclical inheritance graph evaluation will not occur for a given inheritance graph description, for reasonably complex visibility control rules. The oscillation can be detected at evaluation time, but the only reasonable result if no program errors have been detected is a "compiler error" message to the user. If the hypothesis that all oscillating evaluations result from program errors is true (§6.8.3), then oscillation will only occur if the implementor omits an error check. A compiler error in this situation is not desirable, but it is better than compiling the program without complaint, as is the case in conventional implementations where error checks are omitted.

The third disadvantage is the lack of a good, concise, readable specification method based on the Inheritance Graph Model. The attribute grammar-based specification method used for Modula-2 has several drawbacks, discussed in §7.7.

The alternative to automatic generation of an implementation is hand-coding a symbol table manager. If this method is chosen, the inheritance graph description for a language (at the level of the inheritance graph schemata) can be very useful in designing the implementation. The concepts of the Inheritance Graph Model can be applied directly to the implementation, although the extent to which this can be done while achieving an efficient implementation will depend on the language being implemented.

9.2. Relation to Other Work

Only two specific attempts at modeling and specifying visibility control have been made prior to this research: those of Reiss [Reiss 1983] and of Wolf [Wolf 1985; Wolf *et al.* 1986a; Wolf *et al.* 1987]. Reiss's goal was a complete model and specification method. Since Wolf's emphasis was on more on specific concepts of visibility control ("provides" and "requests"), he did not carry his specification ideas very far. In the discussion that follows, their approaches are compared in detail to the Inheritance Graph Model.

Other, more general specification methods have been used to specify visibility control rules as one part of an entire language specification, such as denotational semantics and attribute grammars. The advantages and disadvantages of these specification methods are also discussed.

9.2.1. Reiss's Model

Order is implicit in Reiss's model of visibility control, as a consequence of its original development for use in generating batch compilers. Implicit ordering of visibility constructs makes specification easier in a batch language system, but more complex in an incremental system. Reiss adapted the symbol processing system described in [Reiss 1983] to work in the Pecan system [Reiss 1984a; Reiss 1984b], an interactive programming environment for which language implementations can be automatically generated. However, it is unclear how some of

Reiss's descriptions of the visibility control rules of languages are fairly concise. The specification of Ada's rules is only seven pages long. However, this conciseness is achieved largely through the heavy use of defaults. This approach seems very error-prone, because much of the specification is defined implicitly by omission.

The specification language is declarative, with specific keywords and clauses corresponding to specific visibility control features. For example, the **NONEST** keyword attached to an object (entity) specification (apparently) means that nested references to bindings of that class are not allowed. **AUTO** means that a binding will be automatically created when an undeclared name is referenced. **NOREDEFINE_IN_SCOPE** means that a name can't be redefined in the same scope. This approach also helps to make specifications concise, but it does so at the cost of limiting the power of the specification method.

Consider the issue of declaration-before-use and its interaction with scope contour-relative and declaration contour-relative shadowing (§4.4). The property **NOUSE_AND_REDEFINE** is provided to handle specify these choices. It means that "no redefinition is allowed after a use.[49]" The precise meaning of **NOUSE_AND_REDEFINE** is unclear – several interpretations are possible: (1) A reference "N" cannot precede the declaration of a binding of "N" in the same scope, or (2) A reference "N" that automatically creates a definition cannot precede an explicit definition of "N". Choice (1) can be further subdivided according to what happens if another binding of "N" is declared in the enclosing scope: does the reference get the enclosing binding, or is it an error?

None of the possible interpretations is sufficient to allow all possible design choices for declaration-before-use and shadowing. More such properties could be defined to handle these choices, but this approach of adding a new feature to the model for each new language feature is clearly undesirable, and impractical from the viewpoint of the person attempting to define the visibility control rules of a language. This problem is an inevitable result of using a declarative specification language that provides a list of specific properties that can be attached to the objects being defined, particularly in a problem domain as rich in subtleties as visibility control.

[49] This definition of **NOUSE_AND_REDEFINE** was gleaned from the source code of Pecan.

The Inheritance Graph Model allows straightforward definitions of all of these choices. The disadvantage of the Inheritance Graph Model with respect to this approach is that descriptions are harder to analyze because of their composition from lower level concepts, and because of the freedom the inheritance graph designer has in combining the concepts.

Reiss's Formal Model of Visibility Control

Reiss's formal model of visibility was summarized in §3.5.5. The formal model shares most of the same problems as the model represented by the specification language. The order of visibility constructs and references is still implicit, so variations in the meaning of multiple declarations of a name in the same scope are not possible. Visibility features such as multiple inheritance are not handled.

The advantage of the formal model is that the real meanings of the functions such as **LOOKUP** and **DEFINE** can be fairly easily discerned by examining the definitions of the functions, presuming unambiguous definitions of the basic relations are available. It is also possible to extend the model to handle other features such as multiple inheritance by modifying the functions defined in the model. Adding explicit ordering would require major changes to the model. The fact that the model must be modified to handle even common visibility constructs indicates that the model is not one of the fundamental concepts of visibility control, but is rather more like a partial specification of the visibility rules of a language with fairly general visibility rules: the specification is completed for a particular language by defining the basic relations, and determin-

9.2.2. Wolf's Model

The most significant difference between the Inheritance Graph Model of visibility control and Wolf's model is that the Inheritance Graph Model defines the meaning of each visibility construct locally (except named inheritance) while Wolf's model describes the meaning globally, by placing an edge between an entity E_1 and any entity that requests or provides access to E_1. Since the effect of a visibility construct *VC* is non-local, the visibility function that defines the edges corresponding to *VC* must somehow "bridge the gap" between the local and non-local contexts to determine which nodes to connect with edges. Thus, the visibility functions are likely to be complex for more complicated examples than the one given in [Wolf *et al.* 1987].

Also, many of the visibility functions seem likely to be largely redundant, because similar edges must be "carried" to non-local parts of a program for many different visibility constructs. However, no examples of complex visibility constructs or complete languages have been done using this formalism [Wolf 1987], so it is difficult to know how most of the language features described in this dissertation would be defined.

Since there are two edges between any two entities where one is "visible" to the other in the conventional sense (a requisition and a provision edge), a visibility graph for a program can be very dense. It is unclear whether a language designer, implementor, or user would be able to extract much useful information from this graph. An inheritance graph, on the other hand, has only $O(n)$ edges, where n is the number of visibility constructs in the program, and the structure of the program with respect to visibility control is readily apparent.

The inheritance graph is in a sense related to Wolf's nesting graph, because both represent the inheritance structure of a program. Wolf simply did not carry this idea very far. This is apparently because Wolf's main concern was with language design issues, in particular the precise control over visibility of bindings. Most of his dissertation is concerned with issues related to the **provide** and **request** clauses. He also discusses the problems resulting from nested scopes, which were introduced as a mechanism for controlling visibility.

9.2.3. Denotational Semantics

The traditional approach to specifying the visibility rules of a language in denotational semantics [Stoy 1977] is to pass around an "environment" denotation representing the visibility structure of the program at any given point in the program. This environment denotation may take several forms. Most commonly, it is a structured value representing the nesting structure enclosing the point the particular environment value is associated with, along with the bindings associated with each scope. This method is of little benefit to the person specifying the language, because the manipulation, both modification and lookup, is still procedurally defined. The environment is essentially an encoding of interesting parts of the program tree; the functions for computing new environments must manipulate this encoding, and the access functions must traverse the encoding. The effect of each visibility construct cannot be localized, because all environment-related functions must know the meaning of every visibility construct.

An example of a language specification using denotational semantics in this manner is the formal definition of Ada [Honeywell 1980].

Another disadvantage of denotational semantics is that it is not yet possible to generate efficient implementations from specifications, although efforts have been made to generate better implementations [Ganzinger 1980; Mosses 1975; Paulson 1984].

9.2.4. Attribute Grammars

The specification of visibility rules using attribute grammars is usually done using techniques similar to those used in denotational semantics. A structured environment attribute is passed from production to production in the attributed program tree, with visibility constructs modifying the value passed appropriately. Both modification and accessing of the environment attribute are done procedurally, with auxiliary functions. The attribute grammar formalism helps in organizing and specifying the flow of the environment through the program: in attribute grammars intended for generation of batch compilers, the flow and manipulation of the environment attribute is simply an explicit version of the flow model style of manipulation of visibility control information used in traditional compilers[50].

Hoover [Hoover and Teitelbaum 1986; Hoover 1987] uses a different approach to propagating visibility information through the attributed program tree. The visibility information for a visibility region is stored in a structured attribute he calls an *aggregate value*. An aggregate value is a set of (key, element value) pairs representing a function from keys to element values. This is essentially the Visible Set Model described in §3.3, where each aggregate value represents the visible set for a particular visibility region in program. A (key, element value) pair corresponds to a name-entity binding. The aggregate value for a visibility region is computed by copying another aggregate value, adding a (key, element value) pair to another aggregate value, or by combining two other aggregate values, with a pair in the second aggregate value overriding (shadowing) a pair with the same key in the first aggregate value.

The advantage of Hoover's approach is that it forces the effects of visibility control to be exposed in the attribute grammar instead of in auxiliary functions which produce a new environment attribute from an old environment. And, manipulation of aggregate values is restricted in a way such that the attribute evaluator can accurately trace the effects of a change to an aggregate value (for example, one resulting from a new declaration added to the program), and reevaluate

[50] There is a big difference in implementation though, because evaluation of an attribute grammar often can't be done in one pass. It is usually necessary to keep all versions of the environment attribute, though a lot of sharing between versions is often possible, as in the Pascal example in [Kastens *et al.* 1981]. Also, it is possible to analyze the attribute grammar to determine when new versions of the environment can be produced by destructively modifying the old version, instead of by creating a new copy [Garrison 1984; Hudak and Bloss 1985].

only those attributes that are really affected by the change. This improved efficiency for updates was the primary motivation for the use of these visible-set aggregate values.

The disadvantage of the use of Hoover's aggregate values is that the operations provided for manipulating them are fairly limited. While simple visibility constructs can be fairly easily described, more complex visibility constructs are likely to be quite difficult to describe.

9.2.5. Plotkin's Operational Semantics

Formal specification methods based on operational semantics usually use the same approach to manipulating the "environment" as that used in denotational semantics: structured values manipulated procedurally.

Plotkin [Plotkin 1981] introduced a structural approach to operational semantics that avoids this procedural manipulation. The meaning of a program is defined by syntactic transition rules. These transition rules can be used to compute the visible set at any point in a program, using set operations, in terms of other visible sets and the effects of declarations. He only gives fairly simple examples, and it is unclear how more complex visibility constructs would be described, particularly constructs with non-local effects.

9.3. Language Design Issues

Several problematical areas of language design became apparent in the process of studying visibility control features, designing the Inheritance Graph Model, and applying the Inheritance Graph Model to real languages. These problems make the language more difficult for the language user to understand, and more difficult for the language-system developer to implement. This section discusses these issues and suggests changes in language design to avoid these problems.

9.3.1. Declaration-Before-Use Requirements

Much has already been written in this dissertation about the requirement of declaration-before-use (*DBU*) in many programming languages, primarily concerning how DBU complicates the specification of the visibility control rules of a languages and interacts in often unexpected ways with other visibility control features. This section discusses the history of and motivation for DBU, describes the resulting problems, and suggests an alternative.

Declaration-Before-Use: History and Rationale

Many modern programming languages evolved directly or indirectly from Pascal, and thus "inherited" their DBU requirement from Pascal. Pascal was certainly not the first language to require uses to precede declarations – COBOL [USDoD 1961] required DBU. However, the issue became more important as user-defined types became popular, and the issue of mutually recursive types became important. Also, Pascal is very widely known, and the reasons for the DBU requirement in Pascal are apparent, so Pascal is used as the canonical example of a language with DBU.

As discussed in §2.4.3, the first version of Pascal [Wirth 1972, p. 40] only included DBU as an implementation restriction. The DBU requirement was in the *User Manual* [Jensen and Wirth 1974, pp. 8,18] (but not the *Report*), and became an official part of the language much later [ANSI 1983; ISO 1980].

From the history of the development of Pascal, it is clear that implementation issues were the real reason for the DBU requirement: the issue was originally ignored except in an implementation restrictions section. The restriction was added because it makes one-pass semantic analysis possible (with some back-patching). The reasons for the gradual elevation in status are unclear: most likely were inertia and the desire to change things as little as possible. Explicitly removing the DBU requirement from the language definition would have complicated the lives of many

compiler writers, requiring major changes in most existing Pascal compilers.

However, ease of implementation is *not* the reason most commonly given for the "value" of requiring DBU. Programming style reasons are usually given instead: it is stated that if declarations precede uses, then a program will be easier to read sequentially from start to end. This is more appropriately an argument for a *style* of programming, rather than a language restriction. However, placing a declaration before all of its uses is not always the best way of writing a program to achieve good readability. This may clutter the program unnecessarily. The meaning of many procedures will be obvious if names are well chosen, or if they implement "standard" concepts, such as *pop* and *push* operations on a stack. The location of such procedures is unimportant to the reader most of the time: they should be grouped together according to purpose somewhere where they can be found easily, but they should not clutter up the more "interesting" code. The fundamental style problem with DBU is that it results in over-specification: it results in an ordering on declarations that may have no meaning to the programmer.

Requiring DBU results in a "reverse-order" program. The main body of the program is the very last part of the program text. Claiming that a program should always be read from lowest-level parts to highest-level parts is analogous to claiming that writing a program should be done strictly bottom-up. This is patently absurd: most programmers write programs using a combination of top-down and bottom-up techniques.

Requiring DBU also results in special-casing to handle mutually recursive types and procedures (**forward** declarations in Pascal) to preserve one-pass compilation. It can also lead to subtle problems, as in the forward-reference error in the Pascal example in §6.8.2 that many Pascal compilers miss. It can also result in strange programming styles: in some large Pascal programs, the ordering problem for mutually recursive procedures is avoided by providing forward declarations for all procedures at the beginning of the program. This prevents many forward-reference errors, but it also separates all procedure headings from their bodies, requiring a person reading a program to look in two separate places to get the entire definition of a procedure.

Improved program editors, particularly language-oriented editors, make the textual relation between definitions and uses less important – a use is either "close to" (visible in the same window) or "distant from" (can be viewed in a separate window or with elision) its declaration. If the editor has a "show me the declaration corresponding to this use" operation, the textual relation of "distant" declarations and uses is unimportant.

Placement of declarations with respect to uses is mostly a matter of programming style: declarations should be close to uses where possible; related declarations should be together, etc. The issue shouldn't be clouded with implementation issues, particularly when the the implementation issues aren't very important. One-pass compilation also becomes less important with the move to programming environments, which must maintain a representation of the entire program. Since the entire program is available, information about later points in the program is no longer "special", and can be made readily available. These implementation issues are related to the flow model vs. database model issue discussed in §5.1.

Specification Problems with Declaration-Before-Use

Requiring DBU significantly increases the complexity of the visibility control rules of a language, from the user's, the describer's, and the implementor's viewpoints. Without DBU, there is one contour per scope. With DBU, each declaration creates a new contour, so a scope then corresponds to many contours instead of just one, and the meaning of a name can no longer be determined solely from the scope immediately containing the reference.

Defining what constitutes an illegal multiple declaration is also easy without DBU: two clashing bindings result in a multiple-declaration error if and only if they are defined in the same visibility region (the same scope in this case). With DBU, binding definitions in all contours in a

scope must be considered.

Nested qualified references are fairly complicated to define with DBU, as evidenced in the Nested Qualified References sub-section of §6.5.4, because the ordering of declarations and uses must be considered for all scopes involved in the qualified reference. Without DBU, this problem evaporates. A nested qualified reference is identical to any other qualified reference: only the total set of bindings local to a scope need be considered at each step of the resolution of the qualified reference. Ordering of declarations and uses is irrelevant.

Shadowing and Declaration-Before-Use

Shadowing is straightforward when DBU is not required: a binding in an inner scope shadows any clashing binding in an enclosing scope. If overloading is permitted, then the meaning of a clash must be defined appropriately (the clash function in the Inheritance Graph Model).

If DBU *is* required, then the definition of shadowing becomes much more complicated. The choice between scope contour-relative shadowing and declaration contour-relative shadowing must be made. If scope contour-relative shadowing is chosen, a binding b declared in scope S is only referable after b's declaration, yet all bindings from enclosing scopes that clash with b are shadowed at the beginning of S. A specification of the visibility control rules must reflect the effects of the form of shadowing chosen. If declaration contour-relative shadowing is chosen, then a binding b in an enclosing scope that clashes with a locally declared binding b' is visible up to the declaration of b'. The precise locations where b becomes shadowed and b' becomes visible must also be specified: The declaration of b' may or may not be permitted to reference b. Also, a recursive reference to b may or may not be permitted within the declaration of b.

Finally, the effects of DBU "loopholes" must be considered, such as the allowance of forward references in pointer type declarations in Pascal. Such a forward reference amounts to a reference to a binding defined in a *inner* contour. The "illegal forward reference" error in the Pascal example in §6.8.2 is related: it is defined to be an illegal reference to a binding declared in an inner contour, even though a matching binding is declared in an enclosing contour (in the enclosing scope). It is difficult to devise a simple set of consistent definitions for scopes, contours, etc. because of these loopholes.

Declaration-Before-Use: Summary

Requiring DBU in a language design clearly creates many problems. Some of the problems arise because the interactions between DBU and other visibility control features are complex, and thus are relatively complex and difficult to describe, and difficult to understand properly. Some of the problems arise because DBU results in more choices that must be selected from, yet have little practical impact on the utility of a language. The added choices simply add to the difficulty of specifying and understanding a language.

The complexity of the DBU, particularly the interaction of DBU and scope contour-relative shadowing is illustrated by the frequency with which the Pascal fragment in Figure 6.23 is incorrectly implemented. Even the Pascal editor system developed using the Cornell Synthesizer Generator [Reps and Teitelbaum 1984], which is based on attribute grammars and does not have a one-pass requirement, implements this example incorrectly, binding the reference "T1" in P to the *T1* declared outside of P. In their documentation of the Pascal synthesizer, they state that a few semantic checks are not implemented, but adding this check would apparently require significant additions to their implementation of Pascal.

The cleanest implementation of this test for illegal forward references in Pascal is the one used in the GAG implementation of Pascal [Kastens *et al.* 1981]: every reference where a type name is expected is resolved twice. It is resolved once using the environment preceding the reference, and again using the environment at the end of the block containing the reference: if the two

lookups give different results, then the reference is an illegal forward reference.

The advantages of DBU are relatively small: some speedup and simplification of semantic analysis, primarily in batch compilers. In interactive implementations, DBU may be slower and more difficult to implement than the alternative. The programming style reasons usually given for DBU are not proper justification for the inclusion of DBU in a language: the DBU requirement often harms programming style more than it helps it.

For these reasons, DBU should not be included in language designs.

9.4. Avenues for Future Research

Improvements to the Inheritance Graph Model

Due to the definition of the **shadow** operator in the clash-resolution process, the result when a multiple declaration error occurs is not the most desirable one, although not necessarily incorrect. If two visibility classes ***known*** and ***referable*** are used, with the same meanings as used throughout this dissertation, then when clashing bindings b and b' are declared in the same scope, the result is that ***referable*** visibility of both bindings becomes shadowed, since a ***known*** binding shadows any clashing ***referable*** binding, leaving both bindings ***known*** but not ***referable***. The shadowing of ***referable*** visibility of both bindings was the primary reason for the edge-oscillation in the "**from P import** P" example in Figure 6.21. This shadowing also has the disadvantage that any attempted reference to these bindings will result in either an "name not declared" error or an "illegal forward reference" error, depending on where the reference occurs and on the definition of the lookup function. The former error message is somewhat reasonable, using the logic that erroneous declarations can't be referenced, but the latter error message would clearly be confusing. The ideal result is not to generate any error messages at attempted references to these bindings, on the grounds that since we don't know which of the bindings is the intended one, signaling errors on any of the references is meaningless.

One solution is to add an additional visibility class ***referable'*** such that every binding b is ***referable'*** in every visibility region (and nowhere else) where b would normally be ***referable***, except for the possibility of an illegal multiple declaration. Then, if the lookup function fails to find a matching ***referable*** binding, it can check for a matching ***referable'*** binding, and refrain from generating an error message if one is found. This solution is not entirely desirable, because it is a lot of effort for what seems to be such a simple problem.

A better solution to this problem is needed, either within the existing framework of the Inheritance Graph Model, or through a modification of the Inheritance Graph Model. An important criterion for any modification of the Inheritance Graph Model is the maintenance of the nice mathematical properties of clash resolution that make evaluation of an inheritance graph amenable to data flow analysis, unless another acceptable implementation method can be found without these restrictions.

One reason why inheritance graph descriptions are difficult to analyze for edge $\leftrightarrow$ ref dependency information is that the descriptions are two-level: The assignment to the graph can affect the edges in the graph, and vice versa. A model in which the edge set was fixed might be easier to work with. However, very little effort has been put into pursuing this option.

Specification Languages

The specification method used in Chapter 7 to define the correspondence between a Modula-2 program tree and its inheritance graph is long and repetitive, and therefore not very readable. It is also not amenable to analysis for discovering edge $\leftrightarrow$ ref dependencies, although it is not clear how much this is due to the specification method as opposed to the Inheritance Graph Model itself.

Thus, a better specification method would be useful if the Inheritance Graph Model is to be used in the automatic generation of implementations for visibility rules. Possible methods, still using attribute grammars, are Ganzinger's technique of parameterized compiler modules, and Watt's partitioned attribute grammars, as discussed in §7.7, to separate information about the inheritance graph from irrelevant syntax. A graphical specification language might also be useful, since what we wish to specify is a graph.

Analysis of Inheritance Graph Descriptions

Either in conjunction with or separate from the changes to the Inheritance Graph Model or specification method already discussed, better methods of analyzing an inheritance graph description for edge $\leftrightarrow$ ref dependencies and oscillation caused by dynamic definitions are needed.

A better understanding of precisely what language features and inheritance graph subgraphs lead to oscillating evaluations is also needed. The hypotheses in §6.8.3 concerning the relation between oscillating evaluations and program errors should be investigated further, looking either for a counterexample or better support for the hypotheses.

Incremental Data Flow Analysis

Each successive iteration of evaluation of the inheritance graph involves recomputing the data flow information for the graph. This information can either be completely recomputed for each iteration, or recomputed incrementally. Graham-Wegman DFA seems to be the method of choice for computing the data flow information, but it has never been adapted to incremental analysis. Zadeck's Partitioned Variable Technique is applicable to incrementally updating the data flow information on each iteration, but will be slower for the initial complete analysis. For the best evaluation-time execution speed, the implementor must implement both Graham-Wegman DFA and Zadeck's incremental technique.

An alternative is to adapt Graham-Wegman DFA to an incremental environment. The techniques used by Ryder, Carroll, and Paull for interval analysis [Ryder and Carroll 1987; Ryder and Paull 1987] might be applicable.

9.4.1. Multiple Inheritance and Clashes

When a language allows multiple inheritance, the possibility that multiple clashing bindings can be inherited by the same scope. This can occur with the Ada **use** clause, with Flavors, etc. This possibility introduces the need for a language design choice: what is the meaning of multiple bindings of a single name inherited by the same scope? This may be an error, or one binding may be selected via depth-first search, as in Flavors, or there may be a more complex selection mechanism, as in Ada. There are probably as many choices as there are languages with multiple inheritance.

The large number of choices is simply an indication of the complexity of multiple inheritance. There is a hierarchy of complexity of visibility control features, based on where one must look to find the binding denoted by a reference.

(1) One must look only in the contour containing the reference to find the binding (if any) denoted by the reference.

(2) One must look only in the contour containing the reference, and on the search path from the reference to the declaration of the denoted binding.

(3) One must look in contours not on the search path from the reference to the denoted binding in order to determine that the binding is actually the correct one.

The latter is the most complex, and can result from the presence of multiple inheritance, as it does in Ada. In Ada adding a binding to a **use**'d package can invalidate a reference to a clashing

binding in another use'd package. This can result in unexpected compilation errors, and can make programs much more difficult to understand. Other multiple inheritance designs have similar problems.

It is unclear what the solution to this problem is. The problem may be inherent in the use of multiple inheritance. However, there may be some multiple inheritance design that avoids these problems without destroying the utility of multiple inheritance.

9.5. Summary

This dissertation has presented a comprehensive study of the issues involved in visibility control in languages. The discussion has been limited to programming languages, but most of the basic concepts should apply to any language with structured resolution of names.

An extensive and comprehensive survey of the visibility control features available in programming languages was presented in Chapter 2, with the goal of understanding what is required of a model of visibility control, and what the fundamental concepts of visibility control are. These fundamental concepts were described, along with the resulting requirements for a natural model (Chapter 4), and the available choices in the design of a model of visibility control (Chapter 5).

Chapter 3 presented several models of visibility control, including both models which have been implicitly used by programmers and language designers without ever having been formally stated, and formal models of visibility control that have been developed by others.

Chapter 6 presented the Inheritance Graph Model, a model of visibility control based on the concept of inheritance of visibility, with an extensive example of the application of the Inheritance Graph Model to the description of the visibility control rules of Modula-2 in Chapter 7. Chapter 8 presented a method of automatically computing an assignment to an inheritance graph, allowing resolution of references, based on data flow analysis.

In the process of doing the stages of research roughly corresponding to these chapters, I have discovered many problematic and subtle issues of language design relating to the issue of visibility control. The complexity of the problem of understanding the fundamentals of visibility control and providing an adequate model has turned out to be much greater than expected. Most of the difficult issues discussed here seem to have been largely ignored or overlooked by many language designers: their concerns were more with the functionality provided by various visibility control features than with the difficulties described. This is understandable, because the primary function of the visibility control rules in modern languages is to provide controlled yet flexible access to entities – concerns about ease of understanding and specification can easily take second place to concerns about achieving precisely the desired functionality. It is hoped that the recommendations given in this dissertation will encourage language designers to make the visibility control rules in their languages easier to understand. The difficulties described are not, in general, due to visibility control features that provide any real power or benefit to the user: rather they are due to the interaction of features such as declaration-before-use, which have little value to the user, with useful features.

The Inheritance Graph Model is a general and natural model of visibility control. It is composed of very few concepts: visibility regions, multiple visibility classes, definitions (of visibility of binding/visibility-class pairs), inheritance, and clash resolution. Yet, the Inheritance Graph Model is very powerful because its basic concepts closely match those of the problem domain. Almost all visibility control features considered can be described using the basic Inheritance Graph Model, and all of the remaining features considered can be described using the extensions. Because of the Inheritance Graph Model's success in describing all of the visibility features considered, one would expect it to be adequate to describe other visibilty features not considered or yet to be developed.

The Inheritance Graph Model abstracts out the usual details needed in other methods of defining visibility control rules: it encompasses the Search Model, the Visible Set Model, the Range Model, the relevant parts of Reiss's model[51], and Wolf's model.

The most important ideas presented concern the Inheritance Graph Model, along with the fundamental concepts of visibility control on which it is based, as a way of thinking about visibility control. The Inheritance Graph Model is useful for designing, understanding, and comparing languages. It is also a useful model for implementation: we can automatically generate an implementation of the visibility rules of a language from an inheritance graph description, or we can use the Inheritance Graph Model as a guide to a hand-written implementation.

[51] Reiss's model handles some problems related to visibility control, such as lexical analysis, that I have largely ignored.

Bibliography

AHO, A. V., SETHI, R., and ULLMAN, J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

ALLEN, F. E. and COCKE, J., "A program data flow analysis procedure", *Comm. of the ACM 19*, 3 (March 1976), 137-147.

AMERICAN NATIONAL STANDARDS INSTITUTE, American National Standard FORTRAN, 1966.

AMERICAN NATIONAL STANDARDS INSTITUTE, "American National Standard Programming Language PL/I", ANS X3.53-1976, New York, 1967.

AMERICAN NATIONAL STANDARDS INSTITUTE, "IEEE standard Pascal computer programming language: an American national standard", ISBN 0-471-88944-X, ANSI/IEEE 770 X3.97-1983, IEEE Press and Wiley-Interscience, New York, 1983.

ATKINSON, M. P. and MORRISON, R., "Procedures as persistent data objects", *Trans. Prog. Lang and Systems 7*, 4 (1985), 539-559.

BACKUS, J. W., BAUER, F. L., GREEN, J., KATZ, C., MCCARTHY, J., PERLIS, A. J., RUTISHAUSER, H., SAMELSON, K., VAUQUOIS, B., WEGSTEIN, J. H., VAN WIJNGAARDEN, A., and WOODGER, M., "Revised report on the algorithmic language ALGOL 60", *Comm. of the ACM 3*, 5 (1960), 1-17.

BACKUS, J., "Can programming be liberated from the von Neumann style?", *Comm. of the ACM 21*, 8 (Aug. 1978), 613-641.

BALLANCE, R. A., VAN DE VANTER, M. L., and GRAHAM, S. L., "The Architecture of Pan I", PIPER Working Paper, Computer Science Division, EECS, UCB, Berkeley, CA, Summer 1987.

BARRON, D. W., *An introduction to the study of programming languages*, Cambridge University Press, Cambridge, UK, 1977.

BAUMANN, R., FELICIANO, M., BAUER, F. L., and SAMELSON, K., *Introduction to ALGOL*, Prentice-Hall, Englewood Cliffs, NJ, 1964.

BIRTWISTLE, G., DAHL, O., MYHRHAUG, B., and NYGAARD, K., *Simula begin*, Auerbach, Philadelphia, 1973.

BLOWER, M. I., "An efficient implementation of visibility in Ada", *Proc. SIGPLAN 1984 Symp. on Compiler Const.*, June 1984.

BOBROW, D. G. and STEFIK, M. J., LOOPS: An object oriented programming system for Interlisp, 1982.

BONDY, J. A. and MURTY, U. S. R., *Graph Theory with Applications*, North-Holland, New York, NY, 1976.

BORNING, A. H. and INGALLS, D. H. H., "Multiple inheritance in Smalltalk-80", in *Smalltalk 80: virtual image version 2*, Xerox Palo Alto Research Center, Palo Alto, CA, 1983.

BUSAM, V. A., "A dictionary structure for a PL/I compiler", *International Journal of Computing and Information Sciences 1*, 3 (Sep. 1972), 235-53.

CLARK and WEISMAN, *LISP 1.5 primer*, Dickenson Press, Belmont, California, 1967.

CLARKE, L. A., WILEDEN, J. C., and WOLF, A. L., "Nesting in Ada programs is for the birds", *Proceedings of the ACM-SIPLAN Symposium on the Ada Prog. Lang.*, Boston, Mass.,

Dec. 1980, 139-145.

CLOCKSIN, W. F. and MELLISH, C. S., *Programming in Prolog*, Springer Verlag, New York, 1981.

CORMACK, G. V., "Extensions to static scoping", *Proceedings of the 1983 SIGPLAN symposium on programming language issues in software systems*, San Francisco, 1983.

CURRY, G., BAER, L., LIPKIE, D., and LEE, B., "Traits: An approach to multiple inheritance subclassing", *ACM-SIGOA conference on office automation systems*, June 1982.

DAHL, O. and NYGAARD, K., *Simula 67 Common Base Definition*, Norwegian Computer Center, June 1967.

DAHL, O. and HOARE, C. A. R., "Hierarchical program structures", in *Structured Programming*, DAHL, O., DIJKSTRA, E. W., and HOARE, C. A. R. (editor), Academic Press, New York, 1972, 175-220.

DEREMER, F. L., LEVY, P., HANSON, S., JACKSON, P., JÜLLIG, R., and PITTMAN, T., "Summary of the characteristics of several 'modern' programming languages", *SIGPLAN Notices 14*, 5 (May 1979), 28-45.

DEWAR, R. B. K., *The SETL programming language*, NYU, New York, 1979.

DIJKSTRA, E. W., "Notes on structured programming", in *Structured Programming*, DAHL, O., DIJKSTRA, E. W., and HOARE, C. A. R. (editor), Academic Press, New York, 1972, 175-220.

DIJKSTRA, E. W., "1972 Turing award lecture: the humble programmer", *Comm. of the ACM*, October 1972.

DRISCOLL, J. R., SARNAK, N., SLEATOR, D. D., and TARJAN, R. E., "Making data structures persistent", *Proceedings of the eightteenth annual ACM symposium on the theory of computing*, May 1986, 109-121.

FARROW, R., *Covers of attribute grammars and sub-protocol attribute evaluators*, Dept. of Computer Science, Columbia University, Sep. 1983.

FARROW, R., "Sub-protocol evaluators for attribute grammars", *Proc. SIGPLAN 1984 Symp. on Compiler Const.*, June 1984.

FARROW, R., "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars", *Proc. SIGPLAN 1986 Symp. on Compiler Const.*, July 1986, 85-98.

FELDMAN, J. A. and ROVNER, P. D., "An ALGOL-based associative language", *Comm. of the ACM 12*, 8 (Aug. 1969), 439-449.

FEUER, A. and GEHANI, N., eds., *Comparing and assessing programming languages - Ada, C, and Pascal*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

FOX, P., *LISP 1 programmers manual*, Internal Paper, MIT, Cambridge, Mass., 1960.

GANZINGER, H., "Some principles for the development of compiler descriptions from denotational language definitions", TUM-I8006, Technische Universität München, May 1980.

GANZINGER, H., "Description of parameterized compiler modules", *GI: 11. Jahrestagung, Informatik Fachberichte 50*, 1981.

GARRISON, P. E., "Modification-in-place in attribute grammars", Unpublished Manuscript, Computer Science Division, EECS, UCB, Berkeley, CA, June 1984.

GATES, G. W. and POPLAWSKI, D. A., "A simple technique for structured variable lookup", *Comm. of the ACM 16*, 9 (Sep. 1973), 561-5.

GOLDBERG, A. J. and ROBSON, D., *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.

GRAHAM, S. L. and WEGMAN, M., "A fast and usually linear algorithm for global flow analysis", *Journal of the ACM 23*, 1 (1976), 172-202.

GRAHAM, S. L., JOY, W. N., and ROUBINE, O., "Hashed symbol tables for languages with explicit scope control", *Proc. SIGPLAN 1979 Symp. on Compiler Const.*, Aug. 1979, 50-57.

GRISWOLD, R. E. and GRISWOLD, M. T., *The Icon programming language*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

HANSON, D. R., "Is block structure necessary?", *Software—Practice & Experience 11*, 8 (Aug. 1981), 853-866.

HECHT, M. S., *Flow Analysis of Computer Programs*, North-Holland, New York, 1977.

HEWITT, C., "PLANNER: A language for proving theorems in robots", *Proc. Int. Joint Conf. on AI*, Washington D. C., 1969, 295-301.

HIGMAN, B., *A Comparative Study of Programming Languages, 2nd. edition*, Elsevier North Holland, New York, 1977.

HONEYWELL, INC., CII HONEYWELL BULL, AND INRIA, Formal Definition of the Ada Programming Language, Nov. 1980.

HOOVER, R. and TEITELBAUM, T., "Efficient incremental evaluation of aggregate values in attribute grammars", *Proceedings of the SIGPLAN '86 symposium on compiler construction*, Palo Alto, CA, June 1986, 39-50.

HOOVER, R., "Incremental graph evaluation", 87-836, Dpt. of Computer Science, Cornell Univ., Ithaca, NY, May 1987.

HUDAK, P. and BLOSS, A., "Avoiding copying in functional and logic programming languages", *Conf. Rec. 12th ACM Symp. on Prin. of Prog. Lang.*, Jan. 1985, 300.

INTERNATIONAL STANDARDS ORGANIZATION, "Specification for the computer programming language Pascal", Second ISO 7185, Dec. 1980.

IVERSON, K. E., *A Programming Language*, Wiley, New York, 1962.

JENSEN, K. and WIRTH, N., "PASCAL, user manual and report", *Lecture Notes in Computer Science 18* (1974), Springer Verlag.

JOHNSTON, J. B., "The contour model of block structured processes", *SIGPLAN Notices 6*, 2 (1971), 55-82.

JONES, N. D. and MUCHNICK, S. S., "TEMPO: A Unified treatment of binding time and parameter passing concepts in programming languages", *Lecture Notes in Computer Science 66* (1978), Springer Verlag.

JÜLLIG, R. and DEREMER, F., "Regular Right-Part Attribute Grammars", *Proc. SIGPLAN 1984 Symp. on Compiler Const.*, June 1984, 171-178.

KASTENS, U., HUTT, B., and ZIMMERMANN, E., *GAG: A Practical Compiler Generator*, Springer Verlag, Berlin, 1981.

KEMENY, J. G. and KURTZ, T. E., *Basic Instruction Manual*, Dartmouth College, Hanover, NH, 1964.

KERNIGHAN, B. W. and RITCHIE, D. M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

KNUTH, D. E., "Semantics of context-free languages", *Math Systems Theory 2* (1968), 127-145.

LAMPSON, B. W., HORNING, J. J., LONDON, R. L., MITCHELL, J. G., and POPEK, G. J., "Report on the programming language Euclid", CSL-81-12, Xerox Palo Alto Research Center, Palo Alto, California, Oct. 1981.

LEBLANC, R. J. and FISHER, C. N., "On implementing separate compilation in block-structured languages", *Proceedings SIGPLAN symposium on compiler construction*, Denver, CO, 1979, 139-143.

LISKOV, B. H. and ZILLES, S. N., "Programming with abstract data types", *Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9*, 4 (Apr. 1974), 50-59.

LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, J. C., SCHEIFLER, R., and SNYDER, A., *LNCS 114, Clu Reference Manual*, Springer Verlag, New York, 1981.

MAURER, W. D., *The Programmer's Introduction to SNOBOL*, American Elsevier, New York, 1976.

MCCARTHY, J., "History of LISP", in *History of programming languages*, WEXELBLAT, R. L. (editor), Academic Press, New York, 1981.

MCDERMOTT, D. and SUSSMAN, G. J., "The Conniver reference manual", Memo 259a, MIT AI Laboratory, 1973.

MITCHELL, J. G., MAYBURY, W., and SWEET, R., "Mesa language manual", CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, California, Apr. 1979.

MOON, D. A., "Object-oriented programming with Flavors", *OOPSLA '86 Conf. Proc.*, Nov. 1986, 1-8.

MOSSES, P. D., *Mathematical semantics and compiler generation*, PhD Dissertation, Univ. of Oxford, 1975.

PARNAS, D. L., "On the criteria to be used in decomposing systems into modules", *Comm. of the ACM 15*, 12 (Dec. 1972), 1053-1058.

PAULSON, L., "Compiler generation from denotational semantics", in *Methods and Tools for Compiler Construction*, LORHO, B. (editor), Cambridge University Press, 1984, 219-257.

PLOTKIN, G. D., "A Structural Approach to Operational Semantics", DAIMI FN-19, Computer Science Dpt., Aarhus Univ., Aarhus, Denmark, Sep. 1981.

POLLOCK, L. L. and SOFFA, M. L., *An incremental version of iterative data flow analysis*, Computer TR87-58, Houston, TX, Aug. 1987.

PRATT, T. W., in *Programming languages: design and implementation, 2nd. edition*, Prentice Hall, Englewood Cliffs, NJ, 1984.

QUINE, W. V., *Word and Object*, MIT Press, Cambridge, 1960.

REES, J. A. and ADAMS IV, N. I., "T: a dialect of LISP or, LAMBDA: the ultimate software tool", *Conference record of the 1982 ACM symposium on LISP and functional programming*, 1982.

REES, J. A., ADAMS IV, N. I., and MEEHAN, J. R., *The T manual, fourth edition*, Computer Science Department, Yale University, New Haven, CT, 1984.

REISS, S. P., "Generation of compiler symbol processing mechanisms from specifications", *Trans. Prog. Lang and Systems 5*, 2 (Apr. 1983).

REISS, S. P., "Graphical Program Development with PECAN Program Development Systems", *SIGPLAN Notices 19*, 5 (1984). Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

REISS, S. P., "PECAN: Program Development Systems that Support Multiple Views", *Proc. of the 7th International Conference on Software Engineering*, 1984.

REPS, T. and TEITELBAUM, T., "The Synthesizer Generator", *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, Apr. 1984, 42-48.

REPS, T. and TEITELBAUM, T., *The Synthesizer Generator Reference Manual*, Dpt. of Computer Science, Cornell Univ., Ithaca, NY, August 1985.

ROUSSEL, P., *PROLOG – Manuel de reference et d'utilisation*, Groupe d'Intelligence Artificielle, UER de Luminy, University d'Aix-Marseille II, 1975.

ROWE, L. A., CORTOPASSI, J. R., DOUCETTE, D. P., and SHOENS, K. A., *Rigel Language Specification*, Computer Science Division, EECS, UCB, Berkeley, CA, June 1981.

RYDER, B. G. and CARROLL, M. D., "An incremental analysis algorithm for software systems", (solicited for Trans. Prog. Lang and Systems), Dpt. of Computer Science, Rutgers Univ., New Brunswick, NJ, 1987.

RYDER, B. G. and PAULL, M. C., "Incremental data flow analysis algorithms", *Trans. Prog. Lang and Systems (to appear)*, 1987.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., and WILPOLT, C., "An introduction to Trellis/Owl", *OOPSLA '86 Conf. Proc.*, Nov. 1986, 9-16.

SCHMIDT, D. A., "Detecting global variables in denotational specifications", *Trans. Prog. Lang and Systems 7*, 2 (Apr. 1985), 299-310.

SCHWANKE, B., "Survey of scope issues in programming languages", CS-78-131, Carnegie-Mellon University, Pittsburgh, PA, June 1978.

SCHWARTZ, J. T., *Abstract algorithms and a set-theoretic language for their expression (preliminary draft)*, Computer Science Dept., Courant Institute, NYU, New York, 1970-1971.

SCHWARTZ, J. T., *On programming: An interim report on the SETL project: Installment 1. Generalities; Installment 2. The SETL language and examples of its use*, New York Univ., 1973.

SCHWARTZ, J. T., DEWAR, R. B. K., DUBINSKY, E., and SCHONBERG, E., *Programming with sets: an introduction to SETL*, Springer Verlag, 1986.

SHIMASAKI, M., FUKAYA, S., IKEDA, K., and KIYONO, T., "An analysis of Pascal programs in compiler writing", *Software—Practice & Experience 10* (1980), 149-157.

STEELE JR., G. L. and SUSSMAN, G. J., "LAMBDA: the ultimate imperative", AI Memo 353, MIT AI Laboratory, Mar. 10, 1976.

STEELE JR., G. L., "LAMBDA: the ultimate declarative", AI Memo 379, MIT AI Laboratory, Nov. 1976.

STEELE JR., G. L. and SUSSMAN, G. J., "The Revised Report on Scheme: a Dialect of LISP", AI Memo 452, MIT AI Laboratory, Jan. 1978.

STEELE JR., G. L. and SUSSMAN, G. J., "The art of the interpreter or, the modularity complex (parts zero, one, and two)", AI Memo 453, MIT AI Laboratory, May 1978.

STEELE JR., G. L., *Common LISP: the language*, Digital Press, Digital Equipment Corporation, 1984.

STOY, J. E., *Denotational semantics: the Scott-Strachey approach to programming language theory*, The MIT Press, Cambridge, 1977.

SYMBOLICS, INC., *Reference Guide to Symbolics Common LISP: Language Concepts*, Symbolics Release 7 Document Set, 1986.

TUCKER, A. B., *Programming languages*, McGraw-Hill, New York, NY, 1977.

UNITED STATES DEPARTMENT OF DEFENSE, "COBOL 1961, Revised specifications for a common business oriented language", #1961 0-598941, U. S. Government Printing Office, Washington, D.C., 1961.

UNITED STATES DEPARTMENT OF DEFENSE, "Military standard, Ada programming language", ANSI/MIL-STD-1815A, U. S. Government Printing Office, Jan. 1983.

VAN WIJNGAARDEN, A., *Revised report on the algorithmic language ALGOL 68*, Springer Verlag, 1976.

WAITE, W. M. and GOOS, G., *Compiler Construction*, Springer Verlag, New York, 1984.

WASSERMAN, A. I., SHERETZ, D. D., KERSTEN, M. L., DERIET, R. P., and DIPPÉ, M. D., "Revised report on the programming language PLAIN", *SIGPLAN Notices 16*, 5 (May 1981), 59-80.

WATT, D. A., "Modular description of programming languages", Unpublished manuscript, Computer Science Division, EECS, UCB, Berkeley, CA, May 1985.

WEGBREIT, B., "The ECL programming system", *AFIPS Conference Proceedings*, 1971, 253-262.

WEGMAN, M., "General and efficient methods for global code improvement", Ph.D. Dissertation, Computer Science Division, EECS, UCB, Berkeley, CA, Dec. 1981.

WEGNER, P., "The Vienna definition language", *Computing Surveys 4*, 1 (Mar. 1972), 5-63.

WEINREB, D. and MOON, D., *LISP Machine Manual*, MIT, Cambridge, Mass., Mar. 1981.

WELSH, J., SNEERINGER, J., and HOARE, C. A. R., "Ambiguities and insecurities in Pascal", *Software—Practice & Experience 7*, 6 (1977).

WEXELBLAT, R. L., ed., *History of programming languages*, Academic Press, New York, 1981.

WHITEHEAD, A. N. and RUSSELL, B., *Principia Mathematica*, (no listed publisher), Cambridge, 1910-1913, 2nd. ed. 1925-1927.

WIRTH, N., "The programming language Pascal (revised report)", 5, Eidgeno:ssische Technische Hochschule Zu:rich, Fachgruppe Computer-Wissenschaften, Nov. 1972.

WIRTH, N., "Modula: a language for modular multiprogramming", *Software—Practice & Experience 7* (1977), 3-35.

WIRTH, N., "Report on the programming Language Modula-2: third, corrected edition", in *Programming in Modula-2*, Springer Verlag, New York, 1982.

WOLF, A. L., "Language and tool support for precise interface control", COINS Tech. Rep. 85-23, Computer and Information Science Dept., Univ. of Massachusetts, Sep. 1985.

WOLF, A. L., CLARKE, L. A., and WILEDEN, J. C., "The AdaPIC Toolset: Supporting interface control and analysis throughout the software development process", COINS Tech. Rep. 86-51, Sep. 1986.

WOLF, A. L., CLARKE, L. A., and WILEDEN, J. C., "A model of visibility control", *IEEE Computer Society International Conference on Computer Languages*, Miami Beach, FL (later version submitted for journal publication), Oct. 1986, 182-189.

WOLF, A. L., *Private communication*, Computer and Information Science Dept., Univ. of Massachusetts, 1987.

WOLF, A. L., CLARKE, L. A., and WILEDEN, J. C., "A model of visibility control", COINS Tech. Rep. 87-12 (submitted for journal publication), Computer and Information Science Dept., Univ. of Massachusetts, Feb. 1987.

WULF, W. A., RUSSELL, D. B., and HABERMANN, A. N., "BLISS: A language for systems programming", *Comm. of the ACM 14*, 12 (1971).

WULF, W. A. and SHAW, M., "Global variables considered harmful", *SIGPLAN Notices 8* (1973), 28-34.

WULF, W. A., LONDON, R. L., and SHAW, M., "An Introduction to the construction and verification of Alphard programs", *IEEE Transactions on Software Engineering 2*, 4 (Dec, 1976), 253-265.

ZADECK, F. K., "Incremental data flow analysis in a structured program editor", *Proc. SIGPLAN 1984 Symp. on Compiler Const.*, June 1984.

Index

Prepared by the author using the text processing tools *vi*, *(gnu)-emacs*, *spell*, *awk*, *sed*, *lpr*, *m4*, *bib*, *dtbl*, *deqn*, and *ditroff* with the *–me* macro package, coordinated by *csh*, *soelim*, and *nmake*. Printed on an Apple LaserWriter Plus. Processed mostly from pine.Berkeley.EDU, a Sun Microsystems 3/75, and dougfir.Berkeley.EDU, a Sun 3/60.