3. Symbol Tables

3.1 Introduction

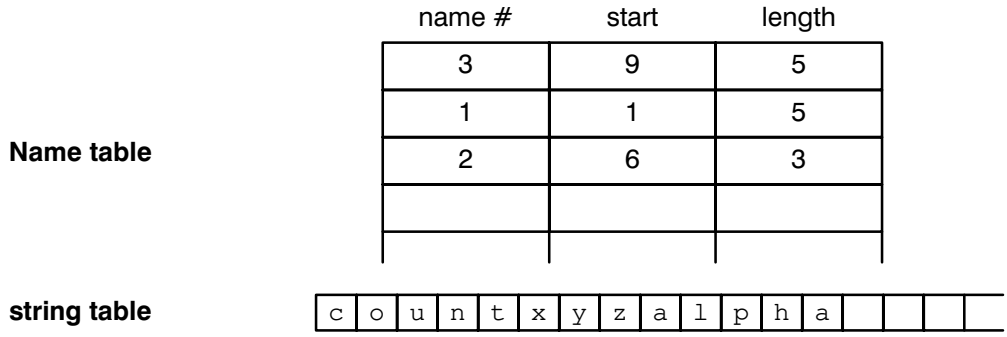
When writing a programme, the programmer creates many named objects – variables, procedures, constants, etc. – and the compiler needs to keep track of the *attributes* of all of these objects. The attributes are all the things the compiler needs to know about the object: the kind of object (variable, procedure, module, constant, type name, ...); the type, in the case of a variable or constant; the address, in the case of a procedure or a variable; the value, in the case of a constant; and so on. Notice that the sort of attributes we need to store depend on what kind of object it is. Early compilers for languages like Fortran had to deal with a fairly restricted set of objects, and could therefore adopt a uniform record structure for a symbol table entry which covered all cases. When compiling modern programming languages, we have to be a bit more flexible.

Many textbooks on compiler design still speak of a symbol table as if it were one single unified data structure. Present-day theories of program design favour an "information hiding" approach where the caller of a lookup routine only has access to that part of the information which it really needs. A good way to achieve this is to distribute the information over several different tables, each hidden within a different module. (This sounds messy when it is described in general terms, but when you get down to the implementation details you will discover that it simplifies the design effort, because each module only has to look after the details which are relevant to itself, and nobody needs to know the global picture.) Let us therefore use the term "symbol table" as a generic term, without requiring that the compiler actually have any real data structure which is "the" symbol table.

3.2 The Dictionary

The first piece of information which ought to be hidden away is the character string form of any user-defined names. Variable-length characters are a nuisance to carry around, and apart from the initial input (where a name has to be compared with previously encountered names) there are very few places in the compiler where the original form of the name needs to be known. If, for example, the programmer defines a variable called `ThisValue`, then all the rest of the compiler needs to know is that it is something like "name number 25". All we need to ensure is that all instances of `ThisValue` are mapped to the same number. The dictionary module, which will be called from the scanner, can keep a table showing the correspondence between the character string form and the number.

The dictionary module does *not* have to resolve the issues which arise from the fact that two separate objects might have the same name. Suppose, for example, there are two variables, in two different blocks, both called `ThisValue`. The dictionary module will not know that this string is the name of two different things, therefore it will return the result "name number 25" in all cases. The fact that name number 25 is used in two ways will be dealt with by higher-level modules.



The dictionary module is free to implement the name/number mapping in any way it likes; the appropriate tables are hidden from the rest of the compiler. One way to store the information is to use one very long array of characters to store all strings, and then have a separate table which records the starting position and

length of each string within the array. This is illustrated above for the case where three names `count`, `xyz`, and `alpha` have been encountered.

Notice that the name table is kept sorted alphabetically (for rapid searching) and that names are kept in the string table in the order in which they were found (since there is no particular reason for keeping this table sorted.) This is just one of many possible implementations – you might well find a better approach.

Whatever approach is used, it should allow for strings of arbitrary length, up to at least the maximum allowed size of a source line. (There is no need to go any higher than this, because the programmer is not allowed to split names across line boundaries.) Some compilers store only the first 32 characters (for example), but this can be very annoying for the user. It is true that user names are almost always unique within their first 32 characters, but very occasionally this will not be true and some mysterious undetected errors can occur.

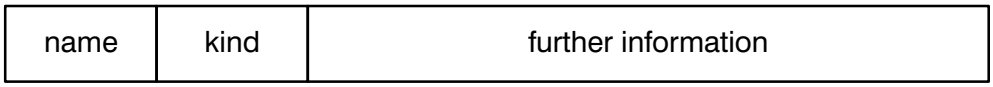
Notice that the dictionary module will have to look after words like BEGIN and IF as well as the user-defined names. The easiest way to do this is to insert all the reserved and predefined words into the dictionary at the start of a compilation. Subsequently, the scanner can tell the difference between these and the user-defined names by seeing that the the number returned by the dictionary lookup procedure is below a certain limit.

> *By the way, in case you don't know the distinction between a reserved word and a predefined word: a reserved word has a fixed meaning to the compiler, and it is illegal for the programmer to use the same string as a user-defined name. A predefined word can be re-used by the programmer; it is treated as if it had been defined in an imaginary block which surrounds the whole module. That is, the compiler treats predefined words no differently from user-defined names, except for the fact that information about them is already in the appropriate tables before the source file is read.*

An alternative approach is to have a separate table (which does not have to be inside the dictionary module) for all the reserved words (but not for the pre-defined words). This means that two tables must be searched every time a user-defined name is encountered, but it is justified if the check for reserved words is exceptionally fast. As it happens, there are very fast search algorithms – see the research literature on "minimal perfect hashing functions" – which can be used for the case of tables whose contents are fixed and known ahead of time.

3.3 Symbol Tables for Variables and Similar Objects

In principle, you could have one symbol table for procedure names, one for variables, one for constants, and so on. However this tends to be rather inefficient, because the compiler does not know what sort of object it is dealing with until it has looked up the symbol table(s). As a result, the preferred approach is to have a combined table with entries of the form



Here, "name" is the string number returned by the dictionary, and the "kind" field says whether this object is a variable, a constant, a procedure, a type, ... The nature of the "further information" depends on the object kind, so it has to be set up using variant records or pointers to information kept elsewhere.

In a block-structured language like Modula-2, an important attribute of almost any programmer-defined object is the block to which it belongs. Rather than put the block number into the symbol table, it is may be better to have a separate symbol table for each block. (This point is controversial, however; some experts would recommend one unified table.)

It is less obvious whether you should have separate symbol tables for names imported from other modules. Making those tables separate can simplify handling the IMPORT declaration, but on the other hand it can complicate checking the scope rules. (For example, it is illegal in Modula-2 to have a variable which is global to the module and which has the same name as an explicitly imported variable; and it is easiest to pick up "duplicate name" errors if the duplicate names are being inserted into the same table.) My own preference is to have a separate table when the programmer uses the "IMPORT ModuleName" form of import, but to use the "ordinary" symbol table for names imported via the FROM construct.

For data types, it simplifies matters to have a separate type table, as discussed in Part 2 of these notes. Then if the source program contains a declaration like

```
TYPE abc = SET OF CHAR
```

the symbol table for this block need only hold the information that string number 54 is the name of type number 27. The dictionary shows that string number 54 is "abc", and the type table shows that type number 27 is "SET OF CHAR".

You might be wondering by now what attributes should be stored in the symbol table for variables, procedures, constants, and so on. I feel quite strongly that that sort of question should not even be answered during the preliminary stages of the design. (And I use that philosophy for all my programming, not just compiler-writing.) All you need to say for now is that "the information stored is the information that will be needed". As you fill in the details of the various modules which make up the compiler, you will gradually discover what is needed, and at that point you can make arrangements to have it stored. The alternative approach − of designing the fine details of the data structures before developing the procedures − can lead to a situation where you store things which are *not* needed; or, more commonly, where you store information in a form which is inconvenient for retrieval.

The question of whether to store machine-dependent information in symbol tables is a delicate one. (Machine-dependent information includes things like the number of bytes used to store a variable.) In principle you should not do it, because the front end of the compiler should not make any assumptions about the target machine. In practice, you might find yourself forced to do it. As a general rule, you should make every attempt to keep machine-dependent detail out of the symbol table.

One of the most difficult problems here concerns storing the values of constants. The character string form of a number is machine-independent, but its binary representation can depend on the target machine. For example, if you have

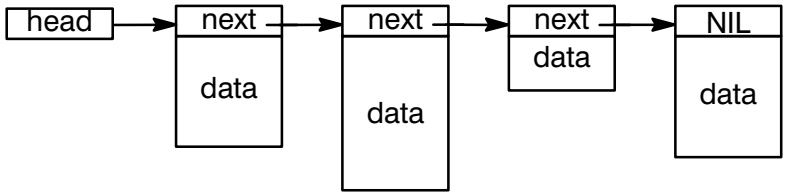
```
CONST a = 65535;  b = a + 1;
```

then the value of b is 65536 on some machines, but is 0 on machines which use 16 bits to represent a CARDINAL (and this should result in an error message). Such complications become even worse when REAL numbers, with their associated overflow/underflow/rounding properties, are considered. For this reason, some compiler designers believe that numeric constants should be stored as strings, rather than as binary numbers, in the front end of the compiler. Unfortunately, this complicates the compile-time evaluation of constant expressions. My own tendency is to cheat and use a binary representation for constants anyway; and to document the fact that this is an area which will need care when porting the compiler over to produce code for a different target machine. Of course, all arithmetic on constants should use the rules of the target machine, not the machine on which the compilation is being done. This requires, for example, that you do constant additions with a function `Add` rather than with the built-in + operator, in order to isolate the machine dependencies in an identifiable module.

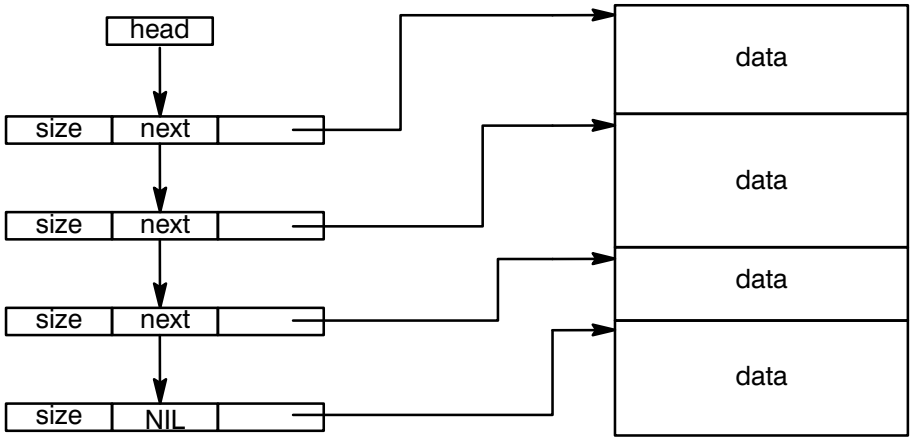
3.4 Separating Information and Access Methods

As a general principle when designing the implementation of data structures, you should make a distinction between a *table* (or other similar structure) and the methods for accessing that table. Although this sounds like a rather abstract statement, I can illustrate it with a concrete example. In order to satisfy the scope rules for Modula-2 and similar languages, you need to arrange symbol searches such that, in the event of several objects having the same name, the one you find is the innermost one in the chain of blocks surrounding the currently active scope. One way to do this (not the only way!) is to have a separate symbol table for each block, and then to link the currently visible symbol tables as a stack. Now, the traditional way to make a stack of large objects is to include a "next" pointer in each object, as illustrated in the following diagram.

The catch with that approach is that the "next" pointer is visible to the module which deals with data type "Symbol table", when logically it belongs to an entirely different data type "Stack of symbol tables". To get



around this, we can reorganise the implementation to separate out the "next" pointers, as shown below.



The advantage of this new structure is that the module dealing with symbol tables does not need to know about details which do not concern it. Furthermore, it simplifies the treatment of the access problem. It is now easy, for example, to make the necessary changes if you decide that you really want a tree of symbol tables rather than a linear list of symbol tables.

3.5 Fast Search Methods

There is just one basic operation to be performed on a symbol table: find a named object, insert a new entry if it isn't already there, and return the properties of the object. (Of course, you might find during implementation that you don't want to implement it precisely this way, but that doesn't change the point I wish to make here.) Since the compiler will be doing this frequently, it is highly desirable to arrange your tables in such a way that they are easy to search.

The two best-known good methods are binary search and hash addressing. (Look these up in just about any text on data structures if you don't know what they are. If you are sufficiently keen, you might also want to read up on various more esoteric methods.)

Using binary search, the table is kept sorted by the name (actually a number, in most implementations) returned by the dictionary. This does not mean that the table which actually contains the data has to be sorted. Following the principle of Section 3.4, you might choose to have one (unsorted) table containing the data, and a separate (sorted) table of pointers to the first table. This creates some extra space overhead, but the overhead can be justified if it brings other advantages. It allows you to design the access table for efficiency in sorting, while arranging your main data tables in ways which satisfy other priorities.

The hash table approach is usually better in terms of speed. To use this approach, set up a hash table as an array of pointers, where each pointer points to a list of objects which map to the same hash code. Again, notice that this separates the "access" data from the main data.

Whatever method you use, deleting entries from a table can cause complications. If these complications turn out to give too much trouble, you can consider associating a "deleted" flag with each datum, so that a deleted entry physically remains in the table – so as not to mess up your search algorithm – until it is overwritten by a new entry which can conveniently go into the same table slot.

3.6 Saving Space in a One-pass Compiler

In a one-pass compiler, and even in some multi-pass compilers, the only symbol table information which the compiler needs is information about the symbols in the current block and any blocks which surround it

(including the outermost global block). Once the compiler reaches the end of a procedure, it can throw away all information concerning the interior of that procedure. (The procedure heading needs to be kept, but that information belongs with the symbols of the next outer block.) Throwing away this information not only saves space; it can also simplify decisions on how to organise the data structures, and as an added bonus it can reduce the time spent on searching tables.

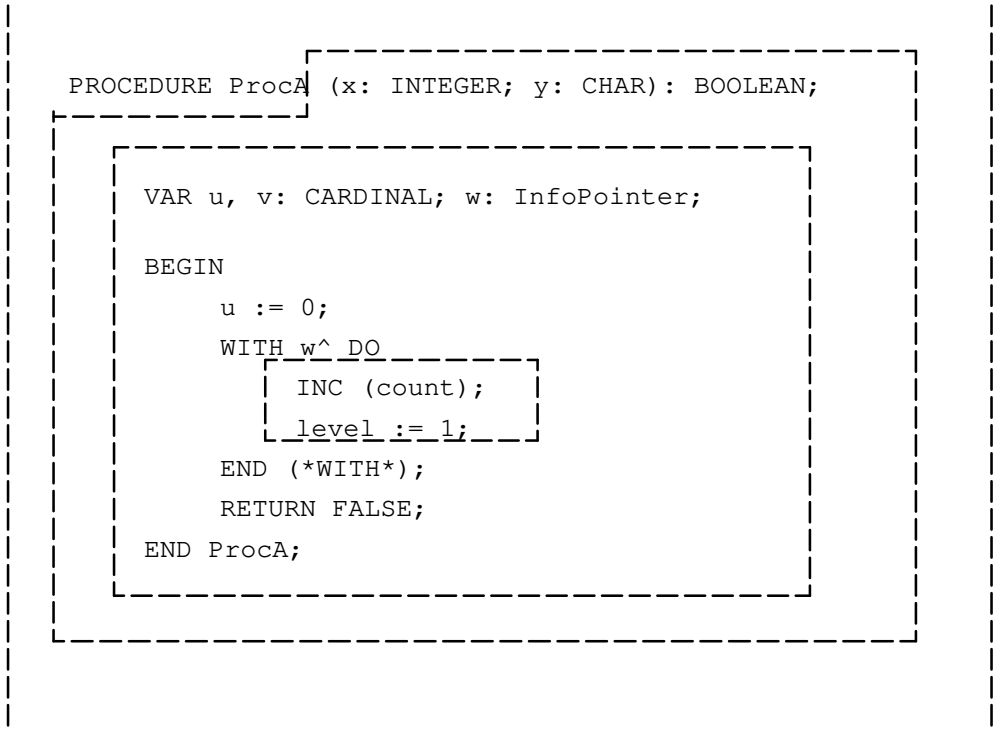
If a separate symbol table is kept for each block, it is obvious how to discard obsolete data. In all other cases, you will have to think carefully about how to do the deletions. These details should, of course, be hidden inside an implementation module. The external interface can be as simple as a couple of procedure calls `StartBlock` and `EndBlock`.

The dictionary can also benefit from being allowed to delete obsolete entries. Remember, however, that the dictionary module knows nothing about the block structure of the program. I would suggest that the dictionary module have a pair of `Mark` and `Release` procedures, which can be called from the parts of the compiler which do know about block structure. The function of `Release` is to discard anything which has been entered into the dictionary since the last `Mark`. However, you should not make it *compulsory* for `Release` to discard anything, because (a) in the early stages of compiler development, it is simplest to set up `Mark` and `Release` as procedures which do nothing; and (b) even in the final version, it might be more efficient (depending on the data structures used) for procedure `Release` to discard only part of what it is allowed to discard, retaining the data which would be difficult to delete.

3.7 Further Comments on Block Structure

To the programmer, the block structure of a Modula-2 module is relatively straightforward. There is one enclosing block per module, which contains all the declarations − including procedure headings − made at the top level. Nested within this is one block per procedure or nested module. The block nesting properties are defined by the way the procedure and module declarations are nested inside one another.

The compiler should, however, invent a few more imaginary blocks to simplify the handling of some of the language rules. Outside the outermost visible block there is an invisible block containing things like the predefined types and the built-in procedures and functions. Internally, it is convenient to assume some extra blocks as shown in the following example. The dashed boxes show the blocks.



At the outer level, we have the procedure definition, which includes the types of the parameters but not the names of the parameters. The definition is processed by the compiler as if it had been written

```
TYPE P = PROCEDURE (INTEGER, CHAR): BOOLEAN;
VAR ProcA: P;
```

(although of course the original programmer did not have the option of writing it this way). Inside the procedure, notice that the parameters are given a separate block of their own. This is not essential, but it is helpful because (a) the processing of the parameters has to be a little different from the processing of the other declarations, and treating this part as a separate block helps to isolate the special cases; and (b) if we think of a block as something which contains some declarations and some executable code, then having a separate block for the parameters gives us a logical place, in the design of the compiler, to look after that part of the procedure entry and exit code which deals with parameter and result passing.

A separate block is created for a WITH statement for a different reason. The declaration

```
VAR x:    RECORD
               a, b: CHAR;
          END (*RECORD*)
```

implicitly defines two CHAR variables called `x.a` and `x.b`, with `x` being stored in the symbol table and `a` and `b` being stored in the type table. Inside a WITH statement, the names `a` and `b` can be used as if they were variable names rather than field names. The easiest way to implement this is to open a new block for the WITH statement, with local variables `a` and `b` whose attributes are copied from the attributes of `x.a` and `x.b`. (Curiously, this is very similar to what happens in the "unqualify" operation when importing "export qualified" names. I don't know whether this observation leads to any practical applications.)

Nested modules can create some extra problems. (A nested module is a module declared inside another module.) If you have the nested module

```
MODULE Example;
     EXPORT w;
     EXPORT QUALIFIED x;
     ...
END Example;
```

then you have to deal with three symbol tables. The object named w goes into the symbol table of the enclosing block, to reflect the fact that it is exported. The object named x is also defined in the scope of the enclosing block, but you need a separate symbol table for it to remind yourself that in the enclosing block its name is Example.x whereas inside the module its name is simply x. Finally, you need a new symbol table for all of the non-exported names.

In fact, I am inclined to believe that a similar approach is helpful in dealing with all modules, and not just the nested ones. That is, for every module you can create two symbol tables (or a single symbol table with two parts): one for the exported objects (provided that qualified export is used) and one for the non-exported objects. (But if you do this, make sure that the symbol search procedure is set up to look in both tables.) As for non-qualified exports, these form a special case in that their symbol table entries do not even belong to the module; they belong to the enclosing block.

Notice, in this connection, that a non-qualified export is possible only from a nested module. An earlier version of Modula-2, now obsolete, allowed non-qualified exports from definition modules, but this nasty feature has now been withdrawn – possibly because the early compilers were failing to handle this case correctly.

Notice too that a symbol table search never needs to go beyond module boundaries. A nested module does *not* have access to the names of the enclosing module, except for the names which it imports – and the way to handle an import, as mentioned earlier, is to copy the imported information into the current symbol table.