Unit 10: The Symbol Table

SCC312 Compilation

Aims

- To describe the function and possible structure(s) of the single most important data structure of a compiler …
- To describe how the Symbol Table copes with scoping, blocks, and scoping rules.

General concerns of the symbol table

The Symbol Table

- An important feature of a high-level language is the use of *symbols*. (or more simply, names)

- We can assign names to instances of various “things” (pieces of code, data objects, and even to classes of object).

- And have the compiler check that only operations of the appropriate type are performed on an object.

The Symbol Table

- Thus the compiler needs a data structure of some sort to hold all the names introduced in a program, together with information about the object(s) associated with it.
- This is the *symbol table*.
- It stores information about anything the programmer *declares* and checks on how the objects declared are used.

Declarations…

What are the requirements?

- Every time the compiler encounters an identifier in the program text it needs to know if it is being used correctly in the context, so it is going to need to look it up in the symbol table.
- Hence searching the symbol table for a particular identifier must be very efficient.

Looking up symbols …

What are the requirements?

- Every time the compiler finds an identifier it has not encountered before it needs to insert the identifier in the symbol table, together with some information about the context in which the identifier was encountered
  - Even if only to say that the identifier has not yet been declared.

- So it must be possible to add new entries into the symbol table efficiently.

Adding new symbols…

Using the Symbol Table

- Every time the lexical analyser recognises that the next token is an identifier, it could pass this fact together with the character string denoting the identifier back to the syntax analyser.

- Then the syntax analyser would
    - Search the symbol table for this identifier, to check the context in which it appears this time,
    - And if necessary to insert the identifier and associated information in the symbol table.

Using the Symbol Table

- Alternatively the lexical analyser could search the symbol table for each *identifier* recognised in the input stream, and return a reference to the appropriate entry

- The syntax analyser could then check the associated fields against the context in which the *identifier* was found.

Using the Symbol Table

- If the *identifier* was one not encountered before, the lexical analyser would insert the new *identifier* in the symbol table, presumably marked as "undefined" or as a new entry, and pass back the reference to the syntax analyser.

Pre-Loading the Symbol Table

- Often extra entries are placed in the symbol table before the compiler starts to scan the source text of the program
    - For example, in Pascal the standard i/o function calls like *readln* and *write* could be pre-inserted into the symbol table
    - Similarly in Java, the compiler could process the *import* statements at the beginning of a file by accessing the named package specifications and copying the symbol information into the symbol table before processing the program text.

What’s in the Symbol Table : Identifier

- The *identifier* itself
- There are two ways we could hold this
  - 1. We could hold it in a field in the symbol table (in which case we may have to decide a maximum length for an identifier, and allocate this amount of space in each entry).

What’s in the Symbol Table : Identifier

- The *identifier* itself
- There are two ways we could hold this
  - 2. Alternatively we could have an area to hold all the character strings, one after the other, and the field in the symbol table entry has a pointer to the beginning of the string in the string area (and maybe its length, unless the end of the string is marked by a special character).

What's in the Symbol Table : Types

- There will be *type* information
- In the simple cases this would be just an indication that the variable is a boolean, an integer, etc.
- We will discuss the use of type information in semantic checking later.

What's in the Symbol Table : Types

- But there are more complicated cases
  - For an array we would need to hold information about the number of subscripts and their types and ranges
  - For a procedure/method we would need to hold the number and types of parameters (and the type of the return value for a function)
  - For a record we would need to hold the list of fields and their types; etc.

What's in the Symbol Table?

- There will need to be information about the *scope* and *lifetime* of particular variables.
  - We're going to take a closer look at this shortly.

- The *store address* occupied by this object, to be used when we are generating the machine code

What’s in the Symbol Table?

- There may be other pieces of information about the identifier or as part of the element representing the identifier
  - Pointers to allow lists of the identifiers to be produced in alphabetical order, if necessary
  - Lists of references to places in the source text where the identifier is declared and used, if we wish to be able to generate this type of information
  - etc.

Symbol Table : data structure

- The Symbol Table is a very important part of the compilation process, but it is “just” a data structure.

- You have met a number of data structures already in your studies and these are candidates for use as a Symbol Table.

- For example, you might use binary trees or hash tables to implement the Symbol Table.

Scoping, Blocks and Scoping Rules

Scoping, Blocks and Scoping Rules

- Every declaration in a program has a ***scope*** — where in the program the declaration is active.

- A ***block*** is a program phrase that delimits the scope of declarations that it encapsulates, e.g.
  - **`procedure three() var a: real; begin ... end`**

- ***Scoping rules*** help us check that an identifier has been correctly declared.

Different schemes for blocks

- In general, there are three schemes for handling blocks:
  - Monolithic
  - Flat
  - Nested

Different blocks: MONOLITHIC blocks

- There’s just one block for the whole program.

- ***Basic*** and ***Cobol*** do that.

- The scoping rules for a monolithic block structured language go something like this:
  - 1. An identifier can only be declared once.
  - 2. There must be a declaration for every identifier.

- Problem is, there’s no locality. Anybody can access any identifier. Not portable. Not modular. Not scalable.

Different blocks: FLAT blocks

- Declarations are local to a block, or global to the whole program.

- ***Fortran*** does this.

- The scoping rules for a flat block structured language go something like this:
  - 1. No globally declared identifier can be re-declared.
  - 2. No locally declared identifier can be re-declared (within its block).
  - 3. There must be a local or global declaration for either identifier.

Different blocks: NESTED blocks

- ***Blocks*** can be ***nested***.
- ***Pascal, Ada, C, Java***, all do this.
- The scoping rules usually go like this:
  - 1. An identifier can only be declared once within any given block.
  - 2. For every occurrence of an identifier there must exist a corresponding declaration, either within the immediate block or in an enclosing block.

What do we require of a symbol table?

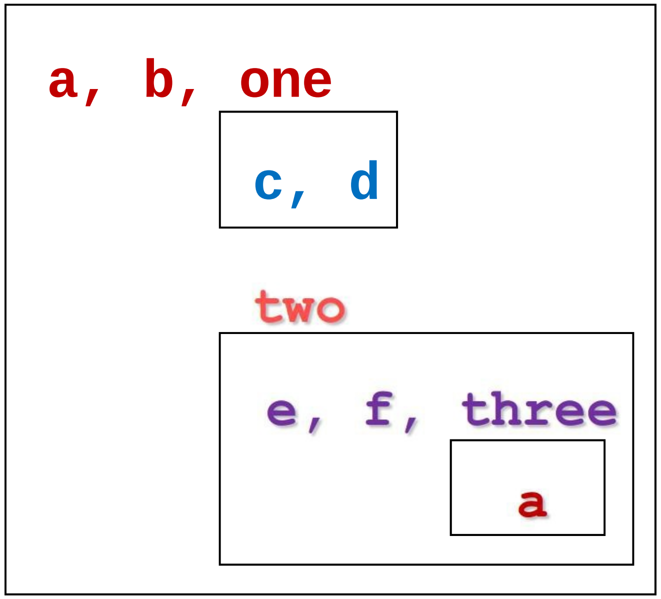
- Given an occurrence of an identifier, it must let us find the corresponding declaration.

- The symbol table must prevent duplicate declarations at the same lexical depth.

- The symbol table must resolve duplicate declarations at different lexical depths according to the scoping rules of the language.

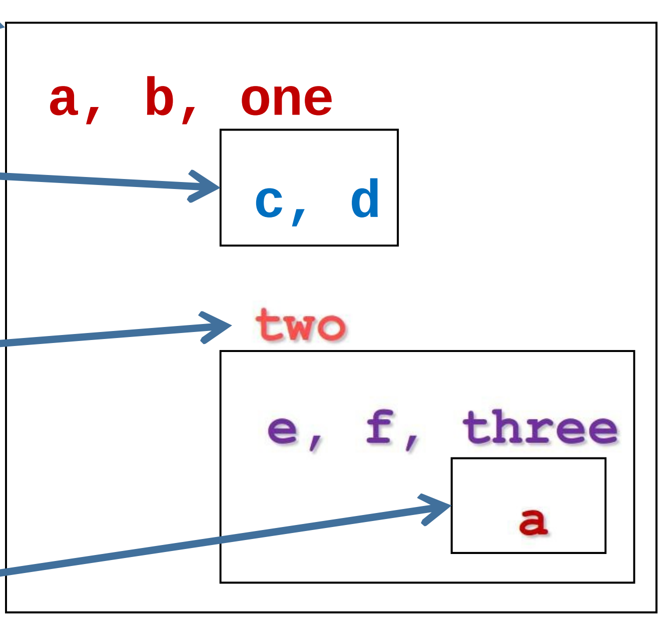
An example of NESTED block structure

```
function main() {
     let a = 10;
     let b = 20;
     ...
     function
     one() {
          let c = 30;
          let d = 40;
          ...
     }
     function two() {
          let e = 50;
          let f = 60;
          function three() {
               let a = 70;
               ...
          }
     }
}
```



An example of NESTED block structure

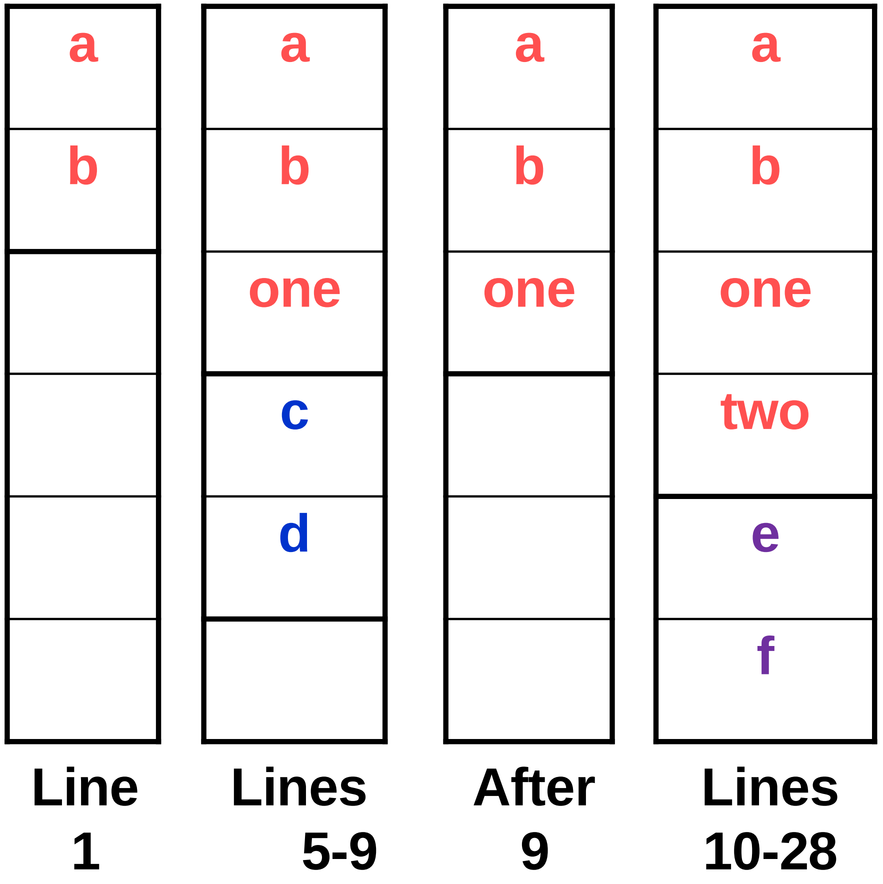
```
function main() {
    let a = 10;
    let b = 20;
    ...
    function
    one() {
           let c = 30;
           let d = 40;
           ...
    }
    function two() {
           let e = 50;
           let f = 60;
           function three() {
                  let a = 70;
                  ...
           }
    }
}
```



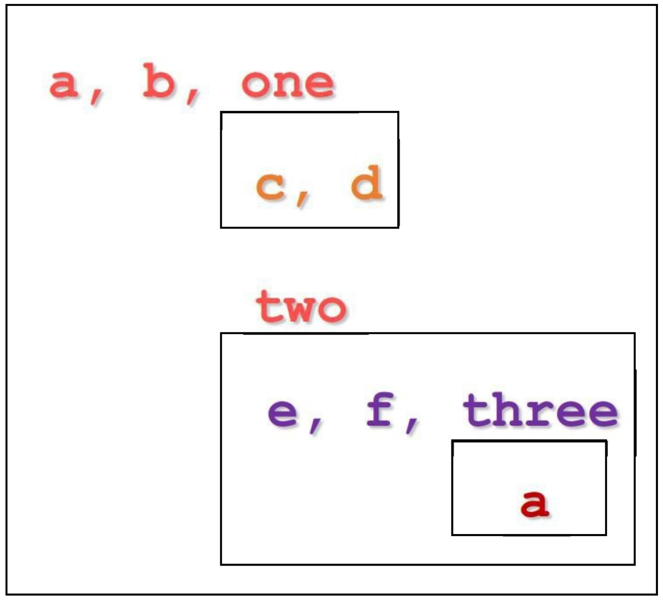
Simple stack organisation

- Each time a procedure is recognized, create space on a stack for local variables.

- When you finish analysing it, deallocate its local variables. They cannot be accessed during the analysis of the rest of the syntax.



What variables can be accessed within a particular range of lines in the source?



Limitations of stack-based organisation

- For an N-pass (N > 1) compiler we require access to local variables on the second and subsequent passes.
- The stack based symbol table deallocates, so it does not preserve information of local variables after the first pass.
- The stack based symbol table organization is not compatible with a N-pass compiler for N > 1.
- Also the complete symbol table would be useful for run-time diagnostics.
- So we may want to keep it after compilation is complete.

Dealing with the problem

- Don’t remove local variables when analysis of their associated block is complete.
- Ensure that outside the block, they are not accessible.

The most-closely nested rule

- The most-closely nested rule for blocks is that an identifier **x** is in the scope of the most-closely nested declaration of **x**;
- That is, the declaration of **x** found by examining blocks inside-out, starting with the block in which **x** appears.
- You may have heard this referred to as 'variable shadowing'

- The subscript is not part of an identifier; it is the line number of the declaration that applies to the identifier.
- All occurrences of **x** are within the scope of the declaration on line 1.

```
1.{ int $x_1$; int $y_1$;
2.   { int $w_2$; bool $y_2$; int $z_2$;
3.         ..$w_2$..;..$x_1$..; ..$y_2$..; ..$z_2$..;
4.    }
5.    .. $w_0$..; ..$x_1$..; ..$y_1$..;
6.}
```

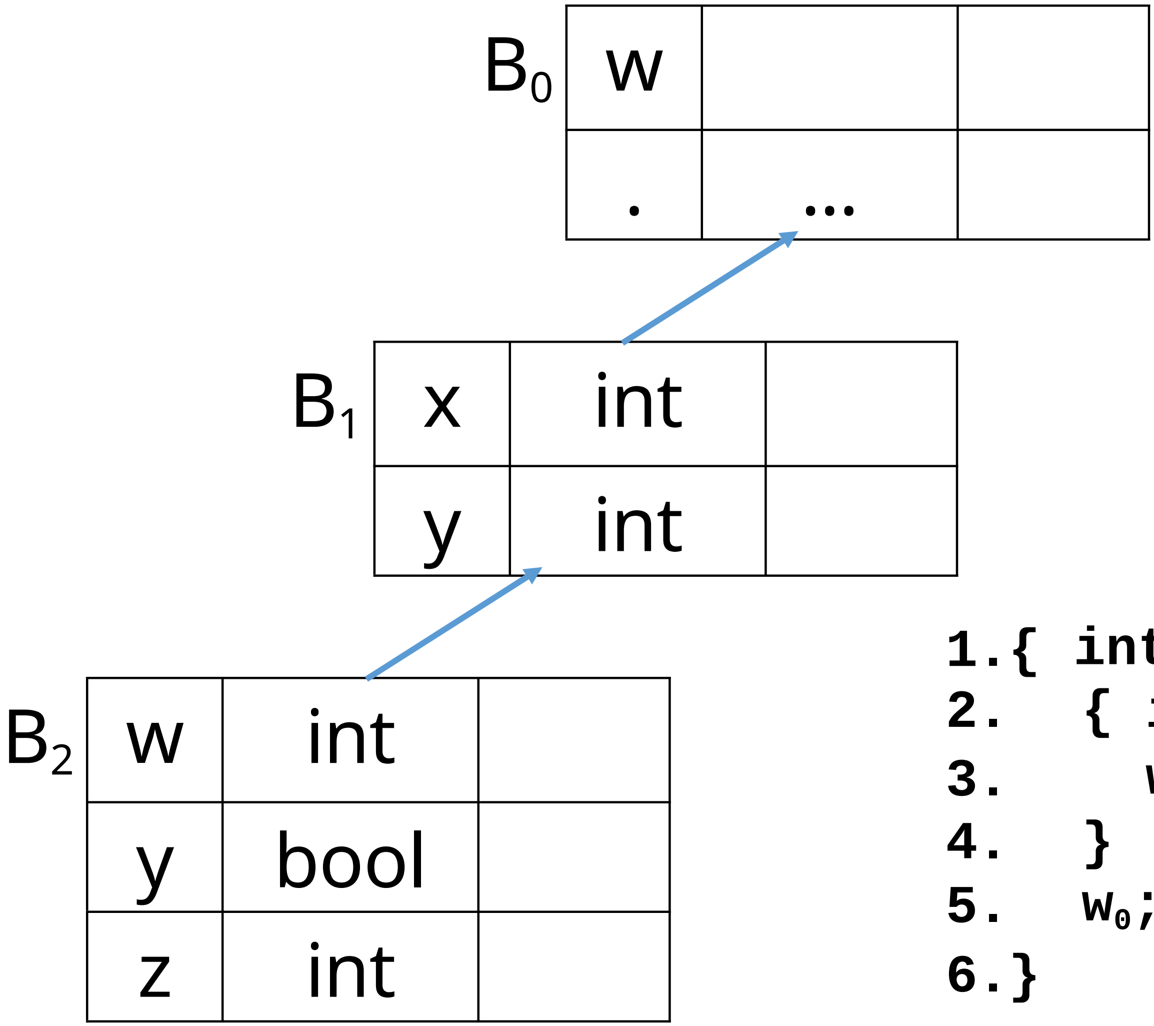
- The occurrence of **y** on line 3 is in the scope of the declaration of **y** on line 2 since **y** is redeclared within the inner block.
- The occurrence of **y** on line 5 is within the scope of the declaration of **y** on line 1.

```
1.{ int x_1; int y_1;
2.   { int w_2; bool y_2; int z_2;
3.         ..w_2..;..x_1..; ..y_2..; ..z_2..;
4.    }
5.    .. w_0..; ..x_1..; ..y_1..;
6.}
```

- The occurrence of `w` on line 5 is within the scope of a declaration of `w` outside this fragment.
- The subscript 0 denotes a declaration that is global or external to this block.

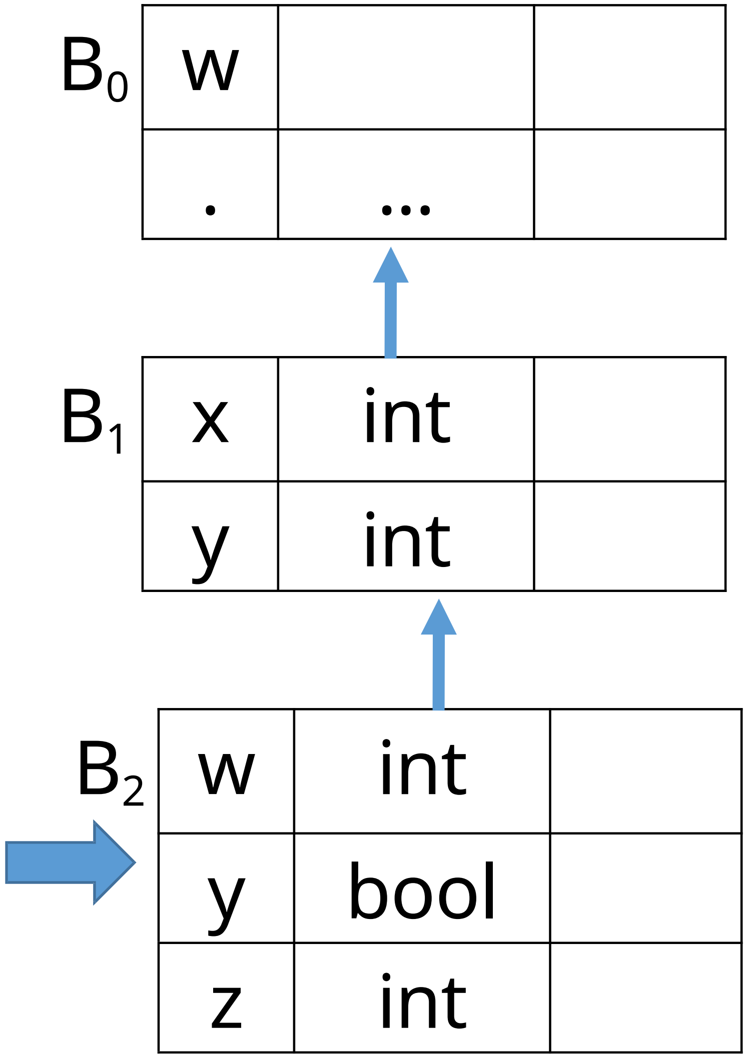
```
1.{ int x_1; int y_1;
2.   { int w_2; bool y_2; int z_2;
3.         ..w_2..;..x_1..; ..y_2..; ..z_2..;
4.    }
5.    .. w_0..; ..x_1..; ..y_1..;
6.}
```

The Symbol Table for our example



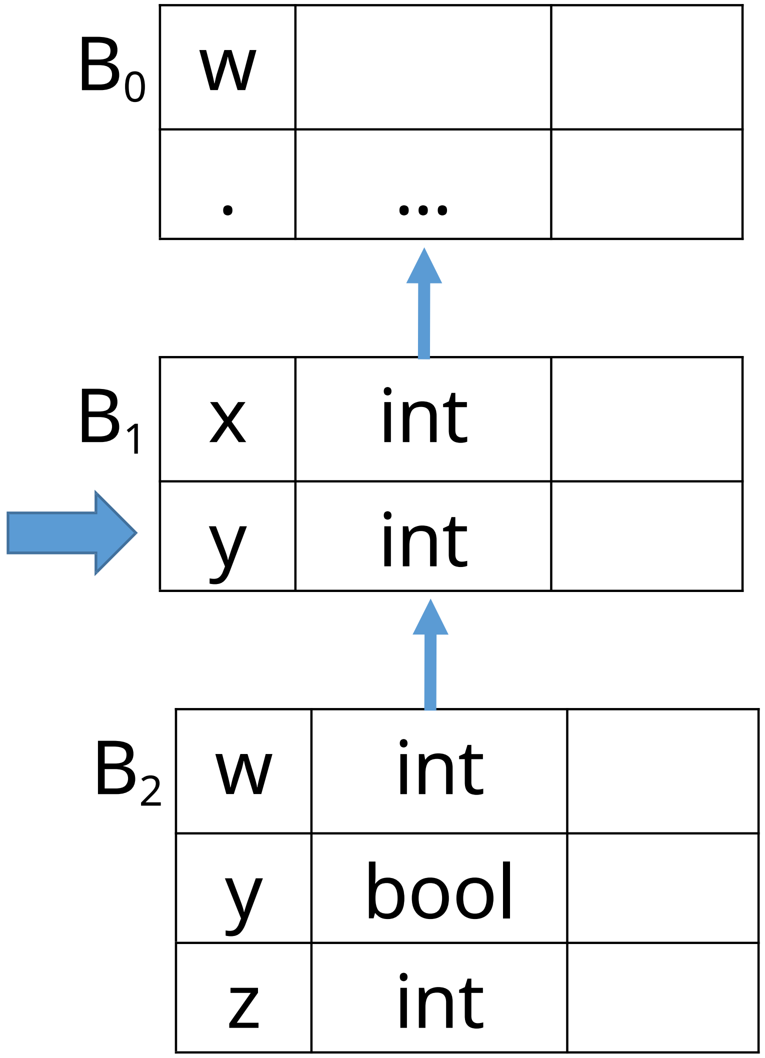
{ int x_1; int y_1;
  { int w_2; bool y_2; int z_2;
    w_2; x_1; y_2; z_2;
  }
  w_0; x_1; y_1;
}

The Symbol Table for our example



- B_0 is for any global declarations
- B_1 is for the block starting on line 1
- B_2 for the block starting on line 2
- When we are analysing lines 2 to 4, the environment is represented by a reference to the lowest symbol table : B_2.

The Symbol Table for our example



- When we move to line 5, the symbol table for B_2 becomes unaccessible.
- The environment instead refers to the symbol table for B_1, from which we can reach the global symbol table but not the one for B_2.

Learning Outcomes

- You should now appreciate the need for, and use of, the symbol table – in general and w.r.t scoping issues.