Hashed symbol tables for languages with explicit scope control[1]

S. L. Graham, W. N. Joy, and O. Roubine [2]
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Abstract

Newer programming languages such as Modula and Euclid provide more control than traditional Algol-like languages such as ALGOL-60 and Pascal over the inheritance, in inner scopes, of named entities available in outer scopes. They also provide mechanisms whereby chosen entities from inner scopes may be made available to outer scopes. In this paper we show how a hashed symbol table can be organized to implement these new scope rules in a time and space efficient way.

1. Introduction

In the traditional static scope rules of ALGOL-60 and its successors, an identifier, together with the object it designates, is declared in a certain scope, typically a block, and is automatically *inherited* or *visible* in all inner or nested scopes, unless the identifier is reused in another declaration. The use of the identifier is not visible outside the block in which the declaration occurs, although the named object may be accessible via aliasing.

Certain modern programming languages depart from the classical ALGOL-60 like scope rules by limiting automatic inheritance in various ways. Examples of such languages include Euclid [LHLMP 77], Mesa [MMS 78], and Modula [Wirth 77]. The new scope rules cause a problem for the compiler writer, in that the symbol table organization appears to be substantially complicated in dealing with this additional complexity. To quote Wirth [Wirth 77]: "The additional sophistication of the compiler's symbol table mechanism is rather considerable, and in any case greater than anticipated."

This paper gives an algorithm for such an organization which is modest in both space (about 4 extra bits per symbol table entry) and time (a small and bounded amount of additional work at scope exit time). We describe briefly the classical block structured symbol table organization, sketch a simple solution when explicit scope control is added, examine some of its problems, and then briefly present our new algorithm which solves some of these problems.

We then consider in some detail how our scheme can be extended to handle many of the forms of scope control presently contemplated by language designers. By exposing the implementation issues, we provide some guidance to the language designer in understanding the consequences of these design choices and offer help to implementers of these features. The issues we address are chosen from Euclid, Modula, and Mesa, but are found in other languages as well.

2. Classical block structured symbol tables

Block structured symbol tables for Algol-like languages are described in textbooks such as [AU 77]. A typical compile-time organization is as follows: There is a record or entry for each named object or symbol. The entries are placed in the symbol table in stack fashion as the declarations are processed and are removed (or popped) from the symbol table when the entire scope has been processed. The entries are accessed via hash addressing, where the hash key is the identifier and collisions are resolved by chaining. A new entry is placed at the front of the chain for its hash bucket. Each entry which is removed from the symbol table must also be unlinked from the hash table. An auxiliary vector, the *blockTable,* contains, for each nested scope, a pointer to the first of the contiguous entries for objects declared in that scope.

3. The new scope rules

The classical block structure scheme in which every name is automatically inherited is easy to implement, but can cause programming problems for the user of the language. The opposite extreme, in which no names can be inherited, is equally easy to implement but is too restrictive a language feature for the programmer. A compromise is to allow the programmer to specify explicitly which names are or are not inherited by inner scopes. There are sometimes two kinds of scope – one in which inheritance is automatic and one in which it is not. We refer to the former as an *open* scope and the latter as a *closed* scope. In most cases the programmer can explicitly indicate inheritance into a closed scope. We refer to explicit inheritance as *importation,* and, like Modula, speak of a **use** list of symbols to be imported [3].

The introduction of closed scopes and imported names complicates symbol table maintenance. In the absence of these features, a name is visible in the entire scope of its declaration. In other words, any name found by the lookup routine is visible. Once the features are introduced, the classical symbol table organization must be modified to distinguish between visible

[1] Research supported by NSF Grant MCS74–07644–A03 and by an IBM Graduate Fellowship to W. N. Joy
[2] Current address: CII-Honeywell-Bull, 78430 Louveciennes, France
[3] We assume for now that symbols are imported without requiring the names to be qualified.

Figure 1 – Example of closed scope

```
var p0,q0,r0: Boolean
module level1
        define x1,y2                    {x1 and y2 are exported to the enclosing scope}
        use p0,q0                       {p0 and q0 are imported from the enclosing scope}
        var x1,w1: integer              {x1,w1 are declared}

        module level2_1
                use x1,p0               {x1 and p0 are imported from level1}
                        .
                        .
                        .
        end level2_1

        module level2_2
                define y2               {y2 is exported from level2_2}
                var y2,z2: real
                        .
                        .               {x1,p0 must not be visible in level2_2}
                        .
        end level2_2
                .
                .                       {x1,w1,y2,p0,and q0 are visible in level1}
                .
end level1
```

and inaccessible names and objects in the symbol table. In general, visible entries can be scattered throughout the table. Since visibility of non-local names changes as inner scopes are entered or exited, the information must be updated at scope entry and exit. Visibility is a property of particular named objects and cannot be inferred from the nesting level.

In order to provide facilities for data abstraction and information hiding it is also permitted, for certain kinds of closed scopes, that names lexically local to the closed scope be made accessible to the enclosing closed scope. We refer to such explicit widening of accessibility as *exportation,* and, like Modula, speak of a **define** list of symbols to be exported. Again in the case of exportation, visibility is a property of particular named objects and cannot be inferred from the nesting level of their declaration. In Modula, exported names are not qualified in the enclosing scope by the name of the scope in which they are declared.

4. Simple symbol table implementation

Suppose we are considering only closed scopes and explicit unqualified importation from or exportation to the immediately enclosing scope. Declarative information about an identifier can occur in one of three ways: in a declaration, in a **use** list importing an identifier, or in a **define** list indicating exportation. New uses of identifiers are placed in the symbol table when they first appear in one of these ways, and are removed when the outermost scope in which they are visible is exited. Figure 1 contains an example from a language in which modules are closed scopes.

When an inner scope is entered, the imported identifiers must be designated as visible and the other entries from the outer scopes must not be visible, but must remain in the symbol table, since they will become visible at some later time. At exit, the non-exported entries local to the inner scope must be removed from the table while the exported entries remain to become visible to the outer scope. After scope exit, all of the entries visible to the outer scope before processing of the inner scope must again be visible to the outer scope. Since scopes can be nested, multiple levels of importation or exportation are possible.

One way to maintain visibility in closed scopes is to make a new entry in the symbol table for each imported object. The new entry will serve as an indirect reference to the primary symbol table entry. It is then necessary to search only the contiguous entries in the current scope to find a visible object. At scope exit all entries for that scope except those for exported symbols can be removed as usual.

However, because of the possibility for multiple levels of importation and the conventions in many languages that certain objects are implicitly imported (for instance, by **use allbut** X), this solution can require a considerable amount of space. Exported objects might also require multiple symbol table entries.

A simple way of maintaining visibility information without adding extra entries is to associate with each symbol table entry a *visibility list*, containing each nesting level at which the entry is visible. The lookup routine will return an entry only if the current nesting level is on its visibility list. Although maintaining these lists is relatively straightforward, they require a varying and potentially unbounded amount of space.

In fact, since an identifier can be exported only if it is visible in the inner scope, and can be imported only if it is visible in the outer scope, the entries in a visibility list must be consecutive level numbers. We can thus represent the sequence by its endpoints. Since, as we will see, it is possible to insure that an entry will be removed from the table upon exit from the outermost level at which it is visible, the lower endpoint of this range is implicit, and we need to add only a *visibilityLimit* field to each symbol table entry. Since the field will be updated, if necessary, at each scope entry, a symbol will be visible in this scheme if its *visibilityLimit* is the same as the current scope level.

Consider the example in Figure 1, disregarding for now the **define** lists and all occurrences of y2. Assume that the declaration of objects always precedes their occurrence on **use** lists, ie. their importation [4]. Figure 2 shows the symbol table at each closed scope. At the outermost level, the symbol table has entries for p0, q0, and r0, each with *visibilityLimit* set to 0. When level1 is entered, a symbol table entry is created for xl and its *visibilityLimit* field is set to 1. The *visibilityLimit* fields for p0 and q0 are also set to 1 but the entry for r0 is unchanged. Therefore p0 and q0 are visible but r0 is not. When the declaration is reached, a symbol table entry is created for w1, with the *visibilityLimit* field set to 1.

When level2_1 is entered, the *visibilityLimit* fields for x1 and p0 are set to 2, making those entries visible. When level2_1 is exited, the *visibilityLimit* fields for x1 and p0 are reset to 1, to insure that they are not visible in subsequent scopes at the same level such as level2_2. It is known that these objects were imported rather than local because their locations in the symbol table are outside the contiguous symbol table entries for level2_1 (which are delimited by the *blockTable* pointers). When level1 is exited, the entry for w1 is removed.

Now let us consider exportation. Typically, export indications (**define** lists in our example) occur before other declarations in a given closed scope and therefore before the declarations of the exported objects. Referring again to the example in Figures 1 and 2, entries are made in the symbol table for x1 and y2 when they are first encountered on **define** lists. This strategy guarantees that the entries are contiguous to

Figure 2 – Symbol table snapshots

symbolTable format: identifier, class, *visibilityLimit*
(other symbol table fields are not shown)
blockTable format: export count, *symbolTable* pointer

at the outermost level

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 0
	q0, *isVar*, 0
	r0, *isVar*, 0

within level1

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 1
1: 2,	q0, *isVar*, 1
	r0, *isVar*, 0
	xl, *isVar*, 1
	y2, *isExport*, 1
	wl, *isVar*, 1

within level2_1

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 2
1: 2,	q0, *isVar*, 1
2: 0,	r0, *isVar*, 0
	xl, *isVar*, 2
	y2, *isExport*, 1
	w1, *isVar*, 1

after exit to level1

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 1
1: 2,	q0, *isVar*, 1
	r0, *isVar*, 0
	xl, *isVar*, 1
	y2, *isExport*, 1
	w1, *isVar*, 1

within level2_2

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 1
1: 2,	q0, *isVar*, 1
2: 0,	r0, *isVar*, 0
	xl, *isVar*, 1
	y2, *isVar*, 2
	wl, *isVar*, 1
	z2, *isVar*, 2

after exit from level2_2

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 1
1: 2,	q0, *isVar*, 1
	r0, *isVar*, 0
	x1, *isVar*, 1
	y2, *isVar*, 1
	w1, *isVar*, 1

after exit from level1

blockTable	*symbolTable*
0: 0,	p0, *isVar*, 0
	q0, *isVar*, 0
	r0, *isVar*, 0
	x1, *isVar*, 0
	y2, *isVar*, 0

[4] This assumption is not valid for Modula. Declarations following **use** lists can be handled, but the space limitations for this paper prohibit inclusion of an explanation here.

those for the enclosing scope, thereby simplifying scope exit. The **define** list serves as a forward reference and the symbol table entries have a special indication (*isExport* in Figure 2) until the declarations are reached. A count of the number of new entries from the **define** list is kept in the *blockTable* so that they can be included in the enclosing scope when the present scope is exited. Since the entries are contiguous to those for the enclosing scope, they can be retained simply by moving the *blockTable* pointer.

When level2_2 is entered, the *visibilityLimit* field for y2 is set to 2, but no new entry is needed in the symbol table. When level2_2 is exited, the *visibilityLimit* field for y2 must be decreased. Because y2 is allocated with the objects for level1, the scope exit mechanism for imports handles this case as well. When level1 is exited the *blockTable* pointer for the outermost scope is moved down to include the two exported entries. The *visibilityLimit* fields are then decreased as before.

Modifications to the classical symbol table routines to work with importation and exportation using the *visibilityLimit* field are straightforward. The considerations become more complicated if, as in Modula, it is necessary to deal with unrestricted forward references of objects. We defer the issue of forward references to a later section of the paper.

5. A refined approach

The method just presented works well and uses little space but requires full symbol table searches at each scope exit to reset the changed *visibilityLimit* fields of symbols which were on a **use** list or a **define** list [5]. This search can be quite expensive. In this section we describe how we can avoid this search without using any additional space.

One might consider chaining the elements of each **use** list and each **define** list together, in order to reprocess the lists at scope exit. However, since a symbol table entry can be on one **use** list for each of many nested levels (or on one **define** list for each of many levels), there would again be a large use of space. Linking only the innermost **use** list or **define** list and restoring the next outer list at scope exit is quite complex, works only if the lists are ordered by outermost visible level, and still requires an extra link field for each symbol table entry. However, as we next show, it is possible to achieve the effect of linking the elements of the **use** list or **define** list by reordering the hash table collision chains. Just as the conventional symbol table organization represents locality implicitly by position, so too will our organization represent visibility by position.

5.1. Rearranging the hash links

In order to gather the imported and multiply-exported symbols together for restoration of *visibilityLimit* values at scope exit, we will rearrange the order of symbols on hash chains. Each symbol to be imported will be moved to the front of the chain before setting its *visibilityLimit* to the current nesting level. The same will be done for exported symbols which are already in the symbol table (such as y2 in level2_2 in our example.) Since the order of the hash chain does not affect the correctness of the search algorithm so long as the relative order of entries for the same identifier is preserved, this rearrangement does not introduce search errors.

At scope exit we must remove from the hash chains those local symbols which are not exported. In general, although all the visible symbols will be at the front of the hash chains, local and imported symbols will be mixed. If no nested importation occurs, then the local symbols will strictly precede the imported names on the hash chains; this is not true in the general case, however. To reconfigure the symbol table at scope exit we must examine each symbol table hash chain and process the symbols which have *visibilityLimit* equal to the current scope level. We do not need to search the full symbol table because all such symbols are at the front of the hash chain. If such a symbol was declared in an enclosing level or exported to that level, then we will leave it on the hash chain with its *visibilityLimit* decremented by one. We can tell whether a symbol is in an enclosing block using the *blockTable* of the classical organization. If a local symbol is not exported, then we remove it from the hash chain.

We can summarize the strategy here in one sentence by noting that the position of an entry in the symbol table reflects its declaration or outermost exportation, whereas its position on a hash chain reflects its current visibility.

5.2. Cost analysis

Both methods presented here use the same amount of space in the symbol table. The advantage of the refined method is that we do not need to search the entire symbol table at scope exit to reset the *visibilityLimit* fields. In the former case we could inspect a number of non-visible entries proportional to the length of the program. Instead, the cost which is incurred in the refined method is the iteration through the symbol table hash buckets. This iteration would be needed to release the local variables even if there were no closed scopes.

Experience with a scheme using the classical symbol table organization in a Pascal translator [JGH 79] shows that a symbol hash table size of 64 is sufficiently large to provide very fast identifier lookups while keeping scope exit costs low. With 500 symbols in the symbol table, the average length of a hash chain is roughly 8, giving a low limit on the number of iterations in the

[5] As illustrated by the previous example, the *visibilityLimit* fields must be reset to avoid visibility errors in parallel scopes. Most implementations of which we know use such a search. See, for example, [Hartmann 77].

inner loop of the lookup procedure. The time spent in the lookup procedure can also be expected to decrease because the self-organizing nature of the symbol table tends to place active visible symbols at the front of the hash chains.

6. Extension to other language features

In our discussion so far, we have considered a simplified set of scoping rules which did not include a number of features found in the languages under discussion. We next show how to incorporate some of these features in our symbol table design.

6.1. Standard Objects and Pervasive Identifiers

In Modula, certain predefined standard objects are automatically imported by otherwise closed scopes. In Euclid, this automatic importation attribute can also be given to programmer-defined objects, by designating them as *pervasive.*

Implementing these features is quite easy in our scheme. We define the *pervasive scope level* to be one more than the maximum scope nesting level and set the visibility of all standard or pervasive objects to this value. If the visibility test is that the visibility level of an object is *at least* the current scope level, (rather than exactly the current scope level) then standard or pervasive objects are visible if they are found by the lookup routine.

All that remains to be done is to insure that standard or pervasive objects cannot be manipulated in invalid ways. We must insure that the scope entry routine does not increment the current scope level beyond the maximum scope level, thereby inadvertently creating automatically imported objects. The import routine must check that an entry to be imported does not have a visibility level of the pervasive scope level, if, as in Euclid, it is an error to explicitly import an automatically imported object. The scope exit routine must not decrement the visibility level of an automatically imported object. In addition, in Euclid, the routine which enters new uses of identifiers must detect an attempt to redefine a pervasive identifier.

Since the programmer cannot create pervasive objects in Modula, we could place all the predefined standard objects at the beginning of the symbol table and identify them by position, instead of using the pervasive scope level. However, that solution would require a second test for every object found by lookup but having too small a visibility level.

Notice that since automatically imported objects may be declared prior to objects which are not imported into inner scopes, the speed–up in finding visible objects caused by rearranging the hash table will not extend to automatic importations.

6.2. Open Scopes

The only open scopes in Modula are routines without **use** lists. Euclid contains, in addition to modules and routines which are closed scopes, Algol–like blocks which are open scopes. The scheme we have described is easily generalized to handle a mixture of open and closed scopes.

We maintain the nesting level for all scopes, open or closed. As before, the *visibilityLimit* field of an entry is set to the current nesting level. However, when we look up an identifier, it is visible if its *visibilityLimit* is at least the scope level of the immediately enclosing closed scope. For simplicity in making this check, we maintain for each scope the level of the enclosing closed scope. The information can be saved in an appropriate new field in the *blockTable* and is adjusted at module and routine entry and exit. An identifier can be imported only if the value of its *visibilityLimit* field is at least the level of the enclosing closed scope of the innermost scope from which it is imported. This check can be made using the information saved in the *blockTable.*

6.3. Attribute Changes

In Modula, exported variables are *readonly* and exported types are *opaque.* The change in access must be recorded in the symbol table entry for each exported object upon exit of the scope in which the object is declared. This change can be made during the scope exit inspection to reset the *visibilityLimit* fields. For example, returning to Figures 1 and 2, the class of y2 could be set to *isConst* when the exit from level2_2 occurred. Upon exit from level1, the class of x1 would also be set to *isConst.* Note that no further change would be needed for y2.

In Euclid, variables can be imported into inner closed scopes with the attribute *readonly.* A readonly variable must be imported readonly into further nested closed scopes.

However, a variable may be imported through several levels and be imported readonly only the last few times. In general we must know at what level the object was first imported readonly so that we can treat it as readonly within this level but as a variable in outer scopes. The easiest solution is to add a *readonly* field to the symbol table field for each variable. The import routine can set that field to the outermost level in which the variable is imported readonly. In order for the mechanism to work properly, the field must be initialized to the pervasive scope level. An analogous mechanism can be used for attribute changes to types and constants.

7. Qualified Names

7.1. Record Fields

In languages containing record types, the fields of a record are normally named by identifiers. Since there are typically multiple instances of records of a given type in a program, record fields of particular records are designated by qualified names, the qualification indicating an object of the particular record type. For example, if the type of a variable x were a record with field *value,* the value field of that variable would be designated $x.value$.

There are two essentially different ways to access field names in a symbol table. The first is to have a logically separate symbol table for each set of fields. The qualification will serve as a signal to shift tables and will indicate which table to use. In a language such as Pascal, in which the field names occur consecutively in the type declaration, the field entries can follow the entry for the record type in the main symbol table but will not be on the main symbol table hash chains. Since there are typically few field names for a given record type, a linear search can be used. Alternatively, the field names can be entered in the hash table, thereby putting the entries in the main symbol table. Even for records with a small number of fields, the hash table lookup would seem to compete favorably in speed with linear search.

The disadvantage of hashing the field names is that entering the field names in the hash table introduces the possibility of multiple uses of the same identifier in the current scope, complicating and slowing the lookup and enter routines. However, by again considering the order of the hash chains and by placing field names on the chains after other uses of the same identifier in the current scope, the usual routines can be used for all objects except field names, and special routines can be provided for field lookup and field entry. Some measurement on typical programs would determine whether hashing the field names or linear search were better since the choice depends on the way in which field identifiers are used in these programs.

Since all exports are opaque in Modula, symbol table entries for fields should be removed when the immediate scope of the declaration is exited. The scope exit mechanism described so far handles record fields without further modification. Notice that if the record type is to be exported, entries for the fields might not immediately follow the entry for the record type. Consequently, in order to do linear search, the record entry would have to contain a pointer to its field entries. Similarly, if hashing is done, the entry for each field must contain an indication of the entry for the record type.

7.2. Modules as Types

In languages such as Euclid in which modules are types, the use of identifiers exported from a module is akin to the use of field names for a record; outside the module, the identifier must be qualified by some indication of the module type, such as a variable of that type.

Since exported identifiers must be qualified outside the module in which they are declared, much as record fields are qualified, the symbol table problem is similar to that discussed in the previous section; namely, there may be multiple bindings of a given identifier in a given scope, all but one of which require qualification. If the symbol table export and enter routines insure that the exported objects follow the hash chain entry for the unqualified use of the identifier, we can use a qualified lookup routine analogous to the field lookup routine described in the previous section, which inspects all entries for the identifier until it finds one exported from the appropriate module type.

Again, an alternative is a logically separate symbol table. Since the module objects must have qualified names in the scope immediately enclosing the module definition, they can be exported through further scopes only with an indication of their module types. Consequently, a logically separate symbol table can be constructed either by building a physically separate table, or by keeping the entries in the main symbol table but separating their access (as in the case of record fields). Since some module identifiers might be exported through further scopes than others, entries for module objects in the main symbol table might not be contiguous. Therefore separate access by linear search requires explicit links in the symbol table. Instead, one could construct a separate hash table for each module with exported identifiers. Either solution requires considerable additional space.

8. Selective Scope Widening

8.2. The with Statement

The **with** statement of Pascal and its derivatives causes the qualification for the fields of a given record type to be "factored out". In effect, a new open scope is entered in which the field names are declared, redefining prior uses of those identifiers. (Of course, qualified references to the fields are also possible.) The cleanest way to handle the widening of scope is to enter the field names at the front of their respective hash chains when the new scope is entered and remove the hash entries (but not the remainder of each symbol table entry) upon exit from the scope enclosed by the **with** statement. In fact, if the field names are already in the hash table, that is, if a logically separate table is not used, only those identifiers for which the appropriate entries would not otherwise be found by the lookup routine need be entered again in the hash chains. No new symbol table entries are necessary. This solution also handles nested **with** statements properly. The lookup routine will work correctly without modification, since visibility is determined by the enclosing closed scope. The disadvantage of this scheme is that it could incur extra overhead in the case that a **with** statement "opens" a record with a large number of fields, only a few of which are accessed. We anticipate that this situation will occur infrequently. Observe that since unqualified field names are indistinguishable from identifiers for other objects, any other solution incurs an overhead for all symbol table lookups within the scope of a **with** statement.

8.2 Definitions Modules

Certain languages, such as Mesa, contain named *definitions* modules which consist of a collection of declarations of variables, constants, types, procedure headings, etc. By importing the name of a definitions module, the declared objects are implicitly imported as well. Although the objects are normally designated by qualified names, the scope can be widened to eliminate the qualification. Green [Green 78] and other proposed languages contain a similar mechanism for unqualified importation of some or all of the objects declared within a closed scope.

The same technique (of entering the "new" names at the beginnings of the symbol table hash chains) can be used to access the imported fields, even if the symbol table for the imported module is separate (perhaps because of separate compilation). As the names are entered, a check can be made for name clashes if module importation does not create a new inner scope. The implicitly imported objects must be removed from the hash chains when the importation scope is exited. There are a variety of ways to detect that situation. Of course if a separate symbol table exists because of separate compilation, qualified names can signal use of the hash table for the separate symbol table.

However, unlike records, definitions modules can be expected to be rather large, particularly if they can be separately compiled. Consequently, there can be considerable time overhead in adding the names to the hash table. The alternative is to have a separate hash table for each definitions module and to have the lookup routine search those hash tables before the hash table for the main symbol table. This alternative introduces a multi-stage lookup for every symbol which is not within the last definitions module to be imported. Furthermore, it seems to have little advantage over the method of the previous section if it is necessary to check for name clashes.

9. Forward References

In Modula, it is possible to import and export objects whose declarations appear later in the text of the program. If such objects are used in statements and expressions, their attributes will not yet be known. Consequently, it will normally be necessary to process the statements in a subsequent pass. However, as we have seen for exports, and as also can be done for imports, it is always possible to bind the identifier uses to their corresponding symbol table entries, even if an identifier appears in a **use** list prior to its declaration. That is, the **define** and **use** lists serve as **forward** declarations for symbol table binding. Consequently, subsequent passes need not carry out symbol table searches or maintain *visibilityLimit* fields.

It is also possible in Modula to have forward references in open scopes; for example, mutually recursive procedures which are open scopes. In this case, symbol table binding cannot be done, except by some sort of complicated back-patching, since the scope of the forward reference is not known. Consequently, some sort of **forward** declaration for open scopes would aid the implementation.

There is also a kind of implicit forward reference that occurs in many of these languages. Typically, it is possible to export a procedure defined in a closed scope. In order to invoke the procedure, it is often necessary to provide actual parameters corresponding to the formal parameters of the procedure declaration. Consequently the visibility of the procedure parameter attributes is the same as that of the procedure name [6].

As the example in Figure 3 illustrates, the stack allocation of the symbol table described in the paper no longer works so well, since upon exit from M2, the symbol table entry for y should be released, but the entry for z should not, even if the identifier z is no longer visible.

Figure 3 – Implicit forward reference to parameters

```
module M1;
    define P;
    var x: integer;
    module M2;
        define P;
        var y: char;

        procedure P ( var z: Boolean);
            .
            .
            .
        end P;
        .
        .
        .
    end M2;
    .
    .
    .
end M1;
```

One solution is to do stack compaction when a scope containing the declaration of an exported procedure is exited. A better solution is to allocate the symbol table entries for the parameters of an exported procedure separately, while maintaining a common hash table.

A similar problem arises in Euclid because an enu-

[6] If, as in Mesa, actual parameters can be identified with the formal parameter names, the visibility requirement extends to the formal names as well.

merated type can be exported together with an exported module by specifying a Pascal–style subrange, thereby separating the symbol table entries for the value identifiers denoting the endpoints from the subsequent entries for the other value identifiers in the enumeration. In this particular case, the forward reference problem could be solved in the language design by allowing the enumeration to occur in the export specification and the subrange in the module declaration.

Note that the introduction of definitions modules alleviates the forward reference problems.

10. Conclusion

We have presented two related techniques to handle the complexity introduced by selective scope control; namely, the notion of visibility limits and the rearrangement of hash chains to reflect visibility. We have also discussed ways to incorporate various kinds of scope issues into our underlying symbol table organization.

As we have attempted to illustrate, the symbol table organization chosen for any one language must reflect a careful choice among various possibilities intended to handle particular aspects of the language design. Seemingly simple changes in the language, such as whether name qualification should be mandatory or optional, can have significant consequences for the symbol table organization [7].

Acknowledgements

Michael Curry participated in several discussions during the early stages of this work. Dan Halbert provided many helpful and constructive comments on an earlier draft of this paper. We thank them both.

References

[AU 77]

Aho, A. V. and Ullman, J. D. *Principles of Compiler Design.* Addison Wesley, 1977.

[Green 78]

*Reference Manual for the GREEN programming Language.*CII–Honeywell–Bull, 1978.

[Hartmann 77]

Hartmann, A.C. *A Concurrent Pascal Compiler for Minicomputers.* Springer-Verlag, 1977.

[JGH 79]

Joy, W.N., Graham, S.L., and Haley, C.B. *Berkeley Pascal User's Manual Version 1.1.* Computer Science Division, University of California at Berkeley, April 1979.

[LHLMP 77]

Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L. Report on the Programming Language Euclid. *Sigplan Notices.* January, 1977.

[MMS 78]

Mitchell, James G., Maybury, William, and Sweet, Richard. *Mesa Language Manual.* Xerox Palo Alto Research Center. Report CSL-78-1, February 1978.

[Wirth 77]

Wirth, Niklaus. Modula – A Language for Modular Multiprogramming. *Software–Practice and Experience* **7**, 1977.

[7] Note added in proof: We have just obtained a draft description of Modula 2. Among the changes, a form of definitions module has been added, the keywords **use** and **define** have been replaced by **import** and **export**, respectively, and the option for modules to export qualified names has been added.