Compilers

Topic 4

The Symbol Table and Block Structure PART II

Mick O'Donnell: michael.odonnell@uam.es
Alfonso Ortega: alfonso.ortega@uam.es

Topic 2:

Block Structure

Symbol tables with block structures

Block Structure

Programming languages with block structures:

- Those languages in which some identifiers only exist inside some sections of the code, and not in others.
- For instance, Algol, PL/I, C, Java, Pascal…
- Example of program with blocks:

```
{ int  b, d;
   { float d, f;
     ...
     L1:...
   }
  { int i, h;
     L2:
        { int a;
        }
  }
}
```

Symbol tables with block structures

Block Structure

- **Scopes associated with each line of code:**
  - Every line of code is inside a set of blocks or ‘scopes’.
  - Each identifier is only valid within its associated blocks .
- **Current scope**:
  - the scope defined by the deepest block of the current line of code.
- **Open scopes**:
  - All the scopes that include a line of code (directly, or through other inner scopes) are said to be open with respect to that line.
- **Closed scopes**:
  - All the scopes that do not include a line of code (directly, or indirectly) are said to be closed with respect to that line.

  - Scope = Ambito o bloque

Symbol tables with block structures

Block Structure

- **Example**

```
{ int a, b, c;                         Scope 1
  a = 0; b = 0; c= 0;
  { int c, d;                        Scope 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                                  Scope 3
  print d
}
```

Symbol tables with block structures

Concepts

- **Rules to determine which identifiers are active:**
  - An identifier is active if:
    - It is defined in the current scope, or
    - It is defined in a scope which contains the current one.
  - The variable is only valid AFTER the point of declaration
  - A variable declaration applies for any block which is embedded within the block where the variable is defined, and starts after the declaration.
  - Whenever an identifier is declared both in this scope, and in a parent scope, then the declaration in the current scope is taken.
  - The names of the **arguments of a procedure** are local to that procedure, and they are not accessible from outside.
  - The same identifier cannot be defined twice at the same level.

Topic 3:

Use of Symbol Tables with Block Structure

7

Symbol tables with block structures

- In the following, we are assuming that we are operating within syntactic analysis of the program (and sometimes semantic analysis)
- This is because scopes (block structure) cannot be identified at the time of lexical analysis, which only sees one token at a time, not how these tokens relate to each other.

8

Symbol tables with block structures

- In languages without scoping, a single symbol table suffices:
  - An identifier always refers to the same variable (or procedure)
- However, when embedded scopes are possible, the information associated with an identifier is particular to the scopes it is declared within
  - E.g., “fred” might refer to a string in one block and an int in another
- We thus need to store and retrieve information about symbols relative to the set of scopes that the current line is within.
- Two solutions:
  - A distinct symbol table for each scope, giving information only about identifiers visible in that scope.
  - A single symbol table, which details, for each identifier, which scopes it occurs in, and its attributes in that scope (e.g. its datatype)

9

Symbol tables with block structures

One symbol table per scope

One symbol table per scope : the basic approach:

- Each block is given a separate symbol table.
- When a line of code declares a variable (e.g., in the argument list of a procedure declaration, or in an explicit type declaration), an entry is created in the current scope’s symbol table.
- When a line of code refers to a variable, the compiler first tries to retrieve the identifier’s record from the current symbol table.
- If not found, the symbol table of the parent scope is tried
- This is repeated until an entry is found in an enclosing scope.
- If none found, then the variable is referenced without declaration – report error (assuming type declaration is required).

10

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                  Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                            Block 3
  print d
}
```

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                  Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                            Block 3
  print d
}
```

Event: enter new block

Action: Create Symbol Table ST1

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                  Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                   Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                             Block 3
  print d
}
```

Event: enter new block

Action: Create Symbol Table ST1

ST1

Symb	Token	Dtype	Init?

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                  Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                   Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                             Block 3
  print d
}
```

Event: recognise type dec

Action: Add to current symbol table

ST1

Symb	Token	Dtype	Init?
a	Id	Int	No
b	Id	Int	No
c	Id	Int	No

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                 Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                           Block 3
  print d
}
```

Event: variable assignment

Query: Found in current ST? Yes

Action: Mark as initiated

ST1

Symb	Token	Dtype	Init?
a	Id	Int	**Yes**
b	Id	Int	No
c	Id	Int	No

15

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                 Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                           Block 3
  print d
}
```

Event: variable assignment

Query: Found in current ST? Yes

Action: Mark as initiated

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	**Yes**
c	Id	Int	No

16

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                         Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                     Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;                      Block 3
  }
  print d
}
```

Event: variable assignment

Query: Found in current ST? Yes

Action: Mark as initiated

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	**Yes**

17

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                         Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                     Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;                      Block 3
  }
  print d
}
```

Event: Enter new block

Action: Create new Symb Table

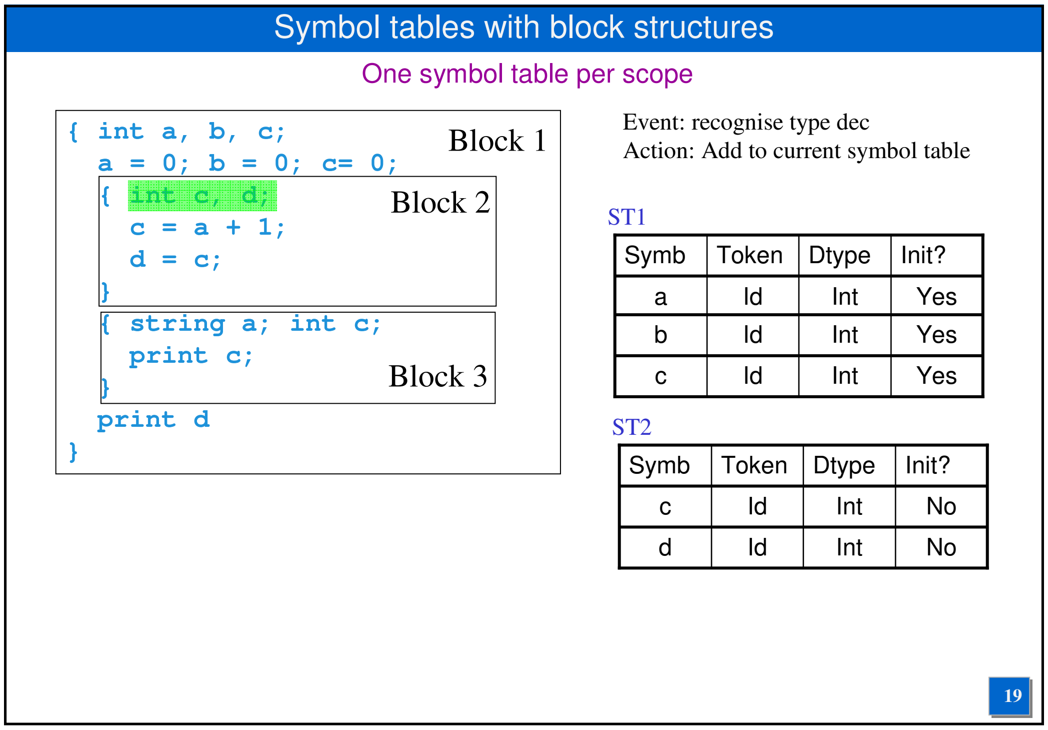
ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?

18



Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                   Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                          Block 3
  print d
}
```

Event: recognise type dec
Action: Add to current symbol table

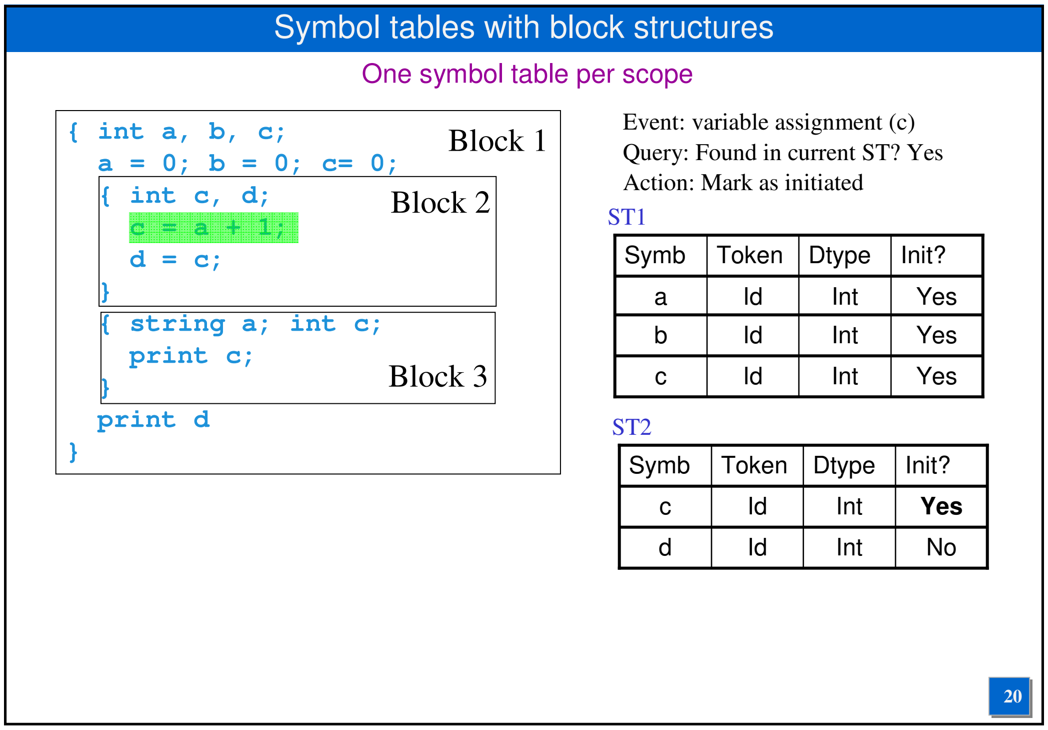
ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	No
d	Id	Int	No

19



Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                   Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                          Block 3
  print d
}
```

Event: variable assignment (c)
Query: Found in current ST? Yes
Action: Mark as initiated

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	**Yes**
d	Id	Int	No

20

Symbol tables with block structures

One symbol table per scope

```
{ int a, b, c;                          Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                      Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                                Block 3
  print d
}
```

Event: variable reference to *a*
Query: Found in current ST? NO
Query: Found in parent ST? YES
Query: Initiated in parent ST? YES

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	No

21

Symbol tables with block structures

One symbol table per scope

Note: the previous example demonstrates semantic checking

- When an identifier is found:
  - Variable Declaration:
    i) Return error if not first declaration of variable in current scope
    ii) Else, add entry for variable in current symbol table
  - Variable Assignment:
    i) Find the first scope in which the variable is declared
    ii) Return error if not declared in scope
    iii) Set the STATE to “initiated”
  - Variable Reference:
    i) Find the first scope in which the variable is declared
    ii) Return error if the variable is not declared and initiated

22

Symbol tables with block structures

One symbol table per scope : single pass

- **One pass compilers:**
  - Sometimes the compiler makes a single pass over the input token stream, and sometimes two or more passes are made.
  - If a single pass, then once a scope is exited, the symbol table of that scope will not be referenced again.
  - The symbol table of a closed scope can thus be cleared from memory (for memory efficiency).
  - Therefore, in one-pass compilers, the list of open scopes can be implemented as a LIFO stack.

23

Symbol tables with block structures

One symbol table per scope : single pass

```
{ int a, b, c;                        Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                     Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                               Block 3
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	No

24

Symbol tables with block structures

One symbol table per scope : single pass

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                   Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                             Block 3
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	Yes

25

Symbol tables with block structures

One symbol table per scope : single pass

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                   Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                             Block 3
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	Yes

26

Symbol tables with block structures

One symbol table per scope : single pass

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                 Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;                  Block 3
  }
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

27

Symbol tables with block structures

One symbol table per scope : single pass

```
{ int a, b, c;                    Block 1
  a = 0; b = 0; c= 0;
  { int c, d;                 Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;                  Block 3
  }
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST3

Symb	Token	Dtype	Init?

28

Symbol tables with block structures

One symbol table per scope: single pass

- Insertion algorithm:

```
status insert(id key, ScopeTable T)
{
    if exists(key, current_scope(T) return FAIL;
    insert_key( key, current_scope(T) );
}
```

29

Symbol tables with block structures

One table per scope, single pass

- **Search algorithm:**

```
Hash_entry search(id key, ScopeTable T)
{
    HashTable symbTable;

    symbTable = current_scope(T);
    while (not empty_hash_table(symbTable))
    {
            If exists(key, symbTable) return symbTable;
            symbTable=next_table(symbTable,T);
    }
    return NULL;
}
```

30

Symbol tables with block structures

One table per scope: multiple passes

- Multiple Pass Compilers:
  - However, if the compiler makes multiple passes, then each scope's symbol table needs to be preserved for later user.
  - One cannot use a LIFO stack.
  - One solution:
    - Data structure to record structure of scopes (e.g., an object for each scope, with a pointer to its parent scope)
    - Each scope object also has a pointer to the symbol table for that scope.

Symbol tables with block structures

One symbol table per scope: multiple passes

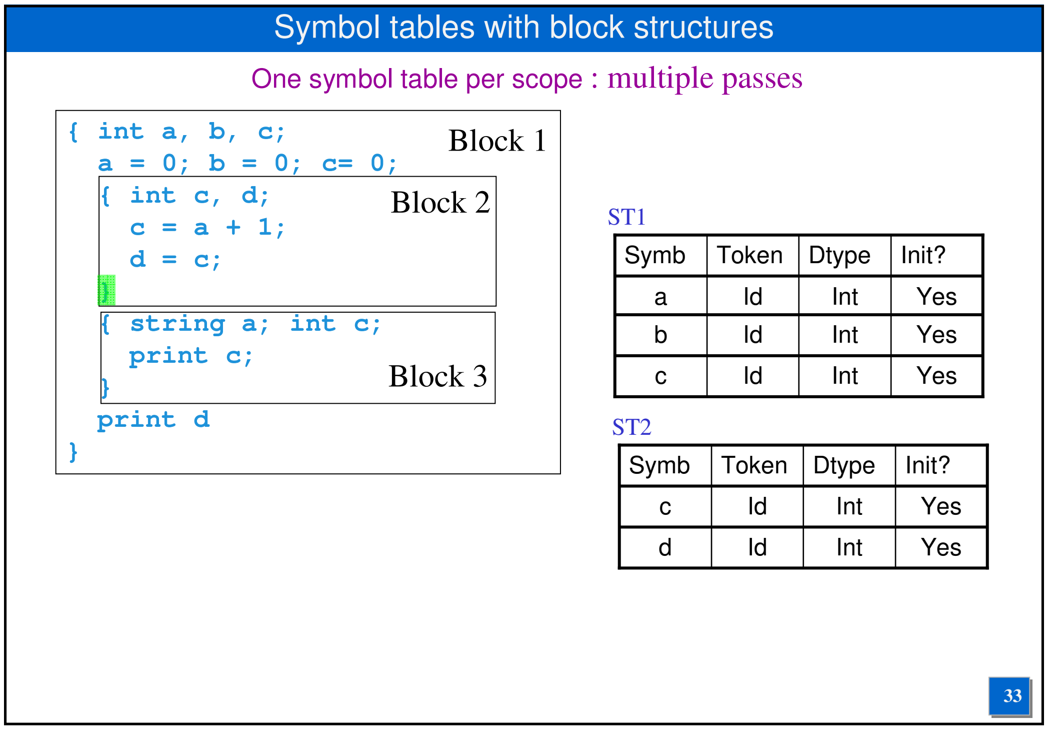
```
{ int a, b, c;                Block 1
  a = 0; b = 0; c= 0;
  { int c, d;               Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;
  }                         Block 3
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	Yes



Symbol tables with block structures

One symbol table per scope : multiple passes

```
{ int a, b, c;                  Block 1
  a = 0; b = 0; c= 0;
  { int c, d;               Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;                Block 3
  }
  print d
}
```

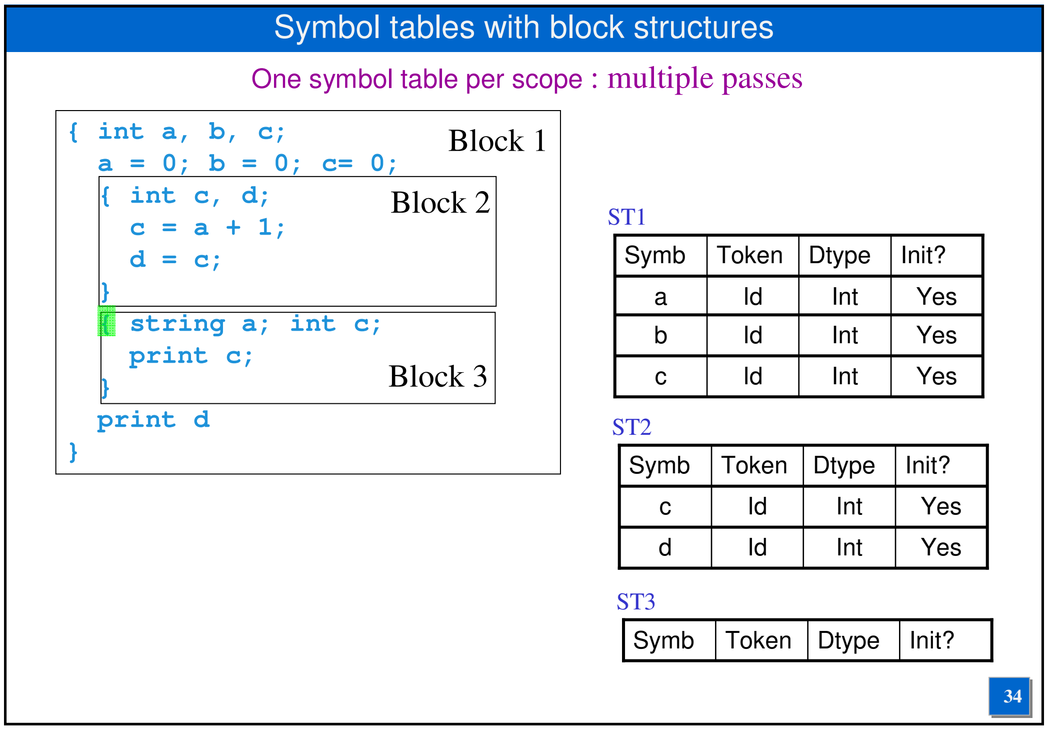
ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	Yes

33



Symbol tables with block structures

One symbol table per scope : multiple passes

```
{ int a, b, c;                  Block 1
  a = 0; b = 0; c= 0;
  { int c, d;               Block 2
    c = a + 1;
    d = c;
  }
  { string a; int c;
    print c;                Block 3
  }
  print d
}
```

ST1

Symb	Token	Dtype	Init?
a	Id	Int	Yes
b	Id	Int	Yes
c	Id	Int	Yes

ST2

Symb	Token	Dtype	Init?
c	Id	Int	Yes
d	Id	Int	Yes

ST3

Symb	Token	Dtype	Init?

34

Symbol tables with block structures

One symbol table per scope : multiple passes

- Multiple Pass Compilers:
  - Note that the set of symbol tables is not stored as a stack.
  - Better to use a tree structure, each symbol tale pointing to the symbol table for the scope that contains it.
  - Open scopes = current scope plus all parent scopes back to the root.

$$\begin{tikzcd}
 & \boxed{ST2} \arrow[dl] & \\
\boxed{ST1} & & \boxed{ST4} \arrow[dl] \\
 & \boxed{ST3} \arrow[ul] & \\
 & & \boxed{ST4} \arrow[ul]
\end{tikzcd}$$

35

Topic 4:

Dealing with Block Structure with a Single Symbol Table

36

Symbol tables with block structures

Just one symbol table

- An alternative is to use just a single hash table.
- All the identifiers will be inserted in the same table.
- Each identifier will have information about the scope to which it belongs.
- **Possible implementation:**
  - Using
    - An array for the data about the blocks.
    - Other for the identifiers. It may be just one structure that uses the end for the identifiers of the open blocks, and the beginning for the ids. of the closed ones. On the other hand, there might be two: one for the open scopes and one for the closed scopes.
  - Each block will contain the following information:
    - Block number.
    - Number of the parent block.
    - Number of identifiers in this block.
    - Pointer to the information of the identifiers in this block.

37

Symbol tables with block structures

Multiple scopes, Just one symbol table

- The problem:
  - Each hash table requires memory overhead
    - Thus using a hash table for each scope is memory inefficient
  - We may be required to search in multiple symbol tables,
    - Thus less efficient for speed as well
- Solution:
  - Use a single hash table for all scopes
  - Less memory overhead
  - (perhaps) Only one search required.

38

Symbol tables with block structures

Multiple scopes, Just one symbol table

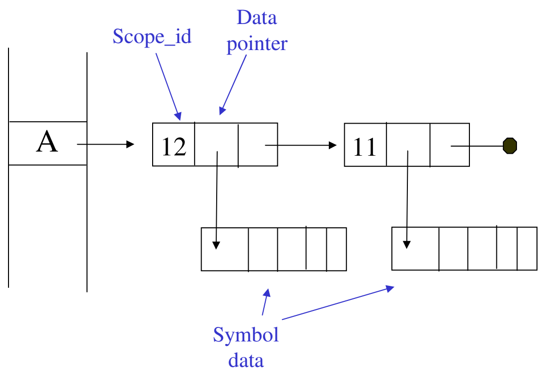
- Solution 1:
  - Each scope has a unique number
    1. Start scope_id = 0 (for global scope)
    2. For each new scope, increment scope_id
  - When entering an identifier into the symbol table, use "identifier,scope_id" as the key.
  - To find the symbol table entry for an identifier
    1. Append the current scope_id to the identifier and look for that
    2. If that fails, try the parent scope, etc. until it is found

    BUT still may require multiple searches!

Symbol tables with block structures

Multiple scopes, Just one symbol table

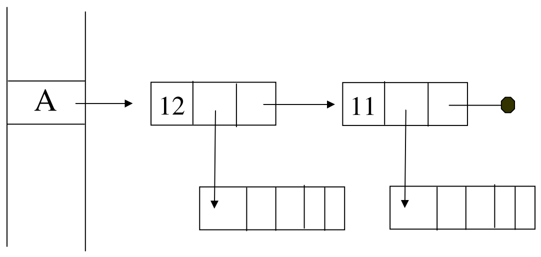
- Solution 2: identifier returns linked list of usages of identifier
  - Symbol table:
    - Key = identifier
    - Value = linked list of pointers to records, one for each usage of the identifier.



Symbol tables with block structures

Multiple scopes, Just one symbol table

- If a single pass compiler,
  1. Each new insertion of an identifier is at front of the linked list
  2. A separate structure keeps track of all identifiers <u>defined</u> in the current scope
  3. When exiting the scope, one uses the list in (2) to delete the frontmost element in each linked list.

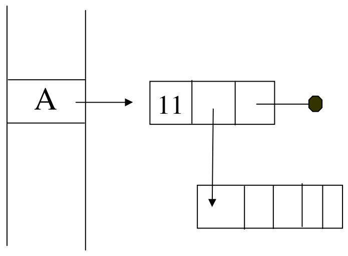


41

Symbol tables with block structures

Multiple scopes, Just one symbol table

- If a single pass compiler,
  1. Each new insertion of an identifier is at front of the linked list
  2. A separate structure keeps track of all identifiers <u>defined</u> in the current scope
  3. When exiting the scope, one uses the list in (2) to delete the frontmost element in each linked list.

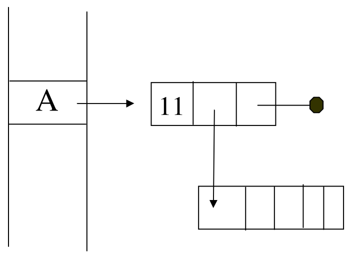


42

Symbol tables with block structures

Multiple scopes, Just one symbol table

- Using this approach:
  - all entries on the symbol table are in open scopes (potentially available within the current block of code)
  - The front item on the linked list is always the appropriate one in the current scope (it was either defined in the current scope, or in a parent scope of the current scope)



43

Summary of Symbol Tables

1. Symbol table for languages without block structure
2. Symbol table for languages with block structure
   1. One Table per scope: single pass
   2. One Table per scope: Multiple Pass
   3. One Table for all scopes

44

Past Exam Questions

During the semantic analysis, the table of symbols is used to verify that the identifiers are used in an appropriate way. What semantic verifications are carried out using the table of symbols when:

a) se declara una variable (por ejemplo, int **v**;)

b) se le asigna valor a una variable (por ejemplo, **v** = 7;)

c) se hace referencia a una variable (por ejemplo, w = **v** * 8;)

- PAST EXAM QUESTIONS

1. Dado el siguiente programa

```
{                               // Línea 1
    int x,y;                    // Línea 2
    {                           // Línea 3
        int z;                  // Línea 4
        {                       // Línea 5
            int p;              // Línea 6      ⇐
        }                       // Línea 7
        int q;                  // Línea 8      ⇐
        {                       // Línea 9
            int r;              // Línea 10
            r = z + 3;          // Línea 11
        }                       // Línea 12
        int s;                  // Línea 13     ⇐
    }                           // Línea 14
    {                           // Línea 15
        int t;                  // Línea 16
    }                           // Línea 17
}                               // Línea 18
```

(a) Mostrar el contenido de la tabla de símbolos después de compilar cada una de las líneas marcadas con ⇐ suponiendo que se utiliza un compilador de una pasada y una tabla de símbolos por bloque. Los bloques deben nombrarse utilizando la notación Bn, donde n es un numero distinto para cada bloque. Para cada identificador solo debe incluirse su nombre.

(b) Al compilar la instrucción de la línea 8 ¿cuántas operaciones de búsqueda se llevarán a cabo y en qué orden? Especificar para cada una de ellas el motivo de la búsqueda y a qué bloque corresponde la tabla en la que se realiza la búsqueda.

(c) Al compilar la instrucción de la línea 11 ¿cuántas operaciones de búsqueda se llevarán a cabo y en qué orden? Especificar para cada una de ellas el motivo de la búsqueda y a qué bloque corresponde la tabla en la que se realiza la búsqueda.

- 1. Se desea realizar un compilador de una pasada para el lenguaje ASPLE que está estudiando en las prácticas y que, por lo tanto, admite funciones. Para su tabla de símbolos se va a utilizar una tabla hash para cada ámbito. Considere los siguientes programas ASPLE. Para cada uno de ellos se le pide:

```
begin
  int a;
  int b;
  function int FRara(int a;
                     int c)
  begin
   int c;

# (1)
   input c;
   return a+b+c
  end
# (2)
  input a;
  input b;
  output FRara(a,c)
end
```

```
begin
   int a;
   int b;
   function int FRara(int a)
   begin
           int c;
# (1)

           input c;
           return a+b+c
   end
# (2)
   input a;
   input b;
   output FRara(a)
end
```

(1) Indicar explícitamente los errores de compilación, en el caso de que se produzcan. Explique brevemente la causa de los mismos.

(2) Si los programas son correctos, muestre el contenido de las tablas de símbolos (completas) en los puntos marcados con (1) y (2). Explique brevemente el contenido de las tablas.