# 6 Symbol tables

## 6.1 -- Theory

The prior chapters of this text have discussed the theory and implementation of increasingly complex machines for handling increasingly complex language subsets of a programming language. The scanner or tokenizer can be implemented as a simple state machine, since it need only locate and classify regular expressions. The parser operates on a context-free grammar, and can be implemented as a push-down automaton. Just as context-free grammars are more complex than regular grammars, so parser theory is more complex than tokenzier theory, and the resulting parser is more complex than the tokenizer.

The next step in our logical progression is context-sensitive grammars. Unsuprisingly, this chapter will address some of the context sensitive issues which appear in any programming language. However, since context-sensitive grammars are extremely complex, the discussion will not be based on context-sensitive grammars, but rather on semantics. (There is a difference between a context-sensitive grammar and semantics. The grammar will only tell you whether a sequence of symbols may have meaning or not – it will not tell you WHAT that meaning is.  For example “A+B”  might mean add the value stored in the integer variable A to the value of interger variable B, or it might mean concatenate the string A with the string B, or it might mean add Matrix A and B, or it might mean that A and B are getting married, or that A loves B, or mix chemical A with chemical B, or …)

In any case, the symbol table is a useful abstraction to aid the compiler to ascertain and verify the semantics, or meaning, of a piece of code. It will keep track of the names, types, locations and properties of the symbols encountered in the program. The type system and the code generations system rely on the symbol table to provide them with information about symbols encountered elsewhere in the code. It makes the compiler more efficient, since the file doesn’t need to be re-parsed to discover previously processed information.

This chapter will focus on two of the most important functions of the symbol table in a compiler. First, they cache useful information about various symbols from the source code program for later use during code generation. Second, they provide the type checking mechanism that determines the semantic correctness of a program.

### 6.1.1 Symbol tables for Information caching

Here we introduce the question of what information needs to be cached and where do we store it?  One difficulty we have an answering this question is of course that the information we need to store depends on the language, and on the architecture to which the language is being compiled.  In this book we discuss statement driven declarative languages like ALGOL, C++, or Pascal.

The first step in designing the symbol tables is to decide what information needs to be recorded.  Generally we only record information about identifiers. Most tokens in the program, such as keywords or symbols, have a fixed meaning, so we need not record information about them. Identifiers, on the other hand, do not have a fixed meaning. They can represent different constructs or have different semantics in different files -- and even in the same file. In order to determine what their meaning is in context, we must record information about these identifiers for use elsewhere in the compilation process.

Having determined that symbol tables are useful, the question invariably arises, “How does one build the symbol table?” There are two questions hidden in this seemingly innocent inquiry. First – what information needs to be stored? Second – how will the information be organized.

#### 6.1.1.1 What should be stored?

Exactly what information is stored in the symbol table depends on many things. The programming language will determine much of the information that is stored, but the target architecture will also influence what data is stored. In fact, some assumptions about how to produce code can effect what values are stored in the table. Rather than attempt to list all the possible structures and values that might appear in a symbol table, this section will provide an general overview of the classes of information typically stored.

When one considers the range of information that can be stored about identifiers, it seems logical to classify identifiers in terms of the constructs they represent. In other words, different information will need to be stored for constants, variables, procedures, enumerations, type definitions and so on. What follows is a description of various common declarative language constructs and typical classes of information symbol tables would record for those constructs.

**Constants**: Constants are identifiers that represent a fixed value - one that can never be changed. Since programmers will wish to access these values by name, the name must be stored. Naturally, the corresponding value must also be stored. Finally, since the values must be used properly in the type system, type information is also included. Unlike variables or procedures, no run-time location needs to be stored for constants. These are placed right into the code stream by the compiler at compilation time.

**Variables**: Variables are identifiers whose value may change between executions and during a single execution of a program. They represent the contents of some memory location. Obviously, then, the symbol table needs to record both the variables name, as well as its allocated storage space at runtime. Typically this location is stored as an offset relative to some position (which can be turned into an absolution location at link or load time). Just as with constants, their type information should be recorded for later use by the type system.

**Types (user defined)**: A user defined type is typically a conglomeration of 1 or more existing types. (Some languages allow you to define recursive structures, but SAL does not). Types are accessed by name, and reference a type definition structure. Each structure will record important information about itself, like its size, the names of its members (if it’s a record) or its upper and lower bounds (if its an array). What information is stored will depend on what “kind of” type is being defined. See arrays, records and classes in the next section (page ???)

**Sub-programs (aka Procedures, Functions, methods)**: Procedures (etc.) are named segments of code. Naturally, the symbol table should record a procedure's name. The type they return (if any) should be noted. When procedures (etc.) are accessed at run time it is typically by their location in the code stream (or some handle to that location) - thus the location of the code generated for a given procedure should also be recorded.

The (formal) parameters and local variables of a function are separate identifiers (variables) in their own right, and should be stored in separate records. Thus, they are treated much like the fields of a user defined record. They are stored in as a list of variable records separate, but accessible from, the main procedure record.

***Class***: Classes are abstract data types which restrict access to (encapsulates) its members and provides convenient language-level polymorphism. They are really a special case of user defines types, and are structurally no different. But it may be convenient to store information about classes above and beyond that required for other user defined types. This includes the location of the default constructor and destructor, and the address of the virtual function table.

***Inheritance***[s1]: There are many different ways to perform inheritance, and a symbol table record is needed to keep track of exactly how inheritance is performed. A compiler might consider whether *shared* or *non-shared* inheritance is specified for a given class. (see chapter XXX for an in-depth discussion of shared and non-shared inheritance). In C++ the keywords *public*, *private* and *protected* modify the visibility of inherited items, and may be recorded with the inheritance information. A reference to the participating classes (parent and child) could also be recorded in an inheritance structure.

Table 6.1.1 summarizes the data field requirements for symbol table records that store information about various identifier declarations.

| | Name | Scope | Type | Size | Offset | Value | Other |
|---|---|---|---|---|---|---|---|
| Variable | 4 | 4 | 4 | 4 | 4 | - | |
| Constant | 4 | 4 | 4 | - | - | 4 | |
| Type | 4 | 4 | 4 | 4 | - | - | |
| Function | 4 | 4 | 4 | 4 | 4 | - | return type |
| BaseClass | 4 | 4 | - | 4 | - | - | inheritance method |

***Table 6.1.1 Common Symbol Table Information Related to Identifiers***

Languages that allow programmer to specify arbitrarily complex types will need some dynamically extensible system for defining types. It is also convenient to encapsulate the "variable-sized" information about program constructs like subprograms. The most intuitive method involves making "trees" of user-defined-type records. … The remainder of this section discusses the general class of records that would store complex type information.

**Array**: Arrays represent a collection of uniformly typed elements that may be randomly accessed by index. For each dimension of an array, the compiler will need to know about: the lower boundary of the array (the lowest valid index), the upper bound (the largest valid index), the index size, index type, the total size, and the type of the elements contained. When many different types can be used to index and array, the index type and size will also be recorded. Finally, the total amount of space to be allocated for each dimension of an array should be stored.

**Records**:  Records represent a collection of possibly heterogeneous members which can be accessed by name. Most importantly, each of members needs to be recorded. However, we also need to know the size of the record (how much space to allocate for all of the members). Each of the fields of a record will probably be a reference to another symbol table record – like a variable or a type – which may in turn reference another record or array. In this manner, very complex user-define structures can be created.

**Sub-Programs**: In many languages procedures are not a user defineable type. However, their parameters and local variables are often stored the same way fields of a record are. It is often convenient to treat this aspect of a procedure the same way records are. A structure for storing the "complex" information about procedures would record the parameters, the local variables, and the amount of stack space occupied by a procedure at run time.

**Class**: Just like a record, the fields of a class can be conveniently stored in a separate record. Classes will also store their methods, constructors, destructors and virtual function table in this complex-information structure.

**Module**: Stores the model size, its name parent this is another one of those exceptional records) its members, and a time stamping-the time stamp is used to guarantee that load time that the models have been compiled in the correct order or are all up to date.

Table 6.1.2 summarizes the data member requirements for symbol table records that store data about complex types and complex language constructs.

| | Name | Member Names | Size | Index Type | Element Type | Other |
|---|---|---|---|---|---|---|
| Array | - | - | 4 | 4 | 4 | array index bounds |
| Record | - | 4 | 4 | - | - | |
| SubProgram | - | 4 | 4 | - | - | |
| Module | 4 | 4 | 4 | - | - | |
| Class | - | 4 | 4 | - | - | |

***Table 6.1.2 Data fields for Complex Type Information***

#### 6.1.1.2 Symbol table Organization

There are two major activities to relate the symbol table, namely: inserting values and accessing stored information. Value insertion occurs primarily when the parser processes declarations, while the value retrieval occurs elsewhere, generally in the expressions and statements of the language.

*Value insertion operations include creating a record for a new symbol, assigning values of fields of the records and making adjustments to the internal scoping.* Value retrieval operations consist of a searching backward to the scope to find an identifier by name, examining the formal parameters of a procedure to match them with the actual arguments and, examining values in a found record to compare with actual values found in the code. It will be useful in discussion the underlying implementation(s) for a symbol table to think of it as an abstract data type, in terms of the data it stores, and the manipulations (or functions) that may be applied to the data. The data it stores is a collection of the various records we discussed in the previous section. The typical operations, and the ones that will affect the underlying data structures are:

1) Find a record by name (and type) (access a variable as in "`x = 5`")
2) Find a record by name (and type) in a specific scope (access a field of a record as in "`a.x = 5`")
3) Find a record by its relationship with another record (type of a variable, the parameters of a function)
4) Insert a new record
5) Update an existing record

We can split the symbol table implementation into two layers where each layer represents a certain data abstraction. The bottom layer is an associative array, and is concerned with storaging records  and retrieving records by name. This corrseponds to operations 1,2 and 4 in the previous list.  The top layer organizes the records into groups and determines how the scoping of the language is maintained. It is responsible for operations 3 and 5 in the previous list, as well as providing convenient access to the operations of the lower level.

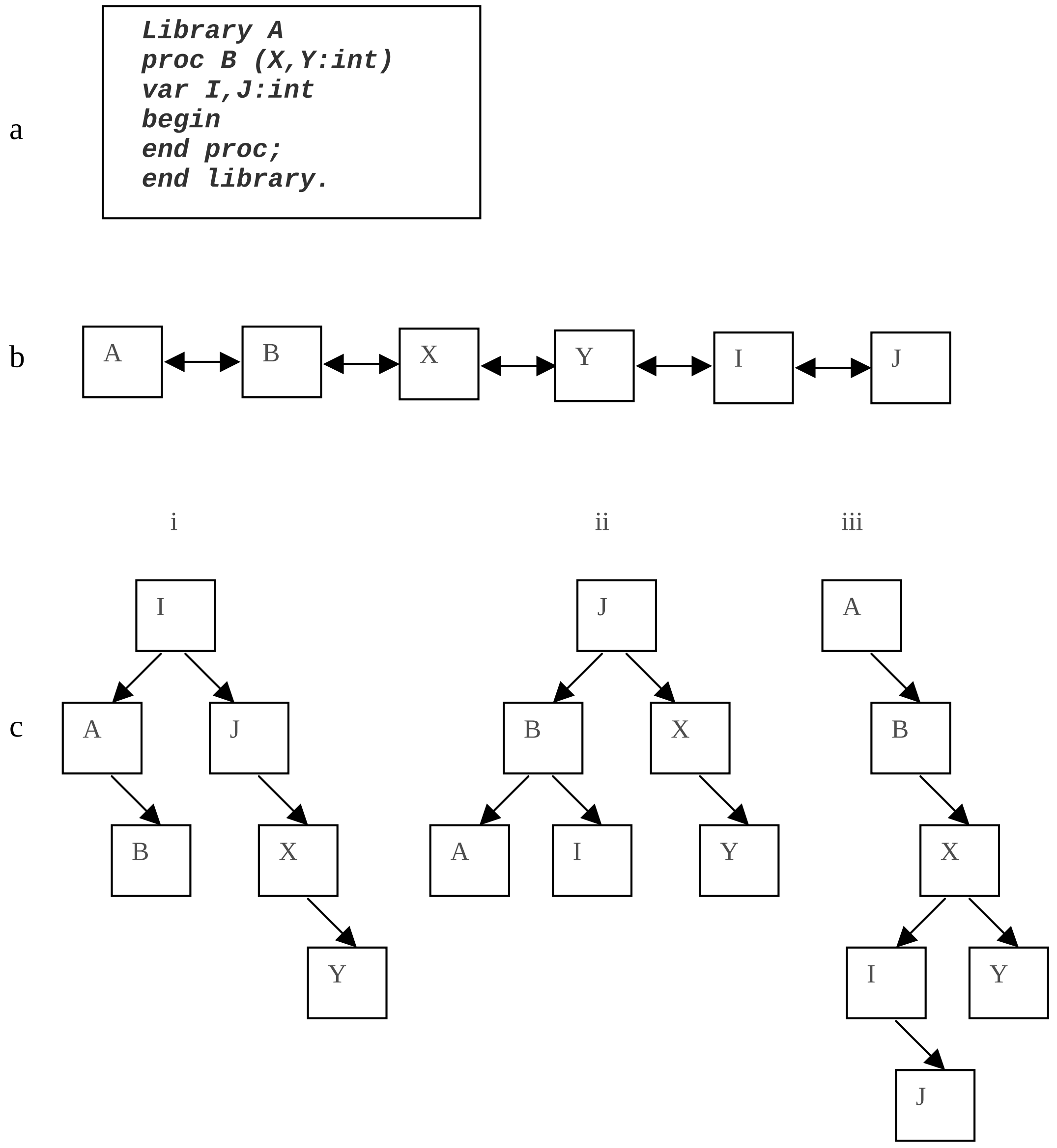
The bottom layer of the symbol table is solely concerned with the storage and retrieval of symbol table records *by name*. It has two basic operations: add(insert) and retrieve (find)[1]. There may be many varietyies of find: find by name alone, find by name and construct-type, find by name and type, find by name in a limited scope, etc.  This is also the description the abstract data type “associative array”. A convential array addresses records by number, an associative array addresses records by label.

Unlike convential arrays, associative arrays have no obvious implementation on modern computer architectures, so symbol table designers will need to develop algorithms to implement this abstract data type. There are many different data structures which provide functions to add, find (and remove) records using labels as an index. These include linked lists, binary-trees and hash tables. Some are easy to implement, some provide efficient methods. Which implementation a particular compiler uses will depend on the needs of the individual compiler and compiler writer.

The first option we discuss is also the easiest to implement: an unordered linked-list. Each record in the linked this would store its name, the other relevant record data, and a pointer to the next (and previous) record in the list.  The add operation merely places new symbol table records at the end of the list in the order they are encountered. The retrieve operation is equally simple. This list is traversed, and the first (or last) record matching

[1] There may be occasions when a remove feature may be useful in the bottom layer. In  a mult-pass compiler a remove function is not needed, because all of the symbols will be needed for future passes. A single pass compiler may not need a remove function either, as long as the computers on which it will run have sufficient memory to hold the entire symbol table for all programs they  compile. However, it can be covenient to remove symbols from the table when they are no longer in scope. (For example, once a single pass compiler has finished processing a procedure block, its local variables can never be accessed anywhere else in the program – therefore they can be safely removed). This textbook does not cover removal in the implementation section.

the retrieval criteria[2] is returned. The list is usually traversed from back to front under the assumption that the more recently declared symbols are more likely to be the used in a program. On the plus side, linked lists are easy to implement and insertions can be performed in constant time. On the negative side, searching the list occurs in linear time.



```
Library A
proc B (X,Y:int)
var I,J:int
begin
end proc;
end library.
```

***Figure 6.1.1 – A library(a) and two methods of storing the symbol table (b) as a linked list and (c) as a binary tree***

[2] In SAL this is always just the name, but some languages may be more particular. The symbol table could be asked for records for a certain type of construct (variable, function, type, etc) and certain type (int, array, etc.) or for something in a specific scope (local, global, within a class, etc). Perl, for example, will distinguish between functions, arrays, hashes and scaler types, even if they have the same name.

Figure 6.1.1 a and b shows the library and resulting symbol table structure for lists.

Since searching for information by name is one of the most common operations for a symbol table (much more common than record inseration), a better approach is often desired.

The next logical step is to move to a binary tree. Search times for binary trees are proportional to log(n). The algorithm for binary trees is explained in any data structure book. By way of quick review: all nodes to the left of a given node occur lexicographically before that node, and all nodes to the right occur after. To find a node, one simply compares the target name to the node name and takes the left or right child accordingly. Unforunately, in order to add a node to the correct place in the tree also requires the tree to be traversed to find the spot where the new record should go (to maintain the ordering of the nodes). This means that the insertion time is the same as the lookup time, roughly O(log(n)) (Try searching the tree in Figure 6.1.1 (i))

Even worse, the O(log(n)) time is based on a *balanced* binary tree (like the one in figure 6.1.1 (ii)!. If the nodes are inserted in near alphabetic order, the basic binary tree algorithm will produce a tree that looks like a linked list. Figure 6.1.1 (iii) shows the tree produced by the basic algorithm on our example program. As an additional solution, many algorithms exist which keep binary trees approximately balanced, at the cost of additional algorithm complexity. (Knuth 1973, AVL Trees in the data strcutres book,….)

When speed really becomes an issue, symbol table implementors turn to hash tables; most large scale or professional compilers take this route. In the best case, hash tables give a constant lookup time (O(1)) and a constant search time. Worst case time depends on the particular hash table implementation, but can be as bad a linear insertion and linear lookup if all the names hash to the same location.

To add a new item to the symbol table, one must apply the hash algorithm to the name of the identifier. Use the number returned by the algorithm as an index into the table. If they slot is empty, insert the element. If the slot is already filled apply the collision algorithm. (Most collision algorithms will repetiviely compute a new hash value, and check a new slot until an empty one is found.) The collision resolution algorithm we present appends the record to a list of records stored at that location.

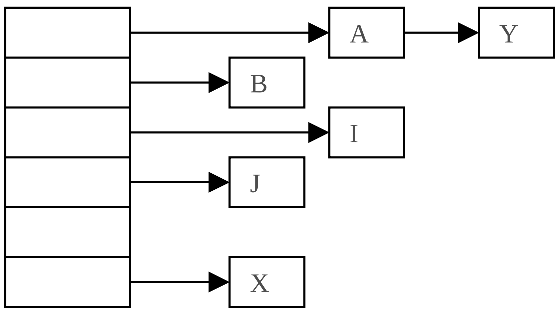


*Figure 6.1.2 Hash table organization of a symbol table*

To lookup a name in the symbol table, one uses the same approach used for insertion, only instead of checking if the slot is empty, check if the slot contains a record with the correct name. (e.g. hash the name, check the slot, if it is empty, no record exists, if it contains a record with the correct name, recturn it, otherwise apply the collision algorithm, and repeat the process)

There are many hasing algorithms and many collision algorithms. We present two fairly simple and fairly effective ones:

Hash algorithm:  (sum of letters % table_size)

Collision resolution algorithm: create a bucket (a list or binary tree of elements) which all hash to the same slot.  Search time is generally O(n/m)  where n is the number of symbols and m is the table size. Figure 6.1.2 shows one possible arrangements of the identifiers of the code of figure 6.1.1a into a hashtable using the bucket collision resolution scheme.

On the plus side, hast tables are fast! They provide a “constant” insertion time and a “constant” lookup time. On the downside, hash tables require space to be pre-allocated. A lot of space can be wasted if the hash table is too big, or have a lot of collisions if the hash table is too small..

## *Scoping*

There are at least two popular approaches to deal with scope in structured languages. In this book we’ll call them “scope-by-number” and “Scope-by-location”. As their names imply, one uses values (numbers) stored in each record to record the scope of the identifier, while the other uses the structure of the symbol table to determine scoping. (It’s kind of an explicit  vs. implicit representation issue)

Let’s look briefly at the purpose of scoping in the symbol table. In general, programming languages allow variables (and other named constructs) to be shadowed by other variables (…) of the same name contained in “nested” blocks. This means that when a compiler looks up an identifier by name it has to retrieve the type with the correct name that was declared most recently within the nested blocks currently visible within the language. The following code segement, Listing 6.1.1, will illustrate:

```
library scoping;
var a,b,c:int
type x is record
   a,b,c:int;
end record
proc outer()
      var a,b:int;

      proc inner1(c:int)
      var b:int;
      begin  ...  end proc;

      proc inner2(a:int)
      begin
      a:=b+c; // ←------
      end proc;
end proc;
end library.
```

**Listing 6.1.1 Sample SAL code to illustrate searching within scope**

When the compiler processes the expression  pointed to by the arrow, it should select the underlined variables. Name the a that is a parameter to inner 2, the b that is a local variable of outer, and the global variable c. The b and c from inner 1 should be skipped because they are not in the set of nested blocks visible at the arrow, the a in outer is shadowed by inner2's parameter. The a, b and c from type x are outside the nested scoping too, and the a and b in scoping are shadowed by the a and b in outer.

The scoping layer does not add operations to the table, rather, it determines how the find method is implemented. That is, it adds another layer to this lookup function which can find the "correct" identifier from among all the identifiers with the same name.

The first method (scope-by-number or single table) stores all the identifiers in the same table. When mutliple identifiers have the same name, they are stored in a common bucket. Each nesting block is assinged a number by the compiler. All identifiers declared within that block store that blocks number as their scope. Thus, the scope of identiifers stored in the same bucket can be distinguished by the scope number. When searching for the correct identifier, a stack of scope numbers (representing the nested blocks), along with the name is used to determine which identifier should be retrieved. Figure 6.1.3 shows one the code of listing 6.1.1 might appear in a "scope-by-number" strucutred symbol table.

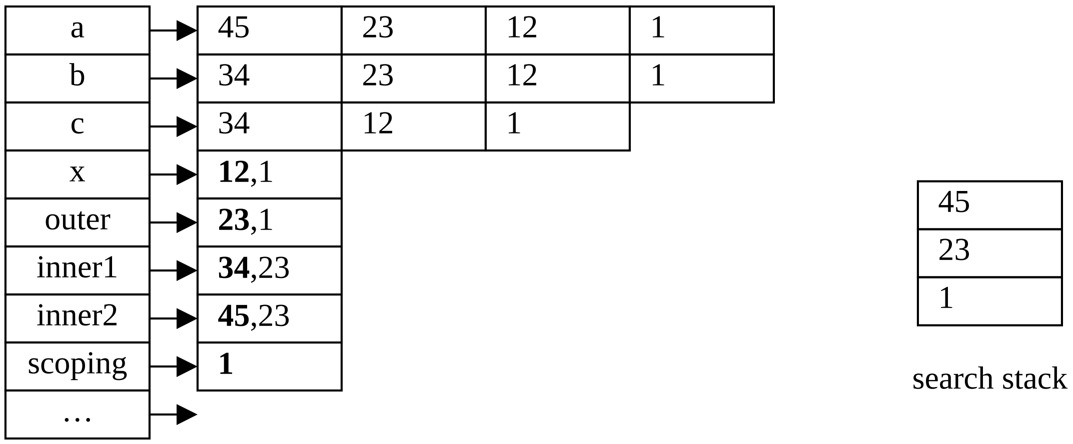


*Figure 6.1.3 Symbol-table scoping in a simple hash table. Bold items indicate the definition of a scope number, normal font items represent the recording of a scope number. In this case, when the compiler searches for c in the expression a:=b+c, it will find c in the hash table, skip the first and second entries in the bucket because their scope number does not appear in the search stack, and accept the last entry because the scope number 1 is also in the search stack.*

The second method (scope-by-structure or nested tables) creates separate tables for each scope block in the language. The block strcutred scoping approach (demonstrated in Figure 6.1.4) will be the apporach taken by the implementation section of this book.

New records are inserted into the data structure (bottom layer) for the current level of scope. When a new level of scope is introduced (we enter a new block -- be it record, class or method), a new bottom layer is started that is referenced by a record in the previous (outer) list. No buckets are needed. Instead a more versatile "find" routine is required.

The find routine checks for records in the associative array for the current level of scope first. If it finds a matching identifer, that one is returned. Otherwise it recursively searches the next (outer) layer of scope until either an appropriate record is found or an error if none are encountered before the outermost layer is exhausted.

If no records are deleted, the scoping layer creates a tree of lower layer records that very closely resembles the structure of the orignial program.

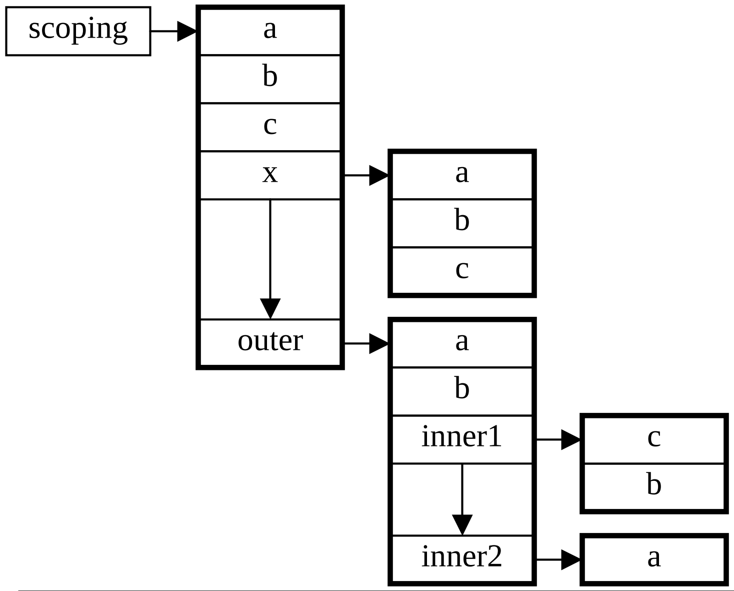


*Figure 6.1.4 Example of a table using block structured scoping.*

### 6.1.1.3 Filling the Symbol Table

The implementation of the two layers described in the previous section will provide the insert and find methods through which the compiler will interact with the symbol table. So, the question of how information is added to the table is partially answered – through the interface provided by the symbol table. The whole answer is not so simple. Before we answer the question in full, let us first examine the question of when information is added to the symbol table.

In a one-pass compiler, the operations on the symbol table must be interleaved with the parsing of the program. The time to add a new record to the table is when it's construct-type is known. (e.g. we can't create a variable record until we know a variable has been declared). Invariably, information about a language construct will be parsed both before and after the construc-type can be identified. Since this information still must be recorded, the parsing functions will need to take some extra effort to route this requisite information to where it belongs.

In the first case, information about a construct may appear in the source code before the type of the construct can be identified. Consider the declaration ***`int x;`*** in C. Until the parser encounters the semicolon, the compiler cannot tell whether x is a variable or a function. By the time it has encountered the variable, the datatype for that variable (int) has long since been processed. This means that some information will have to be stored in a temporary location until it can be used.

In the second case, additional information about a construct may not appear until after the record for that construct has been created. In SAL and Pascal the syntax for variable declarations is ***`var x :int;`***. We know that x is a variable as soon as its name is encountered, and its symbol table record should be created immediately. The datatype will be encountered later in the source code, and x's record will need to be updated with its type information.. In this case, a reference to the variable should be stored so that it can be updated when more information becomes available.

### 6.1.2Symbol tables for Type Checking

There are many levels of program correctness. One level is syntactic correctness, which is achieved when a program is sucessfully parsed. Due to the limitations of context-free grammars, there are still many mistakes that programmers may make which cannot be caught by the parser.

#### *Static vs. Dynmaic typing*

A second level of correctness is provided by type checking. If the compiler can verify, *at compile time*, that a program is guaranteed to be free from type errors, then the compiler has performed *static* type checking.  When every program in a language can be statically type checked, the language is said to be strongly typed.

Many languages can provide a lot of type checking at compile time, but must leave the checking of some types until execution time.  Still other languages leave all the type checking until execution time. Both classes of languages should be called dynamically typed languages, (although many languages in the first group still claim strong typing). Dynamically types languages require that some or all of the symbol table be accessible at run time. Statically typed languages may discard the symbol table before execution time.

Typing (static or dynamic) supports the seperation of specification from implementation [Horowitz, 118]. It gives a name to an abstraction, so that the user of the datatype can interact with all the features of a type – through their declarations – without needing to know all the details of its implementation.

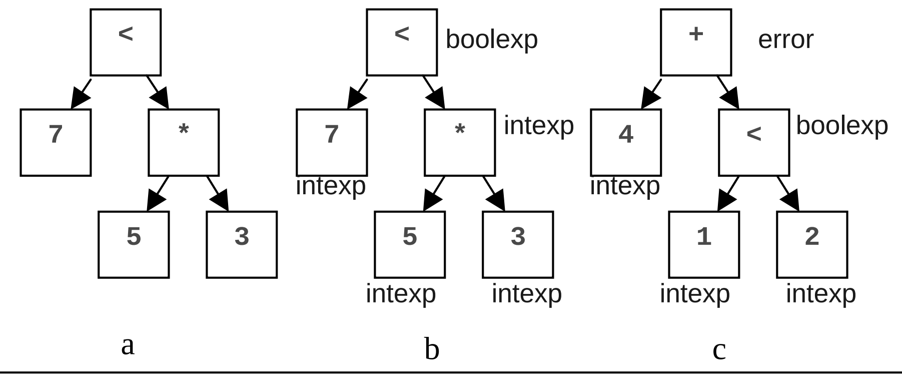


*Figure 6.1.5 labelling parse trees with types*

The important question to ask is, how does the compiler verify that a program is type correct? We answer by giving the theory behind syntax directed type-checking.

Every program has a corresponding parse tree, which represents the syntax of the program. The compiler then assigns some type-value to every leaf in the parse-tree. It then algorithmically applies transformations to the leaf nodes to derive the type-values for the internal nodes of the tree. If the process completes by assigning a non-error type-value to the root node of the tree, the program is type-correct. The remainder of this section will give illustrative examples of how to assign type-values to tree nodes, and how this process will interact with the rest of the compiler.

In this first example (Figure 6.1.5 a and b), ***7<(5*3)***, we deal only with literals. Notice that while all the literals in this example are integers, the root type for this tree is a boolean. This occurs because not all operations on a integer produce an integer. In fact, this very feature is what allows the compiler to reject such statements as ***4+(1<2)***, (Figure 6.1.5c) because there is no valid type-operation for adding an integer to a boolean.

**Explicit and Implicit Type conversion.**

Already, some readers are thinking "But, my favorite programming language will let me do that!" Many languages improve upon this simply type system by providing for conversions or coersions between types. In this scenario, the compiler or type system has some mechanism for changing the type of an expression to another type. If the programmer is required to write extra code to have this conversion performed we call this an "explicit" type cast or type coersion. If the compiler performs this transformation without direction from the programmer, the term is "implicit" type conversion. Typically implicit conversion, if done at all, is done only on the "built in" types. Typically, user defined types are too complex for the compiler to be able to decide what an appropriate conversion might be.

Suppose our language allowed for the explicit conversion of a boolean to an integer. The resulting tree for the expression 4+int(1<2) might look like figure 6.1.6a:

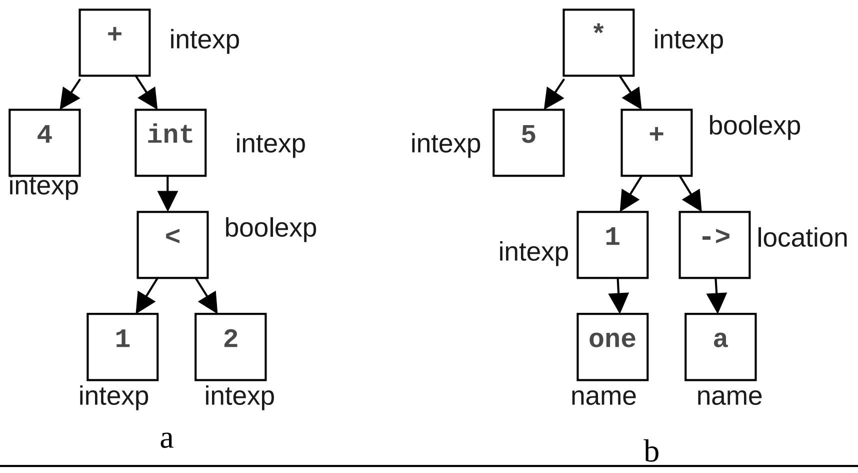


*Figure 6.1.6 – examples of type conversion*
*(a) explicit type conversion*
*(b) implicit type conversion*

The next interesting problem comes with the introduction of names, or abstractions. When an item becomes named, such as a constant, its name and type are bound in the environment of the program. In a compiler for a strongly typed languages, the environment is stored in the symbol table. When that name is used in the program, the type system examines the environment (symbol table) to retrieve the type bound to the name. Now that the name has a type associated with it, the type system can assign a type to that node in the tree and continue verifying. For example, if ***`one`*** were the constant 1 and ***`a`*** were a variable, then the type tree for the expression ***`5*one+a`*** would appear as shown in figure 6.1.6b.

*Scaler vs Structured*

So far we have dealt only with scaler types. A type is scaler if its domain consists only of constant values. Typical scaler types are boolean, character, integer, floating-point and fixed-point numbers. Fortunately, structured types are no more difficult to handle in a type system than are abstracted scaler types. A structured type is one whose domain consists of members which are themselves composed of scalar or structured types. Records, arrays, classes and methods are examples of structured types.

The type system treats each name as a lookup into the environment, and the member access operation as a function mapping the types of the child nodes to a type for the parent node. Consider the expressions ***`4+x.a`*** (where x is a record with a member a), or the expression ***`(2<arry[5])`***, where arry is an array of integers, and 5 is the index into that array. Figure 6.1.7 shows how the corresponding parse trees may be decorated with types.

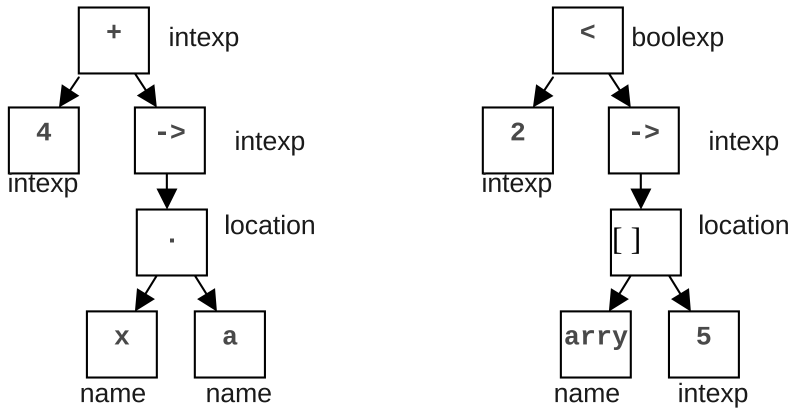


***Figure 6.1.7 – type decoration for structured types***
***– arrow indicates that a location is being dereferenced and turned into a value***

*Complex Type Equivalence*

Since structured types are often named, the question arises, "How do we decide if two complex types are equivalent." There are two popular answers to this question. One is that they are equivalent only if their names are equivalent. This is the approach taken by Pascal. The other answer is that they are equivalent if their structure is equivalent.

Whichever approach is taken, however, the type system still processes the code in the same manner, all that changes are the effects of operations on complex types.

*Implementation overview*

Implementing such a type system, then, is just a matter of allowing each method of a recursive descent parser, (or its equivalent state in a top-down or bottom-up parser) to determine the type of its corresponding syntax tree node. This determination is made based on information retrieved from the symbol table, and from the type-values returned by its children nodes.

The remainder of this chapter is concerned with filling in the symbol table. For specific direction in ading type checking to the SAL compiler, see the Annotated Grammar Diagrams in Chapter 7 – Expressions[DS2].

## 6.2 Symbol tables -- Implementation

The symbol table is a key contribution to the semantic processing of a language. Wherever a name is used in the source code, the symbol table must be consulted. This includes the definition,use and validation of user-define types, resolving a variable into its address, resolving constants into their values, calling functions, matching function arguments to function parameters, and so on.

However, before the information can be used, it must first be stored. This chapter focuses on specific structures and methods to properly populate a symbol table for the SAL language. The process of filling the symbol table will occassionally require that the symbol table be search to record semantic relationships between symbols in the program.

Consider a variable declaration. In the variable declaration ***`var x: foo`*** we say that a new variable called “x” has type “foo”. In order to find out what type “foo” is and store it in x’s type field, we have to look it up in the symbol table.

Thus, filling the symbol table will involve several important tasks.

1. Defining records to store the information.
2. Organizing the records into scope-blocks for easy retrieval at a later point.
3. Identifying the points where information should be recorded, looked-up, cached and propogated to properly populate the table.

At the conclusion of the chapter students should be able to sucessfully organize and populate a symbol table for any subset of the SAL language.

### 6.2.1 Organization

The first step in defining the symbol tables is to decide what information to store. In section 1.1.1 we introduced some of the popular imperitive programming language constructs and gave examples of information that would commonly be stored in the symbol table for each of those records. This information is summarized in table 6.2.1:

| | Name | Scope | Type | Size | Offset | Value | Return Type | Members | Index Type | Element Type | Upper Bound | Lower Bound | Inheritance Method |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Constant | X | X | X | | | X | | | | | | | |
| NamedType | X | X | X | X | | | | | | | | | |
| Variable | X | X | X | X | X | | | | | | | | |
| Function | X | X | | X | X | | X | X | | | | | |
| Module | X | | | X | | | | X | | | | | |
| Record | | | | X | | | | X | | | | | |
| Class | | | | X | | | | X | | | | | |
| Inheritance | | | | | | | | X | | | | | X |
| Array | | | | X | | | | | X | X | X | X | |

*Table 6.2.1 Example of fields appearing in various symbol table records*

It is evident that much of the information needed is common to many types of symbol table records. In order to leverage this repeated information, we made the symbol table records C++ classes. We then factored out the repeated information into two base classes, namely: **Ident** for all identifier types, and **aggregate** for user-defined complex type information. Further, some of the columns of this table represent quite complex information: specifically type and value. Separate classes were made to encapsulate this information as well.

The figure 6.2.1 graphically illustrates one possible class hierarchy (and the one used in the support libraries) to encapsulate the information required by the symbol table, while taking advantage of the sharing of fields between records.

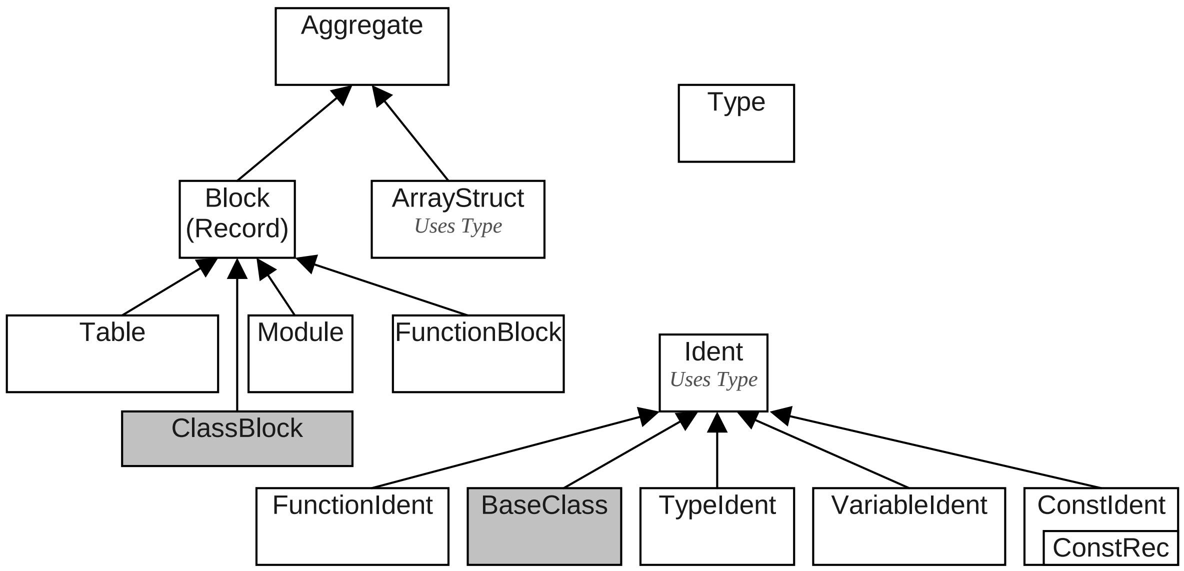


*Figure 6.2.1 The class Hierarchy for the symbol table classes.*

Ident represents the common information for most declarations. Its children are TypeIdent (for user-defined type declarations), VariableIdent (for variables) ConstIdent (for named scaler constants), FunctionIdent (for procedures, functions and methods) and BaseClass (for storing inheritance information.) BaseClass stores the name of one base class, and can be used when searching the table for scope resolution operators (see section ?.?)

Aggregrate represents common information for structured types. Its children are ArrayStruct (for statically sized arrays) and Block (for records). Since classes and functions also need to record information about who their members, ClassBlock (for classes) and FunctionBlock (for procedures, functions, and methods) inherit the fields of Block.

The remaining three classes provide additional support, but do not inherit from Ident or Aggregate. Type represents the information used by the type-checking system. ConstRec holds the actual scalar values for ConstIdents. Table is a class to provide a single access point to the entire symbol table.

We talk about each of the classes in this hierarchy in greater detail throughout the remainder of this chapter. Each section will include a small box summarizing the fields and records relevant for that aspect of the symbol table.

#### 6.2.1.1 Structure of the table at compile time

We have shown the relationship between symbol table records based on the information they record. However, it is also important, and probably more important, to see how the records are related in memory. In other words, once instances of the classes are created, how are the instances organized to store and retrieve information about symbols in a program.

For pedagogical reasons, we have chosen to use an unsorted linked-list as the underlying “associative array” implementation, and a “block-structured” approach to scoping (scope-by-location). The structure of the resulting symbol table very closely resembles the structure of the code it represents. Figure 6.2.2 shows the general structure for any symbol table we can construct.

It starts with a single table object, which represents the entire program or library being compiled. In SAL, every file represents an abstract data type called module. Correspondingly, the table houses a list of TypeIdent records, where each TypeIdent refrences a module.

Each Module record contains a list of all the identifiers declared within that module (or file). These identifiers may refer to types, variables, constant values or functions. Conveniently, each of these language constructs is recorded by a class which inherits from Ident, so the module just stores a list of Ident records.

Since users may declare variables and new types of intrinsic or complex types, the VariableIdent and TypeIdent records may refer to complex type information (records, arrays, and classes). If they are simple types, the pointer to complex information will be NULL.

Functions and procedures, on the other hand, always point to the “complex” information about the function or procedure. Thus, each function is split into two records, one about the “name” of the function, an one about the “block” or scope modifying attributes of the function. The block part (FunctionBlock) records information about the member declared within it’s scope: parameters, local variables, local types, local constants, and nested procedures and functions. These are all stored in a single list inside the Function Block.

Records, like procedures, record their membership in a list of Identifiers. Since records only contain their fields (which are variables), only VariableIdent records will ever be stored in a Block or Record object. Arrays, on the other hand, have no named fields. Still, they may also need to record more complex information (if you have an array of records, or an array of arrays…). Arrays may then reference Blocks (representing records) or another ArrayStruct.

Classses also contain a list of their members – variables and methods. They must also reference any classes they inherit from. In this book, this is accomplished by including a BaseClass record in the list of member which names the base class and points to the ClassBlock of the parent class.

The remainder of this chapter presents each of the symbol table classes in detail. We present the grammar rules in which the records are created and modified. We show sample code and the corresponding table structures (with values). The presentation will occur roughly in the order they may be created during program compilation.

A few more notes about Figure 6.2.2. First, a dotted arrow represents a link between a "Block" record and an "Ident" record, while a solid arrow represents a link bewteen an "Ident" record and an "Aggregate" record.

Second, there are some solid and dotted arrows on the right side of the figure that point nowhere. These represent the fact that the table is recursive. Taking the leftmost arrows from top to bottom we see that 1) ArrayStructs may point to another ArrayStruct, Block (or nothing). 2) Block reference another list of Idents, 3) VariableIdents (that are members of a Block or record) point to more ArrayStructs, Blocks,ClassBlocks, or nothing. 4) Variables (that are parts of a class) point to more ArrayStructs, Records, ClassBlocks (or nothing), 5) FunctionIdents point to another FunctionBlock, 6) ClassBlocks point to a list of VariableIdent, FunctionIdent or BaseClass, 7) TypeIdents (in a FuctionIdent record)

Thus an arrow leaving an ArrayStruct may point to any record or array.

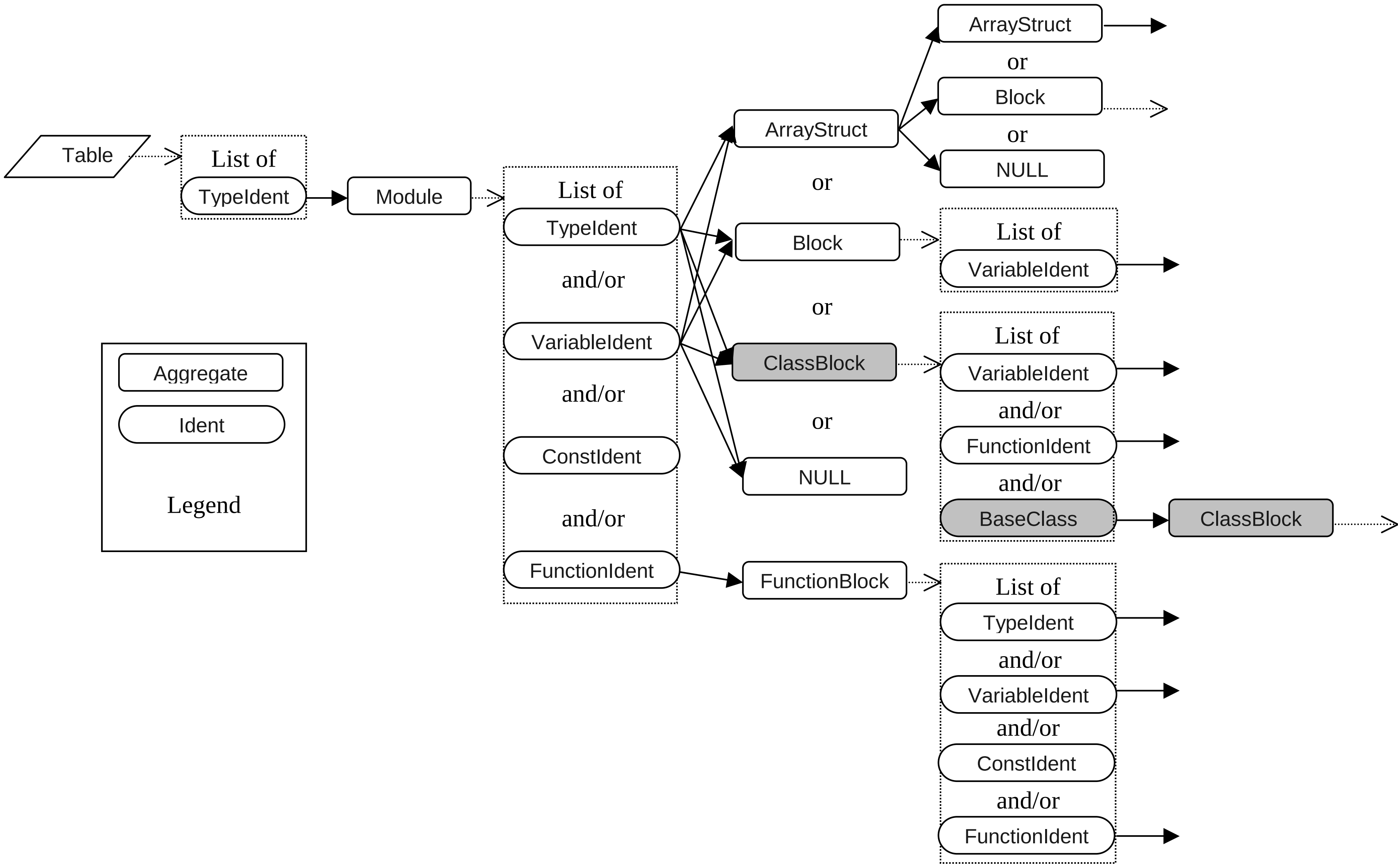


*Figure 6.2.2 Referential relationships between Symbol Table structures. This does not represent a symbol table, but is a road map showing how the elements (structures) of the symbol table relate (or may relate) to each other.*

### 6.2.2Table[s3]

The table class represents the symbol information for the entire program or library. It (and its members) contains all the symbol information gathered by the compiler. It oversees the organization and structure of the symbol table.  It facilitates searching, storing and printing. While poor in variables it is rich in functionality.

The primary purpose of the Table class to provide a convenient interface to the various datastructures in the table. It also caches some convenient information, such as the current block of scope, and the number of modules (or files) in the library. The data structure of the Table class is shown, in figure 6.2.3 with the most frequently accessed fields underlined.

```
class Aggregate
  DWORD   Size;
  Ident * Parent;
  WORD    Block_mod_number;
  WORD    Ctor_proc_number;
  WORD    Dtor_proc_number;
  bool    was_exported;
```

```
class Block : public Aggregate
  List < Ident >  Elements;
  Block *Pred;
  WORD Function_Level;
  WORD Scope_Level;
```

```
class Table : public Block
  Ident         * cur_ident;
  WORD            totalMods;
  WORD            ModNum;
  Stack <Block>   block_use_stack;
  Block         * cur_block;
```

*Figure 6.2.3 Data members of the Table class*

#### 6.2.2.1 Building the table

The first major activity relating to the table is to fill it. For this purpose a number of similarly formed functions have been provided to add records to the overall table structure, while helping to maintain the correct structure. These are broken down into 2 sets: append<Identifier> and append<Aggregate>.

For each Identifier there is an append function - exactly the following:

```
 ConstantIdent *appendConstant ( Alfa _name = "");
 VariableIdent *appendVariable ( Alfa _name = "");
 FunctionIdent *appendProcFunc ( Alfa _name = "");
 TypeIdent     *appendType     ( Alfa _name = "");
 BaseIdent     *appendBase     ( Alfa _name = "");
```

Each function inserts a record of the appropriate type (ConstIdent, VariableIdent, ProcedureIdent, TypeIdent and BaseIdent respectively) into the table in the current block of scope. Since we’re using unordered lists, the record is added to the end of the appropriate list. There is a separate list for each block of scope (because we’re using the block-structured scoping approach). The table keeps track of the current scoping block, so it always knows which list to use.

When the records are added, each append function fills in the record with initial default values. It initializes the name field to whatever parameter is passed in ("" by default). Any field that the programmer is required to manage (these are all specified in the descriptions of the individual records) are set to a default value of 0 or NULL as appropriate. Some of the fields of the various records can be set by the append<Type> function based on information cached in the table. The function returns a pointer to the newly created record.

For each Aggregate there is also an append<Aggregate> function.

```
Module        *appendModule   ( );[s4]
Aggregate     *appendRecord   ( ); [s5]
FunctionBlock *appendFunction ( );
ClassBlock    *appendClass    ( );
ArrayStruct   *appendArray    ( );
```

Each of these functions adds a new aggregate record to the symbol table. The most recently declared identifier will point to the new record. The functions also return a pointer to the newly created record. As with identifiers, the records values are initialized – 0 or NULL for field the student must fill in. The initialization values are discussed in the section about the individual records.

With the exception of Arrays, each of these Aggregates defines a new layer of scope. When the append functions are called, the Table class will also update the cached “current scope” field so that future identifiers are added in the new Module, Function, Class or Record. However, the Table class will not know when to leave this scope. Users must call LeaveScope() [s6]when leaving the scope of this block! LeaveScope causes the Table class to point to the layer of scope in which the current layer of scope was declared.

```
void         LeaveScope( );
```

#### 6.2.2.2 Searching in the table

After the symbol table has been built (and occasionally during its construction) the task turns to finding things stored in the table.  There are a number of function provided to aid searching. They are discussed below.

```
BOOLEAN       isDuplicate ( Ident *t )
```

This function searches only through the current block for an identifier whose name field is identical to the name of the identifier passed in. If one is found it returns true. This is most often used when declaring identifiers - you want to be certain no duplicate identifiers are found in the same scope.  Remember that modules do define scope, so two variables with the same name in different modules are allowable (they can be distinguished by the syntax ::module::identifier).

```
Ident *findInScope ( char *name, Block *scope ) ;
```

The findInScope function looks for an identifier record with the same name as *name*; but it only looks in the scope of the block specified by *scope*. What we mean by “in the scope of the block” depends on whether we’re talking about a Object-Oriented language or not.

For a simple structured language this means only looking through the identifiers contained in the block's list of elements. When classes and inheritance this will be more complicated. First, it searches the list for the current scope. If it is in a method with a self parameter, it then searches the list of the class block the self parameter points to. It also examines (recursively) the BaseClasses of that class.

```
Ident          *find ( char *name )
Ident          *find ( char *name, List<Ident *> &blocks)
```

The find function looks through all the blocks of scope that are currently visibile as defined by the language. Specifically, it starts with the last identifier in the current block and searches to the beginning of the block (by calling findInScope). If no identifier is found it moves to the block in which the current block was declared, and so on.  If nothing is found in the current module, it will also search the (exported) global identifiers in all preceding modules. If no identifier is found, it returns NULL.

The second parameter, List<Ident *> & blocks,  is used for handling classes with mutliple inheritance. It allows the function to return a list of the names of the blocks it had to traverse to find the specified symbol. (This will be helpful during code-generation.)

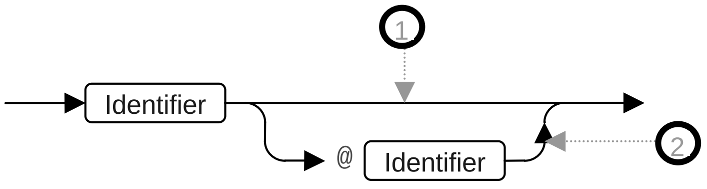
We now look at the QualIdent grammar rule where most of the symbol table searching will occur, given in figure 6.2.4.

#### 6.2.2.3 QualIdent[3]

Ident * QualIdent(….);

This function returns the identifier named in the token stream

Returns NULL (or throws exception) if the identifiers is not found.



**1)** table.find(…);  // Find identifier starting in current scope.

**2)** table.findInScope(…); // Find identifier starting the specified module

*Figure 6.2.4 ParserRule QualIdent*

#### 6.2.2.4 Accessing table status

There are a few functions for getting the current state of the table. These are:

```
Module         *getCurrentModule()
Block          *getCurrentBlock ( )
WORD           curFuncLevel ( )
WORD           curLevel ( )
```

The following four functions are useful for traversing the lists of element. They each manipulate the pointer to the current element kept by the List <Ident*> class of the current block.

***Ident        *goToLast  ( )***

Moves the internal pointer to the last element of the current block, and return a pointer to that last element.

***Ident        *goToFirst ( )***

Moves the internal pointer to the first element of the current block, and returns a pointer to that first element.

***Ident        *goToPrev  ( )***

Moves the internal pointer to the element preceding the current element of the current block, and returns a pointer to the new current element.

***Ident        *goToNext  ( )***

Moves the internal pointer to the element following the current element of the current block, and returns a pointer to the new current element.

The table stores its status in a few member fields. These are

***ModNum***: the number of the module currently being compiled

***Level***: how deeply nested the current function is

***Elements***: (inherited from block) is a list of (pointers to) records which inherit from Ident, for each module in the current program or library.

The table class represents the symbol table for the entire program (or library) currently being compiled – including the imported (and already compiled) modules (a.k.a files) . The table then stores a list of all these imported modules, and places the module being compiled at the end of the list. A module is a type, and like any other complex type, it is composed of two pieces, the Identifer and the Aggregate parts. Thus, we present the TypeIdent class, its parent Ident, and the Type class, in preparation for discussing modules, and the rest of the symbol table records.

### 6.2.3 Identifiers and TypeIdent

Identifiers provide abstraction to a programming language. They allows programmer to refer *by name* to values, types, memory locations, and code segments, among other things. Later, programmers can change what the name refers to, without having to modify the rest of the program. Instead, the compiler will make the necessary changes in the symbol table, and the code. The Ident class stores information common to the various constructs an identifier may represent. The Ident class does not represent any construct that a programmer can produce in SAL, and is simply a common information repository for the classes which inherit from it.

Conveniently, user defined types (in our system) do not require any more information than Ident records do. So, our discussion about the Ident class is also a discussion of the TypeIdent class.We present the data members and their associate access methods for this class here. One should remember that these field and methods will reappear in the child classes.

***`Alfa Name`***

The most important field (and probably the most often accessed) is the name of an identifier. Accessed to the field is provided through the class constructor (to initalize the name) and by *setName(Alfa)* and *getName()*. In SAL, identifier names are case sensitive and limited to 16 characters.

***`BOOLEAN Export`***

This field indicates whether or not this identifier can be accessed outside this module and is accessible by *getExport()* and *setExport(BOOLEAN)*. Since it is semantically incorrect to export anything from a program module, this will only be set in library modules. When this flag is set, the table will automatically record the information for this symbol in the .rll object file produced.

***`WORD Mod`***

Mod may be accessed by *getMod()*. This is the number of the module this variable was declared in. This field will be intialized by the table class, as long as they are added with the Append<Ident>() functions in the table class. This field caches information needed for generating code. Specifically, when the source code accesses a function or variable declared in different file (or module) the module number is needed by a linker to determine its location at run time. (See the virtual machine instructions CX, LE? And SE?)

***`Block * Parent`***

Accessed by *getParent()* and *setParent(Block *).* This field should point to the block that this identifier was declared in. This field is part of the scoping layer.

***`Type type`***

The **type** field can be accessed by *Ident::getType()* and *Ident::SetType(Type &*). The Type class (not to be confused with TypeIdent - a class for working with type declaration identifiers) stores information about datatypes. It utilizes a number of enumerated types which are explained in the following section on the class Type.

***`ObjectType obj`***

This field is provided for convenience, and can only be accessed by *getObj()*. It is set by the constructor and cannot be changed. ObjectType is an enumerated type shown in listing 6.2.1.

```
typedef enum  {  // These should stay in this order:
  badobj = -1, constobj, varobj, typeobj, funcobj, baseobj
} ObjectType;
```

***Listing 6.2.1 The ObjectType enumeration***

Each class will have its own unique object type -- the mapping should be fairly obvious.

Most of the time it will be sufficient to manipulate Ident pointers throughout the code. Occasionally, however, it will be necessary to "upgrade" the pointer to one of the classes that inherits from ident. There are two methods to safely accomplish this.

The first involves the obj field; the second uses some convenient functions. Both are shown in Listing 6.2.2

The above code will safely convert an Ident pointer to a VariableIdent pointer. Converting to other types can be done in similar fashion.

***BOOLEAN was_exported***

This field is not accessible.  It is used to optimize what information is printed to the .rll files.

***BOOLEAN vis***

This field is not used. <* save it for private/public membership maybe *>[DS7]

```
VariableIdent *v;
Ident * i = some_identifer;
if (i -> getObj() == varobj)
   v = (VariableIdent *) i;
```

```
VariableIdent *v;
Ident * i = some_identifer;
v = i-> toVariableIdent();
```

*Listing 6.2.2 Determining the type of an Ident descendant*

## 6.2.4Type

The Type class is a major component of the type system. Every user-defined-type has a type, but there are other types besides. There are the intrinsic types which the compiler knows about without an explicit declaration (like bool, char, int, real, strings, etc). SAL also allows the defnition of anonymous types. For instance, in the declaration

```
var x,y:array[1to10] of char;
```

x and y have a type, but that type has no name, and there is no TypeIdent record for that type. The user cannot declare another variable of that same type later in the program, but both variables still have a type that must be recorded and checked when used. This type is recorded by the Type class.

The Type class has only four fields, shown in figure 6.2.5 and explained below.

```
class Type
{
   DataType type;
   SubType subtype;
   BYTE plev;
   Aggregate *extra;
};
```

*Figure 6.2.5 The Type data structure*

***DataType type***

The **type** field represents the “major” type information. Some languages have various sized integers, for example. ***type*** would record the fact that a variable is an integer, and leave the size of the integer for another field. ***type*** can be accessed by *Ident::getDatatype()*, *Ident::setDatatype(Datatype)*, *Type::getDatatype()* and *Type::setDatatype(Datatype).* The DataType enumeration, shown in Listing 6.2.3, covers the major datatypes in SAL: numbers, characters,  arrays, classes, etc.

```
typedef enum {
  notyp = 0,    booltyp = 1,   chartyp = 2,
  inttyp = 3,   cardtyp = 4,   realtyp = 5,
  arraytyp = 6, recordtyp = 7, classtyp = 8,
  proctyp = 9,  functyp = 10
} DataType;
```

*Listing 6.2.3 The DataType enumeration*

notyp is used to indicate errors. booltyp, chartyp, intyp (signed integers), cardtype (unsigned integers), and reatyp (floating point numbers) designate intrinsic types the user will not need to define.   The remaining fields are for complex structures that require user definition. Whenever one of the complex types are used (arraytyp, recordtyp, classtyp, proctyp, or functyp), the Extra field will reference the addition information that is required.

**`Subtype subtype`**

The subtype field represents the “minor” type, or variations within a basic type. This field gives additional type information and can be accessed by *Ident::getSubype(SubType), tIdent::setSubtype(), Type::getSubtype()* and *Type::setSubtype(Subtype).*  The **subtype** field is an accomodation to languages that allow for numeric types of varying sizes.  The subtyp enumeration, shown in listing 6.2.4, indicates the size (in bits) of the various intrinsic types, especially integer, cardinal and real, as well as their format (s for signed, u for unsigned and f for floating point).

```
typedef enum {
  nosubtyp = 0,
  _s8s  = _s8,   _s16s = _s16,   _s32s = _s32,  _s64s = _s64,
  _u8s  = _u8,   _u16s = _u16,   _u32s = _u32,  _u64s = _u64,
  _f32s = _f32,  _f64s = _f64,   _f80s = _f80,
} SubType;
```

*Listing 6.2.4 The SubType enumeration*

The names in the above enumeration are a bit cyrptic. The first character - s, u or f - stands for Signed (for integers), Unsigned (for cardinals) or Float (for reals). The number designates the number of bits used to represent the number, and the 's' stands for Subtype. In general, characters are ***_u8s***, integers (unless otherwise specified) are ***_s32s*** and real numbers (unless otherwise specified) are ***_f32s***[DS8].  The ***nosubtyp*** field is reserved for complex types.

**`BYTE plev`**

The ***plev*** (short for pointer level) field is only used in the Advanced Compilers class - it records how many levels of dereferencing a variable has. For example: an integer would have a ***plev*** of zero. A pointer to an int would have a plev of 1, and so on. *get/setPLev()* provide access to this field.

***Aggregate *extra***

Complex types require extra information to fully define their structure. The field '**extra**' points to this additional information. More information on the Aggregate class, and its children will be provided after the discusion of identifiers. For intrinsic types, extra should be NULL.

This field is where the real interesting stuff happens. Any information about a complex type (Record, Array, Class, Module or even Function) is referenced by the extra field. Referring back to the overall structure diagram (Figure 6.2.2), every solid arrow in the figure is from the extra pointer.

### 6.2.5Import/Export

In addition to accounting for semantic information within a single program, the compiler must gather symbol information from other files and disseminate its own externally visible symbols. This is accomplished via the object file.

After program compilation is complete, and when the object file is written out, all of the symbols which may be accessed outside the current module should be included in the object file, in a section dedicated to that end. The exact format of this section of the file is described in the appendix on the object file format. The ExportSym methods of the various symbol classes fill in the object file for the student.

If a source file contains an import statement, the compiler searches the disk for an object file whose name matches the identifier in the import statement. If no object file is found a semantic error occurs. Otherwise, the object file is scanned and the symbols recorded in the file are added to the symbol table, where they will be accessible for use within the module. (They will not be included in the output file) The ImportSym methods of the various symbol table classes perform the necessary parsing for the student.

### 6.2.6Modules

Modules form the backbone of the symbol table. If you look at the source code, notice that a module is an aggregate structure -- it can contain many other elements. This is reflected in the object inheritance for the class Module. Module inherits from class Block (the basic class for keeping lists of Identifiers) which inherits from class Aggregate. The important fields in Module are underlined in Figure 6.2.6.

***int Size***

The size field (inherited from Aggregate) can be accessed by *setSize(unsigned int)* or *getSize()*. It contains the sum of the sizes of all global variables declared in this module. So if there are 3 integers, a character and an array of 7 10 byte records, the size recorded would be 3*4 (for the integers) + 1 * 1 (for the character) + 7 * 10 (for the array) = 83 bytes.

***TimeStamp ts***

This field can be accessed with *setTimeStamp()*.  Indicates when this module was compiled. This field will eventually be stored in the object file (.rll or .prg) produced by the compiler to be used by the linker to guarantee the correct version of modules are used.

Since class Module inherits from class Block, you will also have access to

***List <Ident> Elements***

This field is accessed directly.  It contains all the Identifiers declared at the global level in this module. More information about how to use the List class see section 6.2.12.1.

```
class Aggregate
{
     DWORD    Size;
     Ident * Parent;
     WORD    Block_mod_number;
     WORD    Ctor_proc_number;
     WORD    Dtor_proc_number;
     bool was_exported;
};
```

```
class Block: public Aggregate
{
     Block *Pred;
     WORD Function_Level;
     WORD Scope_Level;
     List < Ident >  Elements;
};
```

```
class Module: public Block
{
     TimeStamp   ts;
};
```

*Figure 6.2.6 The Module data structure*

The rest of the inherited fields are not needed for the module data type, and are kept only to keep the Inheritance tree logically organized. The should be left as initialized by the constructor (NULL or 0 usually).

A structural overview of a sample module is given in figure 6.2.7. The table was generated from the code in listing 6.1.1.

### 6.2.6.1 Architectural Considerations

In order to understand how to generate values for some symbol table fields, it is necessary to understand parts of the architecture the compiler will generate code for. For instance, how a function accesses its parameters will depend on the architecture's parameter passing scheme. Thus, before detailing the values inserted into the symbol table, a brief digression must be taken to explain key elements of the target architecture.

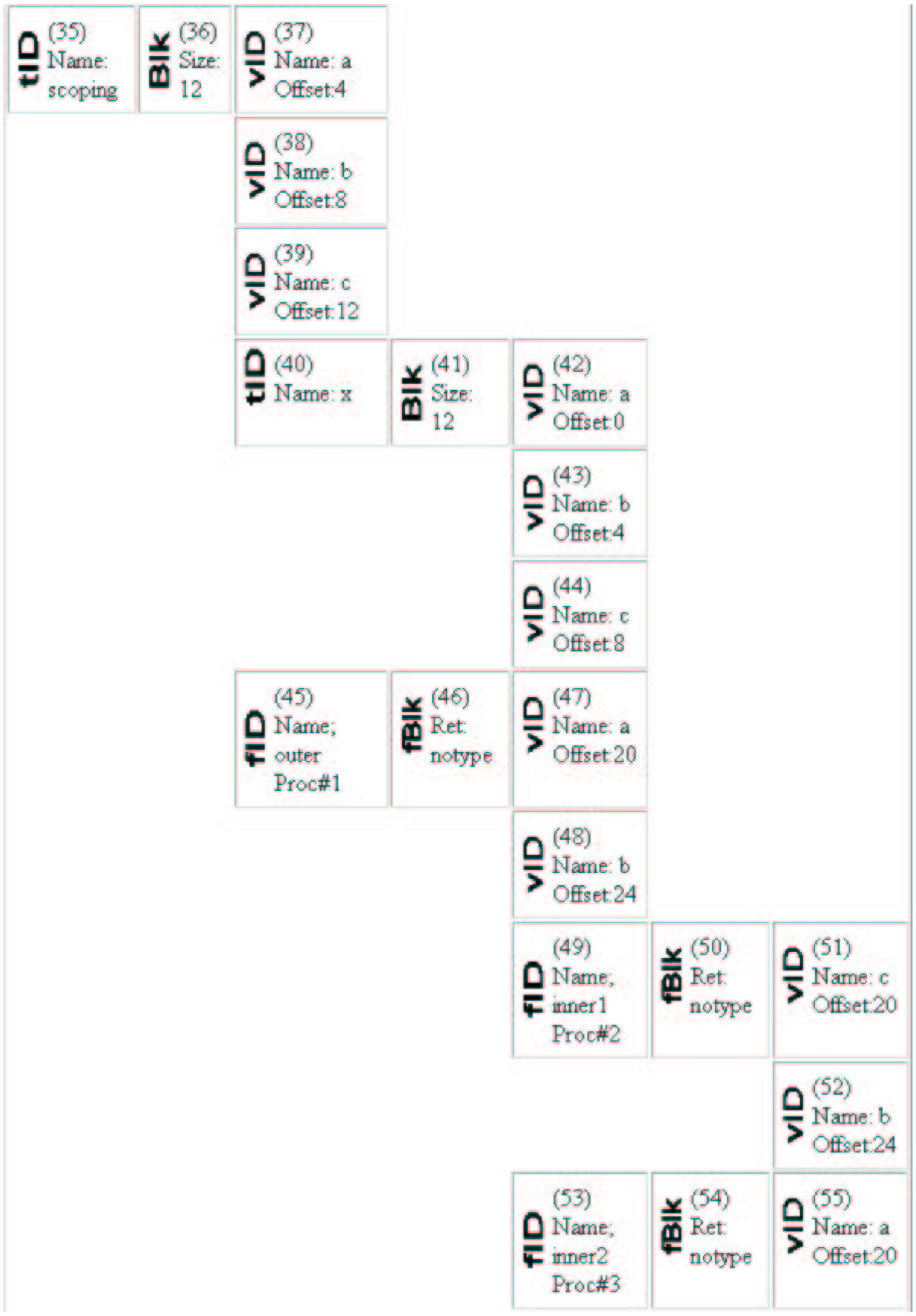

tID (35) Name: scoping
Blk (36) Size: 12
vID (37) Name: a Offset:4
vID (38) Name: b Offset:8
vID (39) Name: c Offset:12
tID (40) Name: x
Blk (41) Size: 12
vID (42) Name: a Offset:0
vID (43) Name: b Offset:4
vID (44) Name: c Offset:8
fID (45) Name; outer Proc#1
fBlk (46) Ret: notype
vID (47) Name: a Offset:20
vID (48) Name: b Offset:24
fID (49) Name; inner1 Proc#2
fBlk (50) Ret: notype
vID (51) Name: c Offset:20
vID (52) Name: b Offset:24
fID (53) Name; inner2 Proc#3
fBlk (54) Ret: notype
vID (55) Name: a Offset:20

*Figure 6.2.7 Sample symbol table (structural outline)*

The SAL Virtual Machine (SALVM) is a stack based architecture, which is to say that all low level computation occurs on an expression evaluation stack (EES), rather than in registers.

There is also a procedure stack - it records the run time values of parameters and local variables of a function, as well as the inter-function flow of control information. The "L" register (L for local) remembers the start of the currently executing procedure's data area. All local variable (and parameter) offsets are expressed relative to the L register. (Thus, there is no confusion using recursive functions at run time)

The "G" register remembers the location of all global data -- the data which, though declared in one module, may be accessed by all modules in a program.

The "S" register keeps track of the static string data for the program.

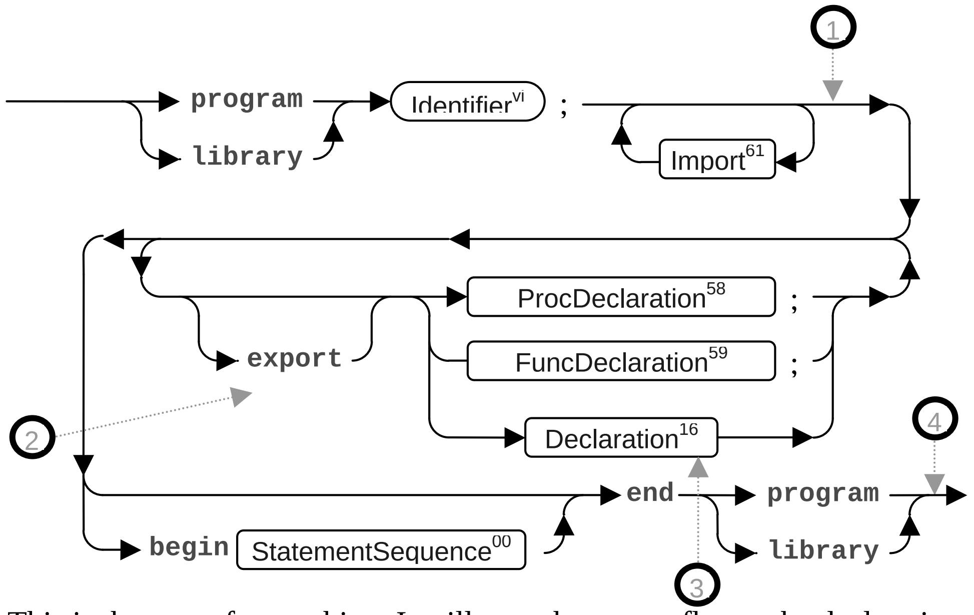
The SALVM uses 32-bit pointers for most operations (so parameters passed by reference require 32 bits.) Furthermore, for the implementation we descried, we assume that all records, arrays *and other complex types?* are made up of a pointer (reference) and content sections (this makes the parameter passing system more consistent) *This is similar to Java's approach.*

Procedures are stored by the SALVM in a lookup table -- one table for each module. So, instead of jumping direction to a PC offset, the SALVM jump instructions take a table index, and will fill in the proper PC at runtime. This means that you will store function indexes or function numbers instead of function offsets.

The way SAL is organized, each file corresponds to exactly one Module. Thus, the root grammar rule, CompilationUnit, is chiefly responsible for define the symbol table information for the file/module. We present the major module-constructions steps below in the context of the CompilationUnit grammar rule, figure 6.2.8.

### 6.2.6.2 CompilationUnit[63]

Void CompilationUnit();



This is the root of everything. It will pass the export flag to the declaration rules.

**1)** table.appendModule(…); //Add a TypeIdent and Module record to the table

**2)** Verify that this is not a program (only libraries can export )
Set an exporting flag to pass to the declaration
**3)** Left empty for alternative method of calculating size.
**4)** globalsize := currentBlockSize;

*Figure 6.2.8 Grammar rule Compilation Unit*

### 6.2.7Import/Export:

When a program is composed of multiple source code files, some system is required to communicate common elements between files. (There must be some coupling between files, or else they couldn't all contribute to the same program! In structured languages, this coupling will occur at the symbol level -- one file will access a type, variable, method or other named construct declared in another file.)

There are two common approaches for making information from one file available for other files. The first is the approach take by C, in which the names to be shared are declared in a declaration section (header file) which can be "included" in other files to describe the foreign symbols. The second approach, used by Java and SAL, is to store the "visible" names from the source code in the object file after compilation.

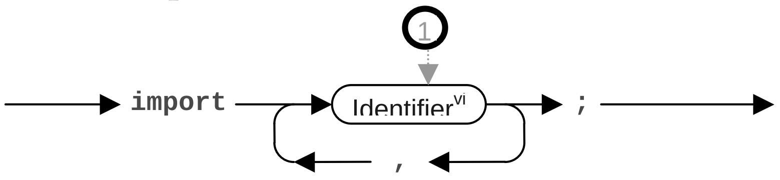
Using C's approach, the programmer is responsible for extracting the relevant symbols from each file -- the compiler merely reads the extracted declarations and processes them as if they were part of the file originally. The linker is responsible for matching up symbols used in multiple files.[DS9]

The second approach requires more work from the compiler. When a file is compiled, every externally visible identifier (and its properties) are identified by the compiler and stored in the object file. Then, when a file needs symbol information from another file, the compiler locates and scans the appropriate object files, and includes the symbol information in its symbol table for the current file. Naturally, the imported information is not replicated in the newly created object file.

SAL takes the second approach. On the plus side, programmers don't have to re-type all declaration information in a "declaration" section or file. On the down side, cyclic references between files are difficult to manage.[DS10]

The symbol table support libraries encapsulate the import/export process. The following grammar rule, figure 6.2.9, shows were the library function should be called.[s11]

#### 6.2.7.1 Import[61]



**1)** call LoadSymFile (*name of the module*)
(name can be accessed from the current token as token.data.id)

*Figure 6.2.9 Grammar Rule Import*

### 6.2.8 Constants

Constants are merely named values. The compiler treats these named entities exactly the same as their corresponding immediate values. They cannot be modified at run time. They do not occupy space outside of the code stream. In SAL they are limited to intrinsic types. The following three lines, listing 6.2.5, give examples of constants being declared.

```
const zero := 0;
      PI   := 3.1.4159;
const NULL := '\x00';
```

*Listing 6.2.5 Sample SAL code declaring named constants*

#### 6.2.8.1 ConstantIdent

The ConstantIdent class maintains information on named constants. In addition to the Ident fields: name, mod, parent, export, type and obj,  ConstantIdent requires one other field to contain the constant's value. The entire ConstantIdent structure is given in Figure 6.2.10.

```
class Ident
{
      WORD        Mod;
      ObjectType  obj;
      bool        was_exported;
      bool        Export;
      Alfa        Name;
      Type        type;
      Block      *Parent;
};

class ConstantIdent: public Ident
{
      ConstRec Value;
};
```

*Figure 6.2.10 ConstantIdent data structure*

`ConstRec value`

The **value** field stores the value of the constant referred to by this identifier. The field can be accessed by *ConstRec &getValue()* and *setValue(ConstRec &).* The ConstRec struct itself can be used in many other places in the compiler, so some of the fields are superfluous in this setting. The entire structure is presented in listing 6.2.5.

```
struct ConstRec
{
  DataType type;
  SubType  subtype;
  BYTE     base;
  union {
    BOOLEAN tf;   // the value as a True/False or boolean value
    BYTE    b;    // the value as 8 bits (unsigned)(think chars)
    WORD    w;    // the value as 16 bits
    DWORD   d;    // the value as 32 bits
    s64     i;    // the value as signed 64 bits
```

```
          QWORD   q;    // the value as unsigned 64 bits
          SINGLE  sp;   // the value as a 32 bit precision float
          DOUBLE  dp;   // the value as a 64 bit precision float
          TENBYTE tp    // the value as an 80 bit precision float
        }
      }
```

*Listing 6.2.5 The ConstRec record*

The type field is accessed directly, and uses the same enumeration as the Type class. However, because it refers to constants, it should only contain intrinsic types (booltyp, chartyp, inttyp, cardtyp and realtyp). Notice that in ConstantIdent, this field replicates ***`Ident::type.type`***. While it is safest to put correct values in both places, our implementation only looks at the field provided by the Ident class.

The subtype field is also directly accessible, and uses the same enumeration as the Type class. As with the previous field, all the code we've provided only looks at the ***`Ident::type.subtype`*** field for information.

Since raw numbers in the SAL source code are not typed, some assumptions will need to be made about their types (and subtypes) when they are stored in the ConstRec record. In this book, we assume boolean values are booltyp (_u8s), characters are chartyp (_u8s), all integer numbers are inttyp(_u32s), and all floating point numbers are realtyp (_f32). All number types are obtainable only by typecast.[s12]

The ConstRec class contains an anonymous union. Each of the fields in that union occupy the same space in memory. Because they actually share the same memory address, one can assign a value to one field as one type and retrieve it from another field as another type, without modifying the bit pattern.

This sharing of space can be useful. For instance, if you always assign values to q, the values for b,w,d and I will reflect the same value (barring overflow) and the high bits will be zeroed out for you. Unfortunately, the same trick does not work for sp, dp and tp. Make sure that your subtype aggrees with the variable you assign it to. (You'll need this trick during code generation to pass floating point values as paremeters to a function expecting integers.)

#### 6.2.8.2 Examples

Listing 6.2.6 provides some examples of declarations yeilding ConstantIdents. The corresponding ConstIdent record values are given in table 6.2.2

```
      export const w := 'a';   // #1
                   x := 5;     // #2
             const y := 23.15; // #3
      export const z := -x;    // #4
```

*Listing 6.2.6 Sample SAL code to illustrate constants*

| Field | #1 | #2 | #3 | #4 |
|---|---|---|---|---|
| name | w | x | y | z |
| export | TRUE | TRUE | FALSE | TRUE |
| Obj | Cnstobj | Cnstobj | cnstobj | cnstobj |
| type.datatype | Chartyp | Inttyp | realtyp | inttyp |
| type.subtype | _u8s | _s32s | _f32s | _s32s |
| type.extra | NULL | NULL | NULL | NULL |

|  |  |  |  |  |
|---|---|---|---|---|
| Value | b=’a’=96 | w=q=5 | sp=23.15 | w=q= -5 |

*Table 6.2.2 Sample values for ContantIdent records*

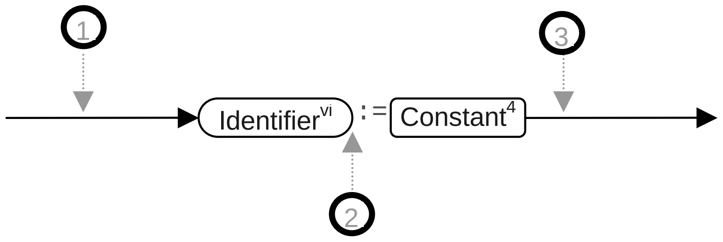
The folloing syntax diagrams help illustrate where in the process of parsing the symbol table values for constants are updated. In this case, two rules, Constant Declaration and Constant are involved. The first rule, given in figure 6.2.11, finds the name of the constant and creates the ConstantIdent record. The second, figure 6.2.12, extract the value of the number and its type, and hands that information back to ConstantDeclaration.

### 6.2.8.3 ConstantDeclaration[5]

void ConstantIdent(…, BOOLEAN Export);

Export tells if this Constant is being exported from a library (passed down from CompilationUnit)

This function adds an identifier to the table.



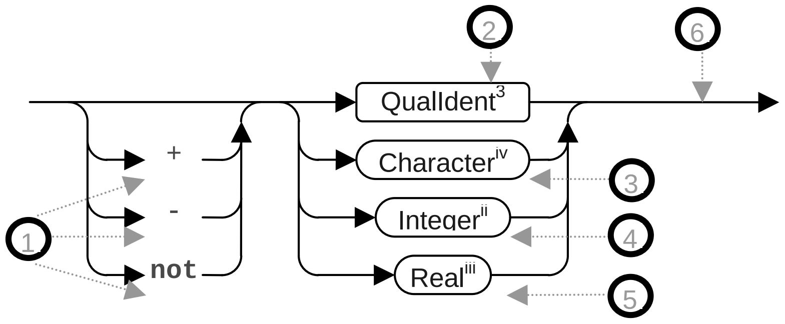
1) table.appendConstant(); //Add a new ConstantIdent to the table
2) table.isDuplicate(ident); // Verify that the name is unique in the current block
3) Set the ConstatIdent’s value to the value returned by RuleConstant.
   Set the export flag, etc…

*Figure 6.2.11 Annotated Grammar Rule Constant Declaration*

### 6.2.8.4 Constant[4]

ConstRec  Constant(…);

This function returns a ConstRec containing the value and type of the constant contained in the token stream. Set the type to notyp (or throw exception) for an invalid constant.



1) Record 'Sign’ information (needed to modify the constant when it is encountered)

    2) Lookup the constant (with QualIdent)
       Retrieve the value from identifier record
       Assign the value to the ConstRec;

    3) Retrieve value from Character rule (no base)
       Assign the value to the ConstRec (tf, ch, i, q, sp,…)

    4) Retrieve value from Integer rule (Don't forget the base)
       Assign the value to the ConstRec

    5) Retrieve the value from the Real rule (base 10)
       Assign the value to the ConstRec

**6)** Verify that the signs are done properly
Apply the sign to the value of the constant

*Figure 6.2.12 Annotated Grammar Rule Constant*

## 6.2.9 User Defined Simple Types

We discussed the TypeIdent record previously in conjunction with Modules, we visit them again to provide some concrete examples of other user defined types. As shown in figure 6.2.2 user defined types include modules, records, arrays, classes (covered in the next chapter). In figure 6.2.2, the NULL refers to the fact that intrinsic types may be user defined as well.

We might also mention that the compiler's symbol table comes with several "built-in" types. Looking at a .TAB file will show the built-in types, as well as the other intrinsic constants and functions.

### 6.2.9.1 TypeIdent

The TypeIdent class, whose data structure is shown in figure 6.2.13, is used for storing information about type declarations. In fact, the TypeIdent records requires no additional fields. Its primary duty is to associate a name (the first underlined field) with a type (the second underlined field. We create a second record to accent the difference between various kinds of identifiers (TypeIdent, VariableIdent, ConstantIdent and so on), and between the language level concept of type (represented by the TypeIdent class), and the compiler's internal understanding of type (represented by the Type class).

```
class Ident
{
   WORD        Mod;
   ObjectType  obj;
   bool        was_exported;
   bool        Export;
   Alfa        Name;
   Type        type;
   Block      *Parent;
};
```

```
class TypeIdent: public Ident
{
};
```

*Figure 6.2.13 Data structure of TypeIdent*

Listing 6.2.7 contains two examples of statements that would require the creation of TypeIdents.

```
export type my_int is int;    // #1
type revisited is ^ my_int;   // #2
```

*Listing 6.2.7 Sample SAL code to illustrate simple user-defined types*

Below, in table 6.2.3, are the resulting TypeIdent structures for these declarations.

| Field | #1 | #2 |
|---|---|---|
| name | my_int | revisited |
| mod | 1 | 1 |
| export | TRUE | FALSE |
| obj | typeobj | typeobj |
| type.datatype | inttyp | inttyp |
| type.subtype | _s32s | _s32s |
| type.extra | NULL | NULL |
| type.plev | 0 | 1 |
| parent | module 1 | module 1 |

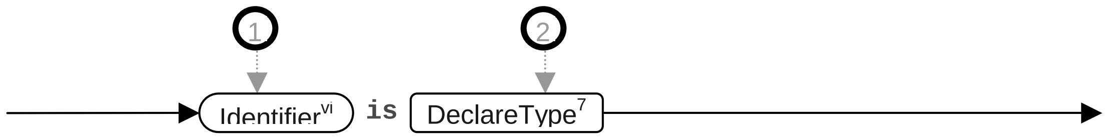
*Table 6.2.3 Sample values for TypeIdent records (simple types only)*

The following parser diagram, figure 6.2.14 provides an algorithm for updating the symbol table when user defined types are declared.

### 6.2.9.2 TypeDeclaration[8]

Void (Rule)TypeDeclaration(…,BOOLEAN Export);

Export indicates whether this Identifier should be exported



**1)** TypeIdent * temp = table.appendType(…); // Add a TypeIdent to the table

Fill in the typeIdent's name and Export fields

**2)** Set the pointer level

Pass the TypeIdent (*temp*) to DeclareType and let it initialize the type.

*Figure 6.2.14 Annotated Grammer rule TypeDeclaration*

### 6.2.10 Arrays: User Defined Types

SAL's array represent a homegenously typed collection of items indexed by an integer in a predefined range. Items could be intrinsic types, records or other arrays. Listing 6.2.8 shows a representative sampling of SAL array declarations. Figure 6.2.15 presents an outline of the corrseponding symbol table structure

```
type x is array [1 to 4] of int;
type y is array ['e' to 'q'] of array [10 to 15] of bool;
var z:array[-5 to 6] of SomeRec; [3]
```

*Listing 6.2.8 Sample SAL code to illustrate array types*

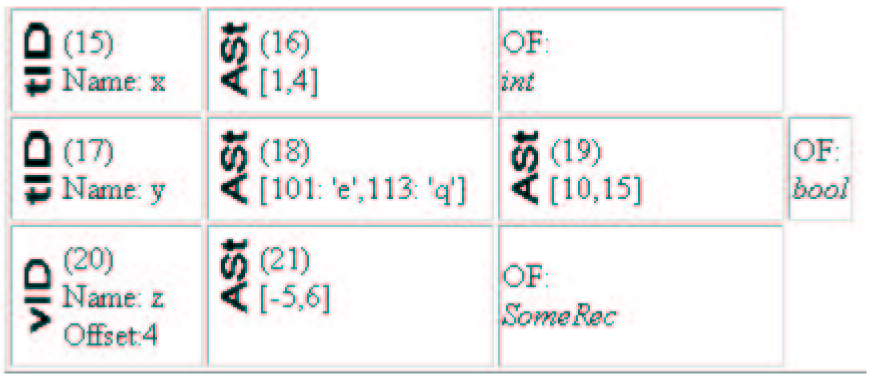


*Figure 6.2.15 Array Structure overview*

Whether an array is declared by a type declaration, or as part of a variable or array declaration, the information to be recorded about these statically sized arrays is the same.

#### 6.2.10.1 ArrayStruct

The syntax of the ArrayDeclaration gives a good indication of what information needs to be recorded. First, an array has a lower and an upper bound, with an associated type. Second, the compiler needs the type if the item being stored. Eventually, the compiler will also need information on the size of the entire array, and its individual members. The field for the size of the entire type is inherited from ArrayStruct's base class: Aggregate. *For convenience, a link to its parent has been borrowed from Aggregate*. The entire data structure ArrayStruct is shown in figure 6.2.16. We discuss the underlined fields.

[3] [0] The reader should notice that not all user defined types are named. in SAL, "anonymous" or unnamed types are still possible thanks to the seperation between "TypeIdent" and "Type". The nested array on the second line, ([10 to 15] of bool) and the array on the third line are examples of these anonymous types.

```
class Aggregate
{
     DWORD   Size;
     Ident * Parent;
     WORD    Block_mod_number;
     WORD    Ctor_proc_number;
     WORD    Dtor_proc_number;
     bool was_exported;
};
```

```
class ArrayStruct: public Aggregate
{
     ConstRec    Low;
     ConstRec    High;
     Type        Element;
     DWORD       Element_size;
};
```

*Figure 6.2.16 ArrayStruct data structure*

***ConstRec Low, High***

In order for the compiler to verify that array accesses remain within the array bounds (or to produce code to check at run time) the value of the bounds of the array must be stored. Because SAL arrays do not necessarily start with 0, we need to record the lower bound as well as the upper bound for valid array indexes.  A ConstRec struct (also used by ConstIdent) is used since the array index values must be stored, and since these values may be of many different types (bool, char, and int).

The **Low** field (accessed by *get/setLow()*) keeps the lower bound.

The **High** field (accessed by *get/setHigh()*) is the compliment of the low field. it stores the largest valid index for this array -- notice the array is from low to high *inclusive*!

Notice that the type stored by the array and the type used to index the array are not necessarily the same. Also, the type system must verify that the Low and High values are of the same type.

***Type Element***

As each array is a collection of items or elements, the type of this element must be recorded. *If only for nested arrays, but for also for other anonymous types, a Type rather than a Ident is used to store the element type.*

The Type class was explained in the identifier section.  As explained in the section on the type record, complex types (arrays and record) are reference bya pointer in ,the Extra field.. Instrinsic types are stored in the datatype and subtype fields of the class record, while the Extra field is left NULL. There are several accessed methods.

| | |
|---|---|
| *get/setElDatatype()* | // manipulate Element::datatype; |
| *get/setElSubtype()* | // manipulate Element::subtype; |
| *get/setElExtra()* | // manipulate Element::extra; |
| *get/setElType()* | // manipulate Element |

***DWORD Element_size***[A13]

Lastly, the size of a single array element is kept. Element_size is accessed by *get/setElSize()*.

A special note: ArrayStruct inherits the **Ident *parent** field from the Aggregate class. However, in the case of nested arrays (like Example #3 below) or arrays of anonymous records, a nested array has no Identifier declaring it directly. Since there is no declaring record for the parent to point to, the parent field should be set to NULL.

Now that the member fields have been present, a detailed example is in order. We examine the code in listing 6.2.9 (the same as listing 6.2.8). The resulting ArrayStruct records are shown in table 6.2.4. The elementType field (row 3) shows, from top to bottom, the values of the DataType, SubType and Extra fields.

```
type x is array [1 to 4] of int;    // #1
type y is array [‘e’ to ‘q’] of array [10 to 15] of bool;
//          ^--------#2--------^  ^--------#3--------^
var z:array[-5 to 6] of SomeRec;
//  ^--------#4--------^
```

*Listing 6.2.9 Sample SAL code to illustrate putting values in an ArrayStruct record*

| **Field** | Array #1 | Array #2 | Array #3 | Array #4 |
|---|---|---|---|---|
| **Size** | 4*4 = 16 | 13*6 = 78 | 6 * 1 = 6 | 12*\|SomeRec\| |
| **Parent** | x | y | NULL | z |
| *Type* **element** | int<br>_s32s,<br>NULL | arraytyp<br>nosub<br>#3 | bool<br>_u8s<br>NULL | rectyp<br>nosub<br>ptr to SomeRec |
| **element_size** | 4 | 6 | 1 | \|SomeRec\| |
| **Low** | {int, _s32s, 1} | {char,_u8s, ‘e’} | {int,_s32s,10} | int,s32,-5 |
| **High** | {int, _s32s, 4} | {char,_u8s,’q’} | {int,_s32s,15} | int,s32,6 |

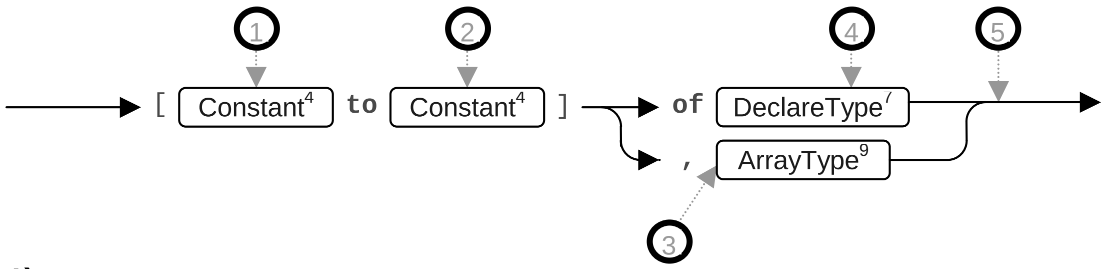
*Table 6.2.4 Sample values for ArrayAtruct records*

The next question is where does the compiler fill in these values. In answer, we turn to the syntax diagrams. Specifically, the ArrayType grammar rule has been annotated with appropriate pseudo code and is presented in figure 6.2.17.

#### 6.2.10.2ArrayType[9]

Void ArrayType(…, ArrayStruct *);

This function fills in all of the fields for the ArrayStruct record passed into it.



1) `ArrayStruct::setLow(…);  // Set the ArrayStruct’s low value with the value obtained from RuleConstant.`
2) `ArrayStruct::setHigh(…);  // Set the ArrayStruct’s high value with the value obtained from RuleConstant.`
3) `table.appendArray(); // Add a new array (ArrayStruct) record to table - (so we have an array of arrays)`
   `ArrayStruct ::setElType(…); … etc. // Pass the new record to ArrayType and set old array element’s type field to arraytyp, nosubtype, with the extra pointer to the new ArrayStruct record. Set the element’s size`
4) `Call DeclareType  (pass in a temporary Ident * to get the values)`
   `Set the Arraytype’s element type, subtype, size, and extra fields.`
5) `Determine and set the array size [(high-low+1) * element_size]`

*Figure 6.2.17 Annotated Grammar Rule ArrayType*

### 6.2.11 Variables

Variables show up everywhere! Looking at variables from the language level they could be described as containers for values which may have many different values in the course of execution. Looking from the compiler’s point of view, variables represent a run time memory location that can be read from or written to.

The scoping and structure of most programming languages allows a single variable to represent many different memory locations. Consider a function parameter. Each time the function is called, its parameters may occupy a different memory location – and when a function is called recursively, the same variable name must point to a different memory location to prevent the overwriting of data in previous calls. By treating a variable as a memory location *relative* to various other addresses, the compiler can support the rich behavior programmers expect from variables.

#### 6.2.11.1 VariableIdent

Variables are probably the most widely used of the identifiers.  Variables may appear as global variables, local variables or function parameters, as well as members of records and classes.  In order to deal with all of these different placements, a few more member fields are needed, not all of which are necessary in every situation. The complete data structure for the VariableIdent record is given in figure 6.2.18. The underlined fields are discussed below.

```
class Ident
{
   WORD        Mod;
   ObjectType  obj;
   bool        was_exported;
   bool        Export;
   Alfa        Name;
   Type        type;
   Block      *Parent;
};
```

```
class VariableIdent: public Ident
{
  DWORD    Offset;
  bool     pass_by_value;
  bool     Isa_Self_Param;
  bool     isConstructed;
};
```

*Figure 6.2.18 Data structure for VariableIdent*

`DWORD Offset`

The **offset** field can be accessed by *getOffset()* and *setOffset(DWORD)*. Offset represents the location in memory relative to some location in memory.

When a program is being compiled, it is impossible to say where anything will be placed in memory exactly. The linker and loader will be moving around various parts of a program in memory to suit the needs of the operating system.

The important question is how to compute these offset values. In every case, a variable is part of some aggregate structure (a record, a class, a function or a module). *A variable's offset is simply the sum of the sizes of all the variables preceding it - (for complex types only count the space for the pointer).* Depending on where the variable is declared, an additional (constant) offset will need to be added – as specified in table 6.2.5.

Local variables and parameters occupy the same offset space - in other words, the offset of the local variables is 20 + the size of all the parameters + the size of all preceding local variables. The offsets are caused by various VM architecture considerations described in section 6.2.11.4.

| Variable type | Offset relative to … | Additional Offset |
|---|---|---|
| global variable | global address space | GLOBALSTART = 4 |
| parameter | start of procedure on stack | LOCALSTART = 20 |
| local variable | start of procedure on stack | LOCALSTART = 20 |
| member of record | beginning of record | RECORDSTART = 0 |
| member of class | beginning of class | CLASSSTART = 0? |

*Table 6.2.5 Base Offset for variable records based on variable context*

`BOOLEAN pass_by_value`

The **pass_by_value** field (use *setByVal(BOOLEAN)* and *getByVal()* methods) tells whether a parameter is being passed by value or passed by reference. This field doesn't make much sense in other contexts - best to leave it in its default TRUE state.

The last two fields, IsaSelfParam and Constructed, are solely for use with classes, and we postpone their presentation until the presentation of classes.

Listing 6.2.10 has some examples of VariableIdents in action. Tables 6.2.6 and 6.2.7 show the corresponding VariableIdent records populated with their values.

```
export var a:char;                              // a: #1

var b:record                                    // b: #2
     c:int;                                     // c: #3
end record;

export proc foo(var d:int);                     // d: #4
  var e:real;                                   // e: #5
begin
end proc;
/*  Since foo is a FunctionIdent, not
    a VariableIdent, we won't cover it here */
var f:my_int;    // from TypeIdent example      // f: #6
```

***Listing 6.2.10 Sample SAL code to illustrate values for VariableIdent records***

| Field | #1 | #2 | #3 |
|---|---|---|---|
| Name | a | b | c |
| Mod | 1 | 1 | 1 |
| Export | TRUE | FALSE | FALSE |
| Obj | varobj | varobj | varobj |
| Type.datatype | chartyp | recordtyp | inttyp |
| Type.subtype | _u8s | nosubtyp | _s32s |
| Type.extra | NULL | (valid) | NULL |
| Parent | module 1 | module 1 | module 1 |
| Offset | 4 | 8 | 0 |
| Pass_by_val | TRUE | TRUE | TRUE |

***Table 6.2.6 Sample values for VariableIdent records***

| Field | #4 | #5 | #6 |
|---|---|---|---|
| Name | d | e | f |
| Mod | 1 | 1 | 1 |
| Export | TRUE | FALSE | FALSE |
| Obj | varobj | varobj | varobj |
| Type.datatype | inttyp | realtyp | realtyp |
| Type.subtype | _s32s | _s32s | inttyp |
| Type.extra | NULL | NULL | NULL |
| Parent | foo | foo | module 1 |
| Offset | 20 | 24 | 12 |
| Pass_by_val | FALSE | TRUE | TRUE |

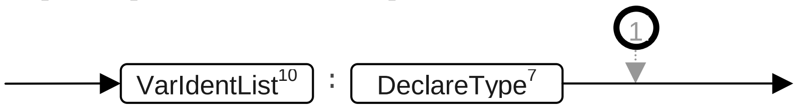
***Table 6.2.7 Sample values for VariableIdent records continued***

Correctly setting the offsets for variables, and correctly updating the sizes of the blocks which contain variables is one of the most complicated aspects relation to symbol table construction. We present a pseudo-code algorithm for this process in the context of the VariableDeclaration syntax diagram, figure 6.2.19. Some small assistance is allso provided by the VarIdentList rule, shown in figure 6.2.20.

### 6.2.11.2 VariableDeclaration[15]

Void VariableDeclaration(…,BOOLEAN Export);

Export is passed in from CompilationUnit and is used to set the export field.



**1)** Determine the type size of each variable from the type; // See size/offset Table

Determine the offset size for the variable from its type and scope;  //See size/offset table

Determine the number of variables from the count passed to VarIdentList

Use the size & number of variables to get address/offset offset info using the following algorithm

```
  totalTypesize:= typeSize*=count;
  totalOffsetSize:= offsetSize * count;
  currentOffsetInScope := currentOffsetInScope + totalOffsetSize;
  currentBlockSize := currentBlockSize + totalTypeSize;

  variableOffset := currentOffsetInScope;
  currentIdentifier := table.getLast()  //   Get the last identifier
in the current scope
  while  count > 0
  count := count-1;
  variableOffset:= variableOffset - variableOffsetSize;
  … //Set the variable type fields and the export and offset fields;
       //Handle anonymous type parents here…………….
  loop;
```

*Figure 6.2.19 Annotated VariableDeclaration Grammar rule*

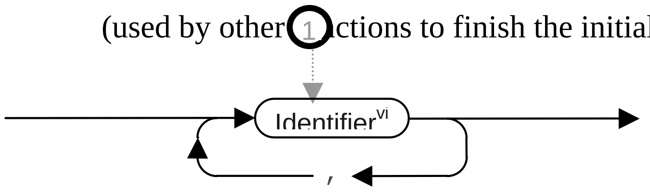
### 6.2.11.3 VarIdentList[10]

Void VarIdentList(…, int & count);

This function adds a variable (just the name) to the table for each variable in a list.

Count (passed in or returned) tells how many variables were added to the table.

(used by other functions to finish the initialization.)



**1)** table.appendVariable();        // Add a VariableIdent record to table
Fill in the VariableIdent’s name field
Increment count;

*Figure 6.2.20 Annotated VarIdentList Grammar Rule*

#### 6.2.11.4Variable Size & Offset Table

What effect a variable has on the offsets of subsequent variables, and its contribution to run time space allocation in the SAL virtual machine depends on where that variable is declared and whether it is a simple (intrinsic) or complex (user defined) type. Table 6.2.8 describes how variables contribute to the size cumulative offset of the current block based on their location in the code.

| | | Global Variables | Parameters & Local Variables | | Record Fields |
|---|---|---|---|---|---|
| | | | By Value | By Reference | |
| Simple | Size | sizeof(type) | sizeof(type) | **4** | sizeof(type) |
| | Offset | sizeof(type) | sizeof(type) | **4** | sizeof(type) |
| Complex | Size | **4** + sizeof(type) | **4** | **4** | sizeof(type) |
| | Offset | **4** | **4** | **4** | sizeof(type) |
| Starting Offset: | | 4 | 20 | | 0 |

*Table 6.2.8 Offset and Size-contribution table*

The column specifies where the variable occurs, the major rows specify what the variables type is (simple or complex) and the sub-rows show whether size or offset it being discussed. So, for example, if you create a global variable x of type int -- it contributes "sizeof(int)" or 4 bytes to the amount of global space that must be allocated, and the next variable will start at an offset "sizeof(int)" or 4 greater than x's offset.

Another example, a user-define record passed into a function by reference would add 4 to the procedure's size (4 being the size of a pointer) and the next parameter (or local variable) would have an offset 4 greater.

The last row indicates the offset of the first variable in that class.  Global variables begin at offset 4 (the first 4 bytes are used by the virtual machine to make sure each module is only loaded once). Parameters and/or local variables start at offset 20 (the previous 20 bytes store the block-marker which regulates function return behavior, etc). Records start at offset 0 (because no regulatory information is stored with records).

These values are specific to the SAL virtual machine. Different architectures will place different (though likely similar) requirements for initial offsets, and perhaps even for size and offset contributions.

The particular values for sizes and offsets for complex types comes from our chosen implementation. We assume that all records and arrays are referenced by a pointer. This makes passing complex types as function arguments very simple. In every case, the pointer is pushed on the expression evaluation stack (EES). If the complex type is passed by reference, the pointer is then copied from the EES to the local variable area. If the complex type is passed by copy, a VM instruction will copy the complex type's contents to the procedure stack, update the procedure stack size and the pointer and to the copied data. The new pointer is then copied to the local variable area. The only complex ramification is that your compiler is responsible for allocating both the pointer and the body of complex types in the global memory area -- hence the 4+sizeof(type) entry.

Complex types could also be implemented "inline" -- e.g. without the pointer direction. This will effect what values are assigned in the size and offset fields of the symbol table, as well as how code is generated for manipulating variables of complex-types (especially in parameter passing).

### 6.2.12 User Defined Record types

Records represent a heterogeneous collection of elements. Each element is accessed by name. The elements (or members) each have a different type, including: intrinsics, arrays and records. The following code segment, Listing 6.2.11 shows two examples of records in SAL.

```
type rec is record
   a:int;
   b:real;
end record;
var anon_rec: record
  x:array[1 to 5] of int;
  y:rec;
end record;
```

*Listing 6.2.11 Sample SAL code to illustrate user-defined records*

#### 6.2.12.1Block or Record

One can easily infer from the syntax that the symbol table needs to store the members of a record. It is also reasonable to expect the size of the record to be stored, as well as a pointer to the Identifier that declared it (if any). However, there is some less obvious information that proves usefull as well. We present the all the information stored in records below. The complete symbol table data structure for the SAL Record type is given in figure 6.2.21.

```
class Aggregate
{
 DWORD   Size;
 Ident * Parent;
 WORD    Block_mod_number;
 WORD    Ctor_proc_number;
 WORD    Dtor_proc_number;
 bool was_exported;
};
```

```
class Block: public Aggregate
{
 List < Ident >  Elements;
 Block *Pred;
 WORD Function_Level;
 WORD Scope_Level;
};
```

```
class Record: public Block {};
```

*Figure 6.2.21 Record/Block data structure*

***DWORD Size***
***Ident *Parent***

These two fields are inherited from Aggregate. The size of a record is the sum of the sizes of all the member fields. Parent still points to the declaring (Type or Variable) identifier.

Blocks also define a new layer of scope. The following fields are used to keep track of the scoping situation.

***List < Ident >  Elements***

As shown in figure 6.2.2, earlier in the chapter, records contain only variables. Each record member is stored in the Elements field. We keep a list of Identifiers (rather than VariableIdents) so that other classes, which inherit fields from record, can use the same list to store constants, types, and other Identifers..

Access to the Identifiers in **Elements** is facilitated by four functions in the Table class: *goToFront()*, *goToEnd()*, *goToNext()* and *goToPrev()*. The list class maintains and internal pointer to the current Ident. These for functions manipulate that pointer. For more information see the helper function in class Table below.

***Block *Pred***

Pred points to the block which contains this block (there is usally an Ident record in between). It is used to facilitate searches within scope – specifically the Table::find method calls “findInScope” and starting in Table::CurrentBlock, and then recursively for each Pred pointer. This field is set by the table class and should not be manipulated by the student.

***WORD Function_Level***[DS14]

This field tells how deeply a function is nested. It is a count of the number of functions between this block's Identifiers and the module. It may be accessed by *getFuncLev()* and *setFuncLev()*.

```
WORD Scope_Level
```

Scope_Level counts the number of functions *and classes* between this block's Identifiers and the module it is declared in. It is accessed via *getScopeLev()* and *setScopeLev()*.

Scope_level and Function level are very similar. In fact, without classes, they are the same. When functions are nested outside of classes, scope_level and function level will be the same, when functions are nested inside a class, they will differ by the number of classes involved. We will visit these fields again when we talk about classes.

Now we look at the syntax diagram RecordType, figure 6.2.22, to show where these symbol table values are assigned.

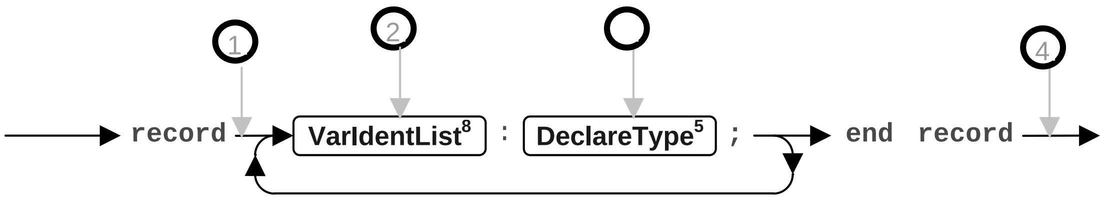
### 6.2.12.2RecordType[11]

Void RecordType(…,Ident *);

The Ident parameter is the identifier that should be the parent of this record.

This function adds lists of variables to the table and fills in all their fields.

Append_Block automatically increases the level, so be sure to call LeaveBlock() at the end.



**1)** `table.appendBlock()->toBlock();  // Add a Block record to the table`

**2)** `Call VarIdentList passing an initial number of variables to zero;`

**3)**
```
Call DeclareType to determine the size of the type
Use size of the type from declare type and the count from VarIdent
list to  determine total space used.
Increment the total block size;
for each identifier in the current scope
{
  calculate offset (sum of preceding variable sizes in record)
  set identifier fields including  offset
}
```

**4)**
```
Set the total block size
Set Ident's extra to point to the block;
Exit the block (LeaveBlock())
```

*Figure 6.2.22 Annotated RecordType Grammar Rule*

### 6.2.12.3Some Examples

The code in Listing 6.2.12 would result in three records of type Record, with data values as shown in table 6.2.9.

```
type X is record        // #1
   a:int;
end record;

export var Y:record     // #2
   Z:record             // #3
      c,d,e:real;
            f:char;
   end record;
   g:char;
end record;
```

*Listing 6.2.12 Sample SAL code to illustrate symbol table values for Block (or Record) records*

| Field | #1 (X) | #2 (Y) | #3 (Z) |
|---|---|---|---|
| Size | 4 | 14 | 13 |
| Parent | Ident X | Ident Y | Ident Z |
| Elements | a | Z,g | c,d,e,f |
| Pred | Module 1 | Module 1 | Block #2 (Y) |
| Function_level | 0 | 0 | 0 |

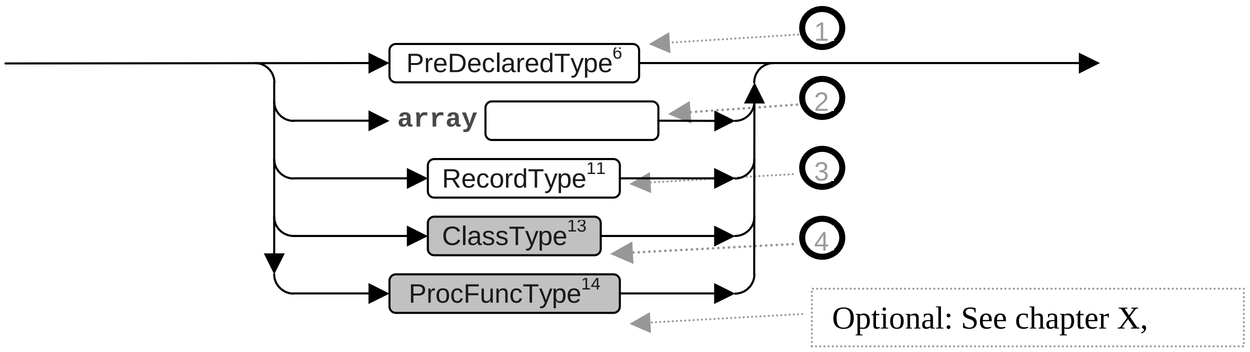
*Table 6.2.9  Sample values for Block (or Record) records*

Confusion may occassionally occur when constructing complex conglomerates of records an arrays. The following grammar rule, figure 6.2.23, is in charge of coordinating thw construction of very complex types.

### 6.2.12.4DeclareType[7]

Void  DeclareType(…., Ident*)

This function "routes" complex type information between several functions. The added Ident* parameter allows this function to pass back the type information to the calling function.



**1)** `Let PreDeclaredType initialize the ident’s type.`

**2)**
```
Set the ident’s type to arraytyp
table.appendArray(…);  // Add an ArrayStruct record to table
Let ArrayType initialize the ArrayStruct
```

```
3) Set the ident's type to recordtyp
   Let RecordType do  the rest of the work
4) Set the ident's type to classtyp
   Let ClassType do the rest of the work
```

*Figure 6.2.23 Annotated RecordType Grammar Rule*

### 6.2.13 Sub-Programs

Subprograms represent blocks of executable code. They have received many names including procedures, functions, methods, and sub-routines. SAL makes a distinction between procedures and functions. Specifically, procedures have no return values, while functions do. We use the term *method* to refer to procedures or functions that are members of a class, and the term sub-program to refer to any block of executable code. Listing 6.2.13 illustrates the syntax of procedures and functions in SAL.

```
proc outer (b:real)
  var x:char;
  func inner():int;
    var y:char;
  begin
    // code goes here
  end func;

begin
  // code goes here
end proc;
```

*Listing 6.2.13 Sample SAL code for Subprograms (precedure and function)*

With the possible exception of classes, sub-programs are the most complicated programming construct that the SAL symbol table handles. Not only do we worry about a function's name, but also about its parameters, its local variables, how it modifies the scoping rules, what value(s) it returns, where it will be located at runtime and so on.  This complexity caused us to use two symbol table records to store a sub-program instead of one. These are FunctionIdent and Function Block.

Many of the issues relating to sub-programs are complicated by the inclusion of classes. We will postpone the discussion of these complications until the next chapter. We present the data structures for FunctionIdent and their use in the following 3 subsection, then turn our attention to the FunctionBlock half in the remaining subsections.

#### 6.2.13.1FunctionIdent

Function Ident inherits from Ident several key fields, including: ***`Name`***, ***`Type`*** and ***`Parent`***. The complete data struct for FunctionIdent records appears in Figure 6.2.24

***`Name`*** contains the identifier for this function or procedure. ***`Type`*** does NOT contain the return type of a function! ***`Type`*** records that fact that this is a procedure or function. The ***`Parent`*** field points to the block in which this sub-program was declared. Looking at figure 6.2.2 we see that this containing block may be a module, a class or another sub-program.

```
class Ident
{
   WORD        Mod;
   ObjectType  obj;
   bool        was_exported;
   bool        Export;
   Alfa        Name;
   Type        type;
   Block      *Parent;
};
```

```
class FunctionIdent: public Ident
{
   WORD proc_number;
   LONG Virtual_Function_Number;
   bool Is_Constructor;
   bool Is_Destructor;
};
```

*Figure 6.2.24 FunctionIdent data structure*

***WORD proc_number***

**proc_number** is accessible by *getProcNum()* and *setProcNum()*. In many compilers functions are referenced by function pointers which must be stored in terms of some relative address. Fortunately, the virutal machine has simplified this task, by removing the difficulty of managing these function pointers. Instead, the virtual machine constructs and maintains a table of function pointers. All function calls are made via an index into the virtual machine's function table. **proc_number** is this index number.

Index numbers should be generated sequentially from 1 to N where N is the number of sub-programs in the module (not including the main body of the module). Index number 0 is reserved for the main body of the module (i.e. the section between ***begin*** and ***end library.*** or ***end program.***) Index numbers are generated in the order that the sub-program headers are encountered.

### 6.2.13.2Some Examples

The SAL code in listing 6.2.14 would result in FunctionIdent records with values as given in table 6.2.10.

```
export proc A();
begin
end proc    // A

func B():int;
      proc C();
   func D():char;
   begin
   end func; // D
  begin
  end func; // C
begin
end func; // B
```

*Listing 6.2.14 Sample SAL code for nested functions*

| Field | #1 | #2 | #3 | #4 |
|---|---|---|---|---|
| Name | A | B | C | D |
| Export | TRUE | FALSE | FALSE | FALSE |
| Obj | Funcobj | funcobj | funcobj | funcobj |
| type.datatype | Proctyp | functyp | proctyp | functyp |
| type.subtype | Nosubtyp | nosubtyp | nosubtyp | nosubtyp |
| type.extra | (valid) | (valid) | (valid) | (valid) |
| Parent | module 1 | module 1 | B | C |
| proc_num | 1 | 2 | 3 | 4 |

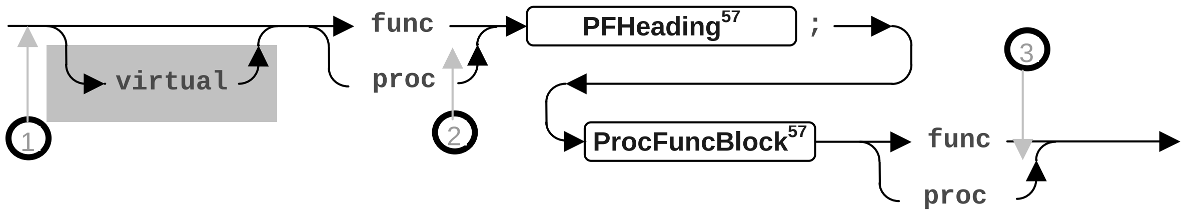
*Table 6.2.10 Sample values for FunctionIdent records*

In the now familiar pattern, we present the annotated syntax diagrams that create and update the values of the FunctionIdent records. These are PFDeclaration (figure 6.2.25) and PFHeading, (figure 6.2.26). The PF stands for Proc/Func since both procedures and functions may be declared in these rules.[DS15]

### 6.2.13.3PFDeclaration[56]

Void PFDeclaration(… BOOLEAN Export); // For declaring procedures or functions

Export is used to set the export field. Came from Compilation Unit



```
1) table.appendProcFunc(); // Add a FunctionIdent record to the table
   table.appendFunction(); // Add a FunctionBlock record to the table
   saveCurrentOffsetInScope := CurrentOffsetInScope;  // Save the
   settings for the outer scope
   saveCurrentBlockSize:= CurrentBlockSize;
2) Set the FunctionIdent’s export field
   Set the FunctionIdent’s type to functyp (proctyp)
3) Verify that proc/func matches opening proc/func
   Exit the current scope (table.LeaveBlock())
   CurrentOffsetInScope := saveCurrentOffsetInScope; // Restore the
   settings for the outer scope
   CurrentBlockSize:= saveCurrentBlockSize;
```

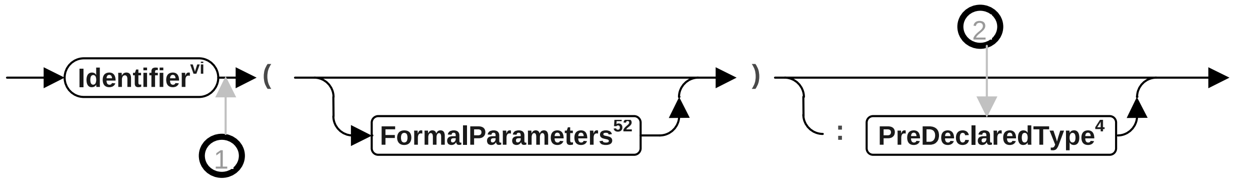
*Figure 6.2.25 Annotated PFDeclaration Grammar Rule*

### 6.2.13.4PFHeading[57]

Void FuncHeading(…,FunctionIdent *, FunctionBlock *)

This function needs both the function identifier (to set the name, etc)

And the FunctionBlock (for size, parameters, return type, etc.)



**1)** Assign the FuncIdent's name
table.isDuplicate(ident) // Verify that the name is unique in the current block
CurrentOffsetInScope:= LOCALSTART;  // Sets current offset after function header;

Increment the current procNum global variable and assign it as the FunctionIdent's procedure number.

**2)** Pass a dummy identifier to DeclaredType and use its type to set the function's return type

*Figure 6.2.26 Annotated PFHeading Grammar Rule*

### 6.2.13.5FunctionBlock

The last important type of record for the Basic Compiler class is the FunctionBlock. This class stores all the extra information required on functions. A FunctionIdent and FunctionBlock will always be paired together, each maintain separate, but necessary, information about the function.

Functions may themselves contain many identifiers (the most obvious examples being parameters and local variables). These are stored in the ***`Elements`*** field inherited from class Block.

The FunctionBlock data structure appears in figure 6.2.27.

```
class Aggregate
{
     DWORD   Size;
     Ident * Parent;
     WORD    Block_mod_number;
     WORD    Ctor_proc_number;
     WORD    Dtor_proc_number;
     bool was_exported;
};
```

```
class Block: public Aggregate
{
     Block *Pred;
     WORD Function_Level;
     WORD Scope_Level;
     List < Ident >  Elements;
};
```

```
class FunctionBlock: public Block
{
     BYTE   Number_of_Parameters;
     DWORD  Size_of_Parameters;
     bool   Return_by_Value;
     Type   *ReturnType;
     bool   Throws_Exception;
};
```

*Figure 6.2.27 FunctionBlock data structure*

`BYTE Number_of_Parameters`

Since parameters and local variables are stored in the same list, some method is needed to distinguish the two.  This is the role of the **Number_of_Parameters** field.

`DWORD Size_of_Parameters`

How much space the parameters take up on the stack. IMPORTANT NOTE:  if a variable is passed by reference, it only takes up space for a pointer -- 4 bytes.

`Type *ReturnType`

return_type contains the return_type of the function.  For procedures it should either be NULL or be filled as follows {notyp, nosubtyp, NULL, 0} indicating it cannot return a value.

## 6.2.13.6Some Examples

To illustrate how the various fields are used, we present the code of listing 6.2.15.  Please note the presence of a nested function, and the effect of passing a character (normally 1 byte) by reference. Table 6.2.11 provides the corresponding FunctionBlock record values.

```
proc a(x:int; y:char);  // #1
begin
end proc;
```

```
func b():int;           // #2
begin
   return 5;
end proc;

func c():char;          // #3
   var x:int;
   proc d(var ch:char); // #4
      var y:real;
   begin
   end proc;
begin
   return 'a';
end func;
```

*Listing 6.2.15 Sample SAL code to illustate FunctionBlock values*

| Field | #1 | #2 | #3 | #4 |
|---|---|---|---|---|
| Size | 8 | 0 | 4 | 8 |
| Parent | Ident a | Ident b | Ident c | Ident d |
| Elements | x, y | - | x,d | ch, y |
| Pred | Module 1 | Module 1 | Module 1 | Block #3 (c) |
| #_of_parameters | 2 | 0 | 0 | 1 |
| Size_of_parameters | 5 | 0 | 0 | 4 (by ref!) |
| return_type | {no,no,-} | {int,s32,-} | {char,u8,-} | {no,no,-} |
| function_level | 1 | 1 | 1 | 2 |

*Table 6.2.11 Sample values for FunctionBlock records*

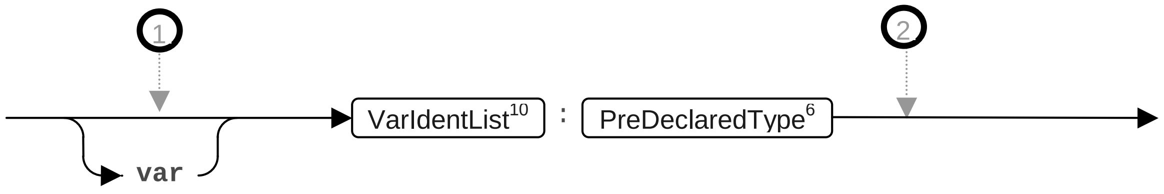
After this simple example, we turn to the syntax diagrams to provide general algorithms for updating the symbol table in the syntax directed compiler. The following two grammar rules (figures 6.2.28 and 6.2.29) parse most of the members contained in the procedure block. However, the handling of parameters and the return type was presented in the previous section 1.1.4.

#### 6.2.13.7 FPSection[52]

Void FPSection(…,FunctionBlock*);

This function creates the table information for parameters.

The new parameter is the Record for the function. It contains a parameter count and other



information.

**1)** Set a pass by reference flag

**2)** Use the type returned by PreDeclared type and the parameter passing type to determine the size of the parameter type.

Use the count passed into VarIdent list to determine the number of parameters of that type

Assign offsets to all the variables using the following algorithm

```
  totalTypesize:= typeSize*=count;
  totalOffsetSize:= offsetSize * count;
  currentOffsetInScope := currentOffsetInScope + totalOffsetSize;
  currentBlockSize := currentBlockSize + totalTypeSize;

  parameterOffset := currentOffsetInScope;
  currentIdentifier := table.getLast() // Get the last identifier
                                       // in the current scope
  while  count > 0
    count := count-1;
    parameterOffset:= parameterOffset - variableOffsetSize;
    … // Set the variable type fields
      // and the byValue and offset fields;
      // Handle anonymous type parents here…………….
  loop;
Update the FunctionBlock's parameter size and count field
```

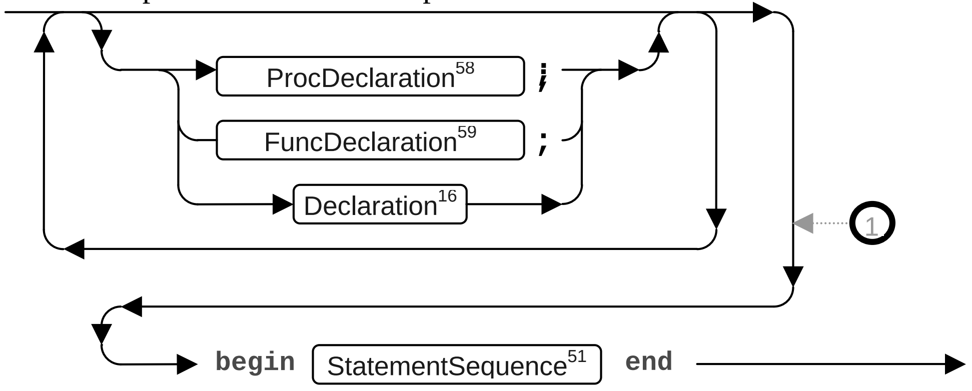
*Figure 6.2.28 Annotated FPSection Grammar Rule*

### 6.2.13.8 ProcFuncBlock[60]

Void ProcFuncBlock(…, FunctionBlock *block*);

This function is used to add local variables, types and functions to the symbol table.

The block parameter is used to update the size field as needed



**1)** block->setSize(currentBlockSize); // Update the size of the block

Observation: half of this grammar rule is concerned with declarations, half with code. Since the declaration is already processed and in the symbol table before the code is processed, a subprogram in SAL can be recursive.

*Figure 6.2.29 Annotated ProcFuncBlock Grammar Rule*

### 6.2.14 Conclusion

As indicated in the theory section of this chapter, the symbol table represents an increase in complexity over the scanner and parser sections. This increase in complexity is manifest in the number of data structures required to handle the increase in information. A further complication is that the organization of the symbol table, the type of data to be stored and many of the values stored in the data structres will be highly dependent, not just on the languagebeing compiled, but on the architecture that is the target of compilation.

Further, we point out that this chapter has only presented the symbol table data structres and their initialization. Chapters 6, 7 and 8 will present the details of code generation for the SAL language. These chapters will rely heavily on the ground work laid in this chapter. In fact, almost every generated instruction will require some information from the symbol table, ranging from the offsets of variables, to the size of complex types, and from the types of constants to the number and order of parameters in a function.

Page: 3
[s1]We’ll probably want to move this somewhere else, it doesn’t really fit with identifiers anymore.

Page: 14
[DS2]Is this true?

Page: 19
[s3] Where do we mention intrinsic functions?

Page: 20
[s4]Not to self: change implementaiton of appendModule to return a pointer

Page: 20
[s5]Note to self: remove “AppendBlock()” from library file

Page: 20
[s6]Note to self: change LeaveBlock to LeaveScope() or ExitScope() in Library file.

Page: 24
[DS7] Remove vis from the implementation

Page: 25
[DS8]Consider changing this to _f64s

Page: 30
[DS9]How does the linker do this?  I assume some symbol info must still go into the object file…

Page: 30
[DS10]I assume the only solution is an incremental compilation where a file is compiled up until the point where it requires data from another file -- the 2nd file is compiled, until it needs info from the first, and back and forth until both are compiled. Or at least until the 1st is completely compiled. Another Solution is to Compile the second file just for the symbol information, and leave dangling references wherever names cannot be found w/out recursion.The dangling pointers are filled in as the first file is compiled. (unresolved names result in errors)

Page: 30
[s11]A usefull assignment would be to replace the library export/import functions with a student implememtation.

Page: 32
[s12]Somewhere we need to talk about module 0 and the types that are defined there, as well as the values and functions.

Page: 37
[A13] I'm not sure, but this could (and should) probably be removed, along with the setElSize() method. getElSize() would just call the size() method of its element -- but it may not have one since it's a Type record rather than an Ident descendent.

Page: 45
[DS14] This should probably be placed in FunctionBlock.

Page: 50
[DS15]We can now enforce the semantic requirement that procedures have no return type, and that a declaration that begins with a the keywork proc must end with the keyword proc. While this could have been accomplished by stratifying the grammar, the semantic similarities between the rules make it easier to combine them together. (This way, the programmer won’t have to key in the code generation algorithms twice).