Symbol Tables in Block Structured languages

March 12, 2020

Block Structured Languages

- Basic building units are blocks.
- There can be nested blocks upto any depth.

Block Structured Languages

- Basic building units are blocks.
- There can be nested blocks upto any depth.

```
float x=10,y=5,z=30;
int main()
{
    float y=40;
    if(1){
        float x=50;
    }
    if(y<z){
        printf("z=%f",z);
    }
    else{
            if(y<x){
                printf("z=%f",z);
            }
            else{
                printf("x=%f",x);
            }
    }
}
```

```
float x=10,y=5,z=30;
int main()
{
    float y=40;
    if(1){
        float x=50;
    }
    if(y<z){//y=40,z=30
        printf("z=%f",z);
    }
    else{
            if(y<x){
                printf("z=%f",z);
            }
            else{
                printf("x=%f",x);
            }
    }
}
```

```
float x=10,y=5,z=30;
int main()
{
    float y=40;
    if(1){
        float x=50;
    }
    if(y<z){//y=40,z=30
        printf("z=%f",z);
    }
    else{
            if(y<x){//y=40,x=10
                printf("z=%f",z);
            }
            else{
                printf("x=%f",x);
            }
    }
}
```

Scope of Identifiers

Definition

Scope of identifier x: refers to a portion of a program that the identifier is visible.

- Same identifier can be declared and used for different purpose in different parts of the program.
- Same method names can appear in subclasses to override a method in super class.

**Scope of an identifier is from the recent opening brace ('{') it has seen, until the corresponding matching closing brace ('}').

Scope of Identifiers

Definition

Scope of identifier x: refers to a portion of a program that the identifier is visible.

- Same identifier can be declared and used for different purpose in different parts of the program.
- Same method names can appear in subclasses to override a method in super class.

**Scope of an identifier is from the recent opening brace ('{') it has seen, until the corresponding matching closing brace ('}').

Scope of Identifiers

Definition

Scope of identifier x: refers to a portion of a program that the identifier is visible.

- Same identifier can be declared and used for different purpose in different parts of the program.
- Same method names can appear in subclasses to override a method in super class.

**Scope of an identifier is from the recent opening brace ('{') it has seen, until the corresponding matching closing brace ('}').

Scope of Identifiers

Definition

Scope of identifier x: refers to a portion of a program that the identifier is visible.

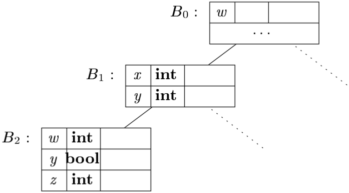
- Same identifier can be declared and used for different purpose in different parts of the program.
- Same method names can appear in subclasses to override a method in super class.

**Scope of an identifier is from the recent opening brace ('{') it has seen, until the corresponding matching closing brace ('}').

Scope Example

**Subscripts are just for the sake of distinguishing //Occurrence of *w*, possibly within the scope of *w* outside this segment of code.

```
{    int $x_1$; int $y_1$;
     {    int $w_2$; bool $y_2$; int $z_2$;
               ⋯ $w_2$ ⋯; ⋯ $x_1$ ⋯; ⋯ $y_2$ ⋯; ⋯ $z_2$ ⋯;
     }
          ⋯ $w_0$ ⋯; ⋯ $x_1$ ⋯; ⋯ $y_1$ ⋯;
}
```



B_0 :

w		
⋯		

B_1 :

x	**int**	
y	**int**	

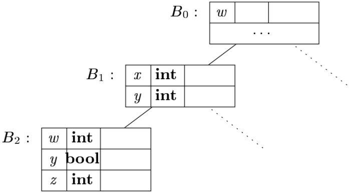
B_2 :

w	**int**	
y	**bool**	
z	**int**	

Scope Example

**Subscripts are just for the sake of distinguishing //Occurrence of *w*, possibly within the scope of *w* outside this segment of code.

```
{    int $x_1$;  int $y_1$;
     {    int $w_2$;  bool $y_2$;  int $z_2$;
              $\cdots w_2 \cdots$;  $\cdots x_1 \cdots$;  $\cdots y_2 \cdots$;  $\cdots z_2 \cdots$;
     }
          $\cdots w_0 \cdots$;  $\cdots x_1 \cdots$;  $\cdots y_1 \cdots$;
}
```



Solve the following task

Input:

```
{ int x; char y; { bool y; x; y; } x; y; }
```

Output:

```
{ { x:int; y:bool; } x:int; y:char; }
```

Design

- Whenever an '{' is being processed, we create and link (using the *prev* field in the given implementation) a new *Symbol Table Node* to the current node (pointed by *top* variable in the given Implementation) in the Table-Tree.
- Whenever an '}' is being processed, delete the current *Symbol Table Node* and **move to the parent node** in the Table-Tree.
- For each newly declared variable, add it to the *Current Symbol Table Node*.
- Whenever a variable in use (say $x = y + z;$), check whether the variable is present in the Current Symbol Table Node. If not, **move to the parent node** in the Table-Tree and repeat the same.

  **Note that every parent node in table tree corresponds to a block which is active (have seen '{', but yet to see matching closing '}')

Design

- Whenever an '{' is being processed, we create and link (using the *prev* field in the given implementation) a new *Symbol Table Node* to the current node (pointed by *top* variable in the given Implementation) in the Table-Tree.
- Whenever an '}' is being processed, delete the current *Symbol Table Node* and **move to the parent node** in the Table-Tree.
- For each newly declared variable, add it to the *Current Symbol Table Node*.
- Whenever a variable in use (say $x = y + z;$), check whether the variable is present in the Current Symbol Table Node. If not, **move to the parent node** in the Table-Tree and repeat the same.
  **Note that every parent node in table tree corresponds to a block which is active (have seen '{', but yet to see matching closing '}')

Design

- Whenever an '{' is being processed, we create and link (using the *prev* field in the given implementation) a new *Symbol Table Node* to the current node (pointed by *top* variable in the given Implementation) in the Table-Tree.
- Whenever an '}' is being processed, delete the current *Symbol Table Node* and **move to the parent node** in the Table-Tree.
- For each newly declared variable, add it to the *Current Symbol Table Node*.
- Whenever a variable in use (say $x = y + z;$), check whether the variable is present in the Current Symbol Table Node. If not, **move to the parent node** in the Table-Tree and repeat the same.
  **Note that every parent node in table tree corresponds to a block which is active (have seen '{', but yet to see matching closing '}')

Design

- Whenever an '{' is being processed, we create and link (using the *prev* field in the given implementation) a new *Symbol Table Node* to the current node (pointed by *top* variable in the given Implementation) in the Table-Tree.
- Whenever an '}' is being processed, delete the current *Symbol Table Node* and **move to the parent node** in the Table-Tree.
- For each newly declared variable, add it to the *Current Symbol Table Node*.
- Whenever a variable in use (say $x = y + z;$), check whether the variable is present in the Current Symbol Table Node.

  If not, **move to the parent node** in the Table-Tree and repeat the same.

  **Note that every parent node in table tree corresponds to a block which is active (have seen '{', but yet to see matching closing '}')

Design

- Whenever an '{' is being processed, we create and link (using the *prev* field in the given implementation) a new *Symbol Table Node* to the current node (pointed by *top* variable in the given Implementation) in the Table-Tree.
- Whenever an '}' is being processed, delete the current *Symbol Table Node* and **move to the parent node** in the Table-Tree.
- For each newly declared variable, add it to the *Current Symbol Table Node*.
- Whenever a variable in use (say $x = y + z;$), check whether the variable is present in the Current Symbol Table Node. If not, **move to the parent node** in the Table-Tree and repeat the same.

  **Note that every parent node in table tree corresponds to a block which is active (have seen '{', but yet to see matching closing '}')

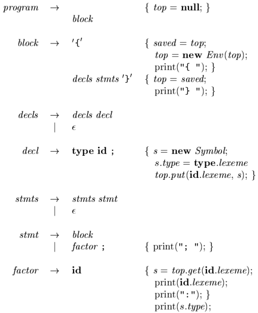
Design

- Whenever an '{' is being processed, we create and link (using the *prev* field in the given implementation) a new *Symbol Table Node* to the current node (pointed by *top* variable in the given Implementation) in the Table-Tree.
- Whenever an '}' is being processed, delete the current *Symbol Table Node* and **move to the parent node** in the Table-Tree.
- For each newly declared variable, add it to the *Current Symbol Table Node*.
- Whenever a variable in use (say $x = y + z$;), check whether the variable is present in the Current Symbol Table Node. If not, **move to the parent node** in the Table-Tree and repeat the same.
  **Note that every parent node in table tree corresponds to a block which is active (have seen '{', but yet to see matching closing '}')

Implementation

```
package symbols;
import java.util.*;
public class Env {
    private Hashtable table;
    protected Env prev;

    public Env(Env p) {
        table = new Hashtable(); prev = p;
    }

    public void put(String s, Symbol sym) {
        table.put(s, sym);
    }
    public Symbol get(String s) {
        for( Env e = this; e != null; e = e.prev ) {
            Symbol found = (Symbol)(e.table.get(s));
            if( found != null ) return found;
        }
        return null;
    }
}
```

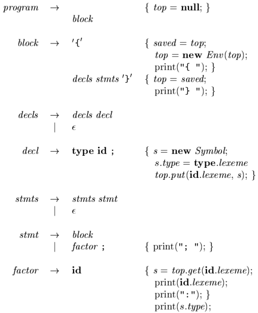
The Parser



```
program  →                        { top = null; }
              block
  block  →    '{'                 { saved = top;
                                    top = new Env(top);
                                    print("{ "); }
              decls stmts '}'     { top = saved;
                                    print("} "); }

  decls  →    decls decl
         |    ϵ

   decl  →    type id ;           { s = new Symbol;
                                    s.type = type.lexeme
                                    top.put(id.lexeme, s); }

  stmts  →    stmts stmt
         |    ϵ

   stmt  →    block
         |    factor ;            { print("; "); }

 factor  →    id                  { s = top.get(id.lexeme);
                                    print(id.lexeme);
                                    print(":"); }
                                    print(s.type);
```

***Corresponding to each node in the parse tree which is being the head (LHS) of some production, there is a distinct copy of the code fragment (that is there in the action part) getting executed like a function call (and hence backs up the data associated with old copy on the stack).

The Parser



```
 program  →                    { top = null; }
               block

   block  →    '{'             { saved = top;
                                 top = new Env(top);
                                 print("{ "); }
               decls stmts '}' { top = saved;
                                 print("} "); }

   decls  →    decls decl
          |    ϵ

    decl  →    type id ;       { s = new Symbol;
                                 s.type = type.lexeme
                                 top.put(id.lexeme, s); }

   stmts  →    stmts stmt
          |    ϵ

    stmt  →    block
          |    factor ;        { print("; "); }

  factor  →    id              { s = top.get(id.lexeme);
                                 print(id.lexeme);
                                 print(":"); }
                                 print(s.type);
```

***Corresponding to each node in the parse tree which is being the head (LHS) of some production, there is a distinct copy of the code fragment (that is there in the action part) getting executed like a function call (and hence backs up the data associated with old copy on the stack).