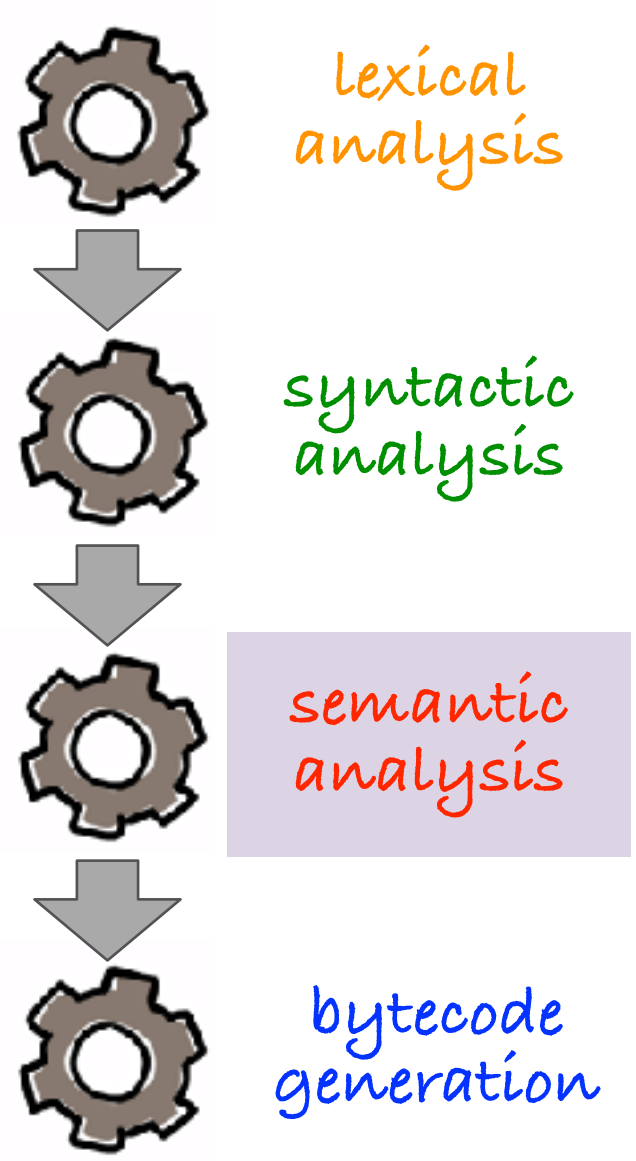
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

SEMANTIC ANALYSIS

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 72671 - COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

THIS LECTURE



- scopes and symbol tables
- type checking

the `SimpLan` interpreter

OUTLINE

- how to build the symbol table
- how to use it to find
  - multiple declared identifiers
  - undeclared identifiers
- symbol tables in `ANTLR`

reference:

- Torben Morgensen: Basics of Compiler Design, Chapter 4

THE COMPILER SO FAR

lexical analysis

- *detects **inputs with illegal tokens** — e.g.* `mainf ();`

parsing

- detects inputs with ***ill-formed parse trees*** — e.g. missing semicolons

THE PURPOSE OF SEMANTIC ANALYSIS — AN EXAMPLE

an **erroneous** code

```
class MyClass implements MyInterface {
    string myInteger;
    void doSomething() {
        int[] x = new string;
        x[5] = myInteger * y;
    }
    void doSomething() {
    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n – 1);
    }
}
```

how many **errors** can you find?

THE PURPOSE OF SEMANTIC ANALYSIS — AN EXAMPLE

an **erroneous** code

```
class MyClass implements MyInterface {
    string myInteger;
    void doSomething() {
        int[] x = new string;
        x[5] = myInteger * y;
    }
    void doSomething() {
    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n – 1);
    }
}
```

Interface is not declared

wrong type

can't multiply strings

undeclared id

can't redefine functions

can't add void

no main function

how many **errors** can you find?

SEMANTIC ANALYSIS

- catches the errors that have not been found by the lexer and the parser
- typical **semantic errors**:
  - **undeclared variable**: a variable should not be used before being declared
  - **multiple declarations**: a variable should be declared (in the same scope) at most once
  - **type mismatch**: type of the left-hand side of an assignment should match the type of the right-hand side
  - **wrong arguments**: functions/methods should be called with the right number and types of arguments

question: why these errors cannot be caught before?

LIMITATIONS OF CONTEXT-FREE GRAMMARS

using context-free grammars

- how would you prevent **duplicate** identifier definitions?
- how would you differentiate variables of one type from variables of another type?
- how would you ensure classes implement all interface methods?

for most programming languages, these are **provably** impossible

- use the **pumping lemma for context-free languages**

A SIMPLE SEMANTIC ANALYSER

works in **two** phases

1. **ScopeChecking** — it traverses the AST created by the parser and, for each scope in the program:
   - **processes the declarations** that is **(a)** adds new entries to the symbol table and **(b)** reports any variables that are multiply declared
   - **processes the statements** that is **(a)** finds uses of undeclared variables and **(b)** updates the "ID" nodes of the AST to point to the appropriate symbol-table entry

   this is the `checkSemantics` method in `SimpLan`

2. **TypeChecking** — it traverses the AST (**again**!) and processes all the statements in the program
   - uses the symbol-table information to determine the type of each expression, and to find type errors

   this is the `typeCheck` method in `SimpLan`
   - actually there is an optimization: the symbol table is defined after the scope-checking and nodes only contain the relevant infos

A SIMPLE SEMANTIC ANALYSER

why there are **two** phases?

- because this simplifies the analysis
- because we show two different ways of raising errors
- because they return different values (list-of-errors and types)

this is my design choise: not sure it is better than the other choice (= make just one visit)

remark: modern semantic analysers performs **several visits** (not just two!) of the AST

- **because there are identifiers that are used before their declaration**
  - **methods in oo languages**
  - mutual recursive functions

SCOPE CHECKING: WHAT IS IN A NAME?

- the same name in programs of modern languages may refer to fundamentally different things

- this is a perfectly legal `Java` code:

```
public class A {
      char A;
      A A(A A) {
                  A.A = 'A';
                  return A((A) A);
      }
}
```

what all these `'A'` are?

SCOPE CHECKING: WHAT IS IN A NAME?

this is a perfectly legal C++ code:

what all these 'x' are?

```
int Awful() {
                int x = 137;
                {
                  string x = "Scope!";
                  if (float x = 0)
                          double x = x;
                }
                if (x == 137) cout << "Y";
}
```

SCOPES AND SYMBOL TABLES

the **scope of a declaration** is the set of locations in a program where the name refers to the declaration's name

- the introduction of new variables into scope may hide older names
- how do we keep track of **what's visible**?

we use **symbol tables**

- a symbol table is a map from a name to the thing that the name refers to
- as we run our semantic analysis, we continuously update the symbol table with information about what is in scope
- the symbol table design is **influenced by what kind of scoping rule** is used by the programming language

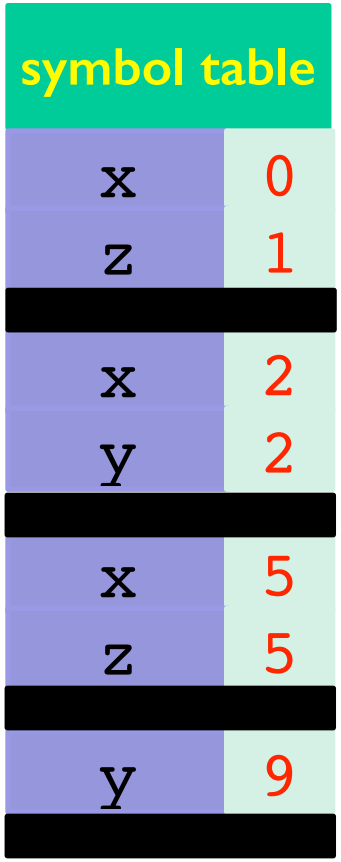
SCOPES AND SYMBOL TABLES

questions:

- what does the symbol table looks like in practice?
- what operations need to be defined on it and how do we implement it?

SYMBOL TABLES — A FIRST EXAMPLE

```
0:  int x = 137;
1:  int z = 42;
2:  int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n",x@2,y@2,z@1);
4:      {
5:        int x, z;
6:        z@5 = y@2;
7:        x@5 = z@5;
8:        {
9:          int y = x@5;
10:         {
11:           printf("%d,%d,%d\n",x@5,y@9,z@5);
12:         }
13:         printf("%d,%d,%d\n",x@5,y@9,z@5);
14:       }
15:       printf("%d,%d,%d\n",x@5,y@2,z@5);
16:     }
17: }
```

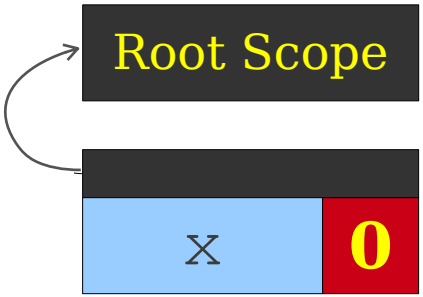


SYMBOL TABLES — A SECOND EXAMPLE

Root Scope

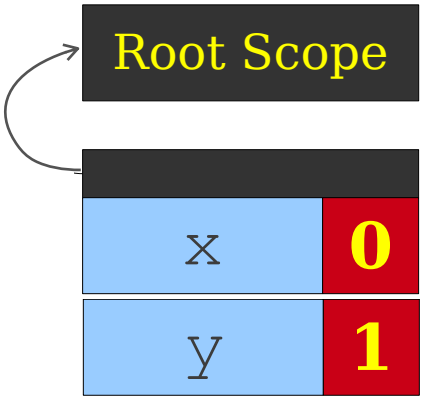
```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:      int w, z;
5:        {
6:          int y;
7:        }
8:        {
9:          int w;
10:       }
11:   }
```

SYMBOL TABLES — A SECOND EXAMPLE




```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:      int w, z;
5:        {
6:          int y;
7:        }
8:        {
9:          int w;
10:       }
11:   }
```

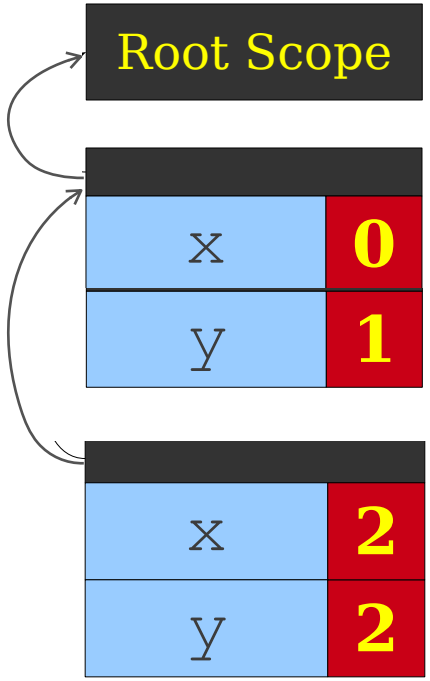
SYMBOL TABLES — A SECOND EXAMPLE




```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```

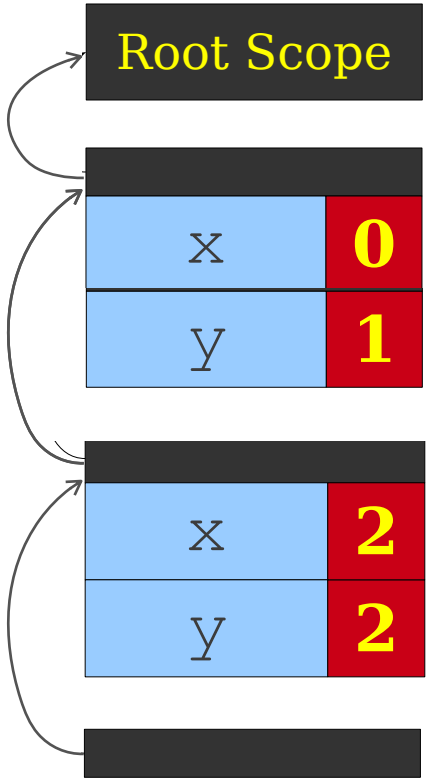
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



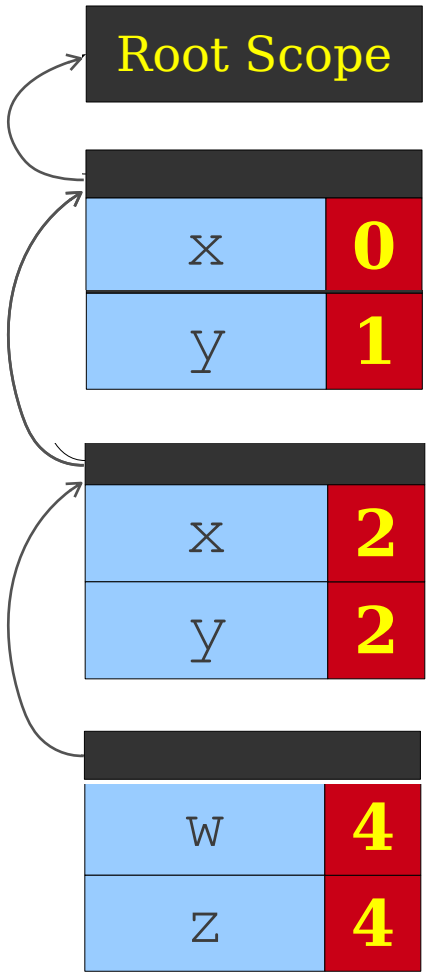
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



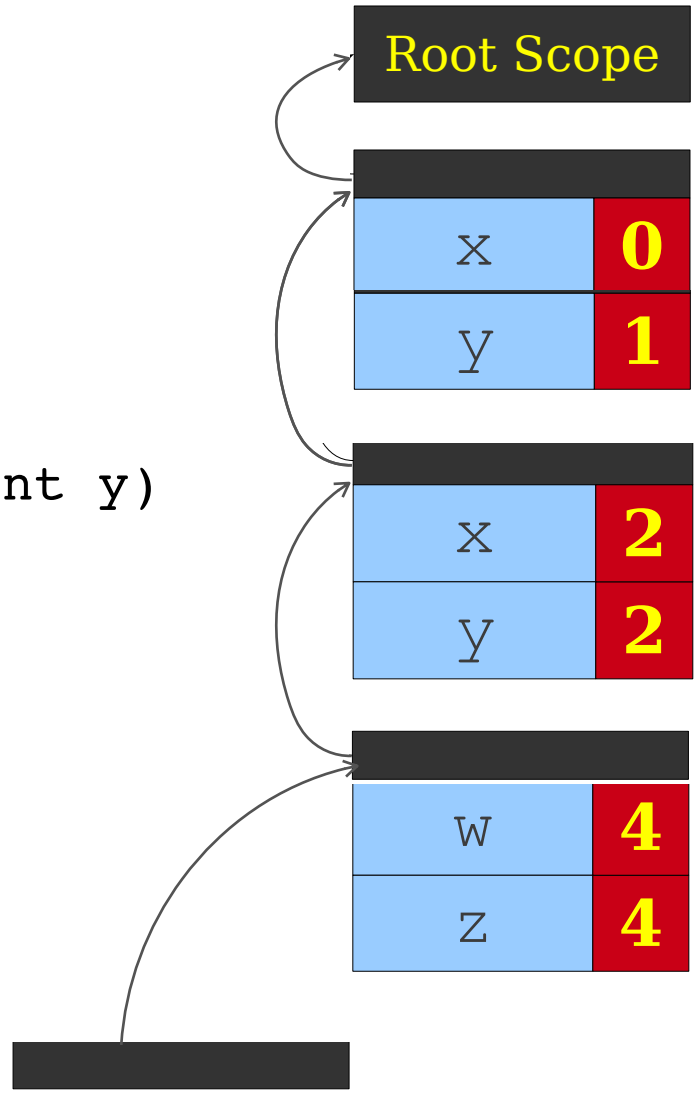
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



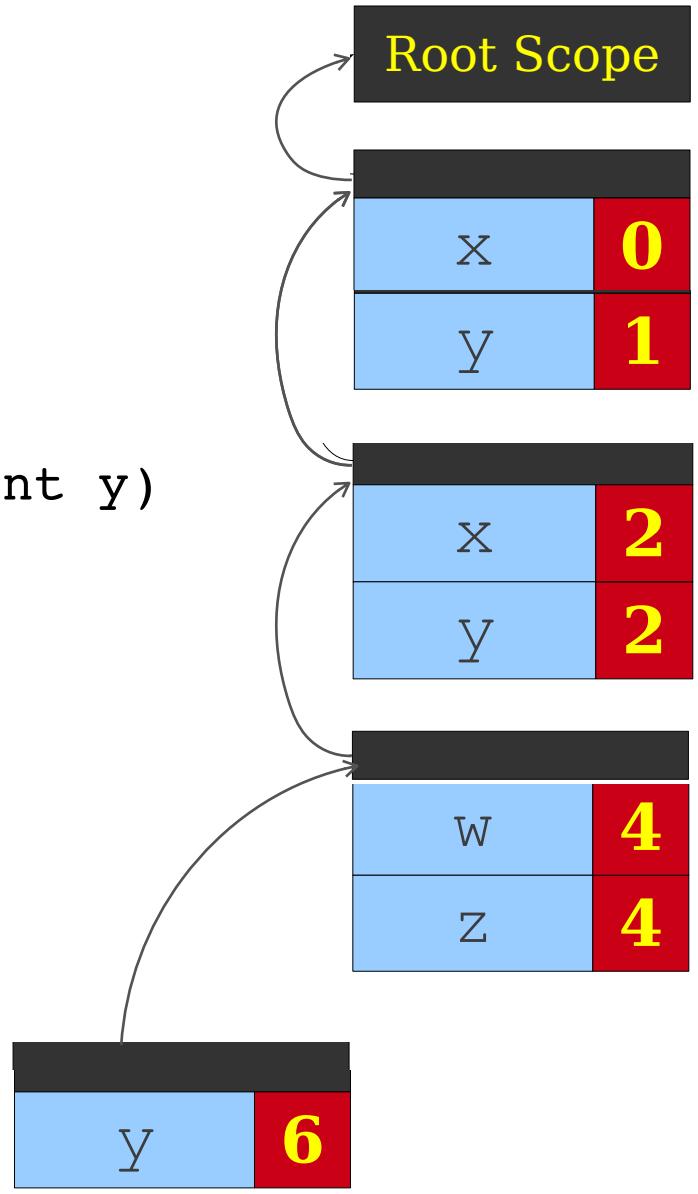
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



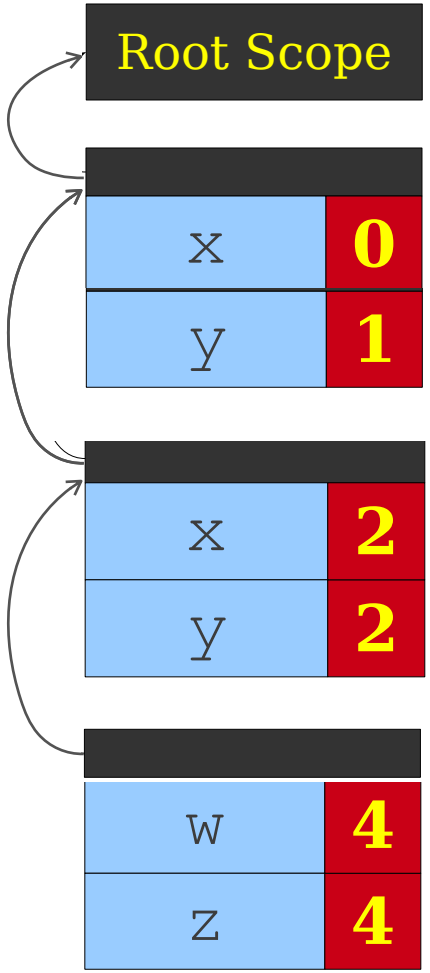
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



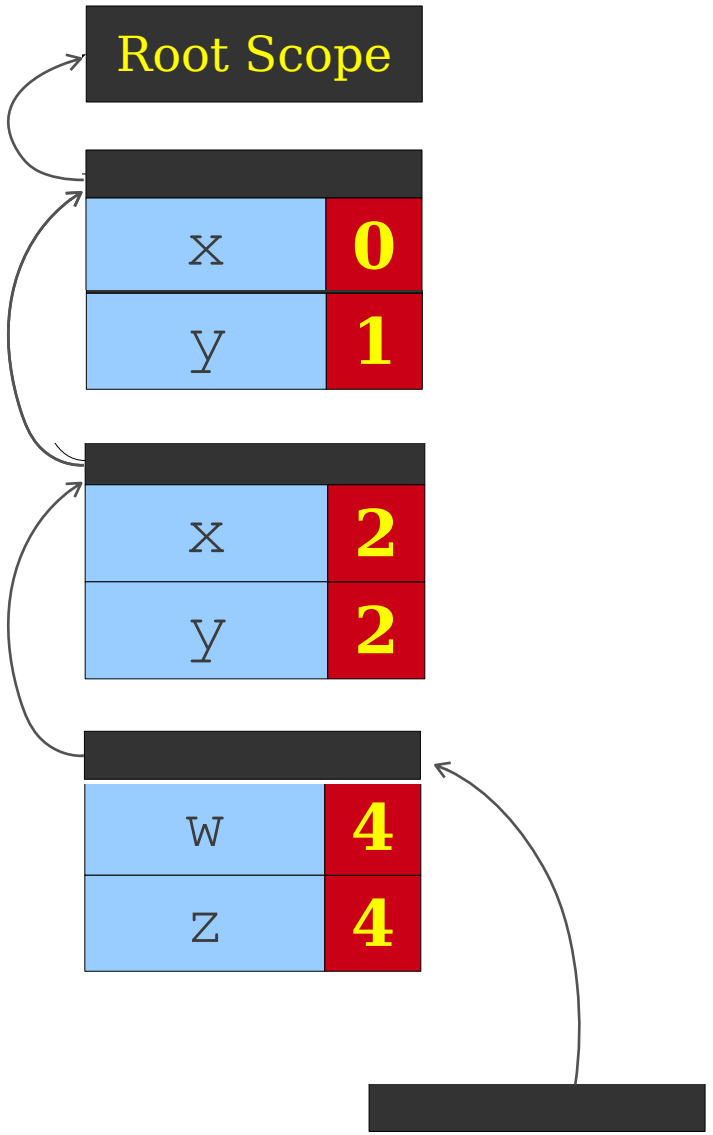
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



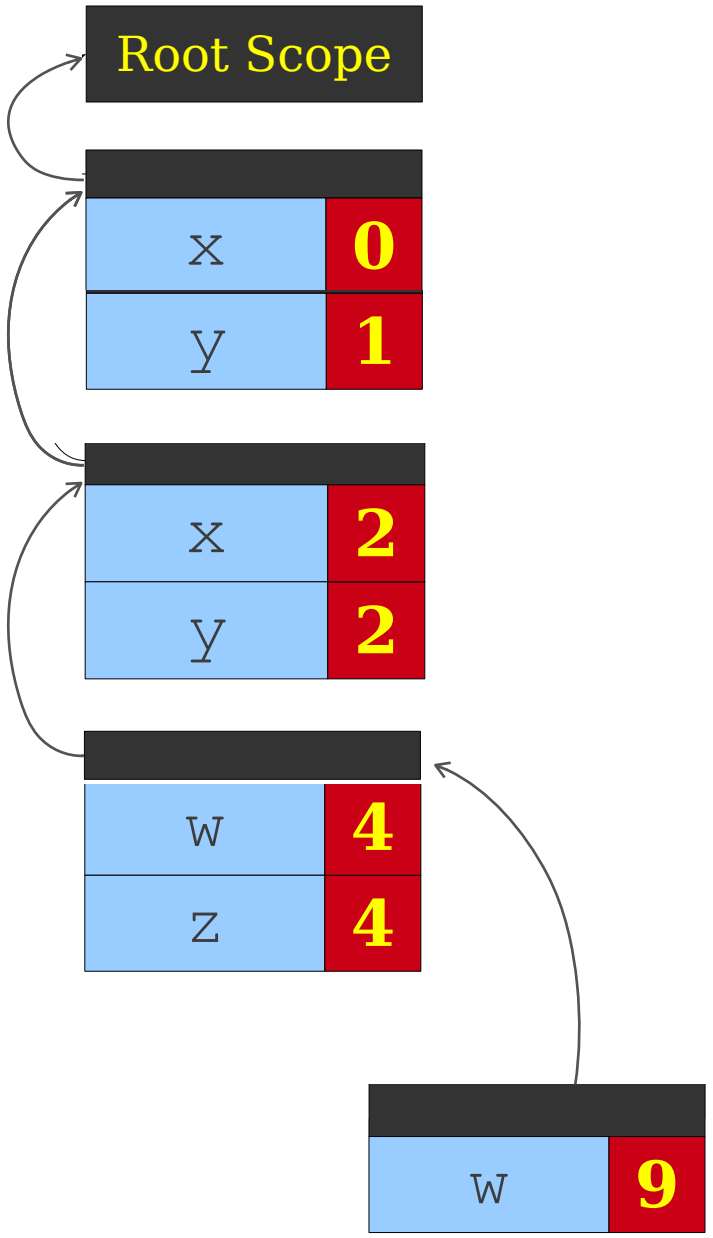
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



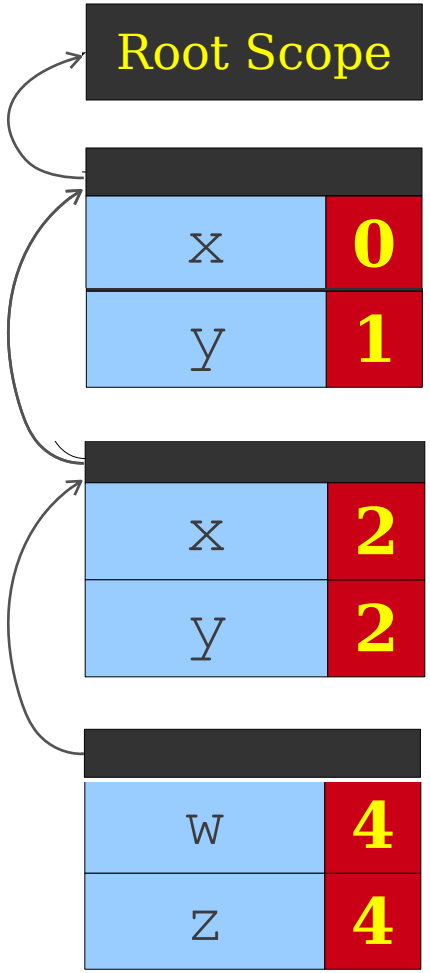
SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```

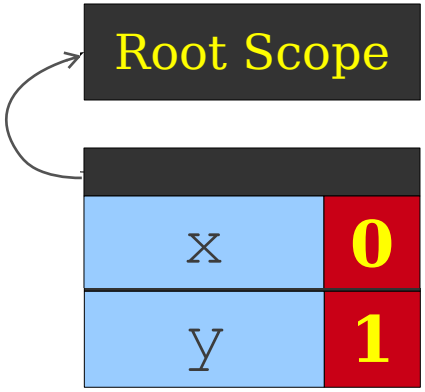


SYMBOL TABLES — A SECOND EXAMPLE

```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```



SYMBOL TABLES — A SECOND EXAMPLE




```
0:    int x;
1:    int y;
2:    int MyFunction(int x, int y)
3:    {
4:       int w, z;
5:          {
6:             int y;
7:          }
8:          {
9:             int w;
10:         }
11:   }
```

the structure does not mention MyFunction, which is incorrect!

SYMBOL TABLES = STACK OF SCOPES

logically, the symbol table is a **linked structure of scopes**

- each scope stores a pointer to its parents, but not vice-versa
- from any point in the program, the symbol table appears to be a **stack**
  - every point of the program (e.g every node of the syntax tree) has its own symbol table
  - the symbol table implementation may use pointers between nodes to avoid copying

WHAT OPERATION DO WE NEED?

given the above assumptions, we will need:

- add a new name into the symbol table with its attributes
  - in the type checking this operation will be `insert(Env,x,T) = Env[x ⟼ T]`
- lookup a name in the **current and enclosing scopes**
  - to check if it is multiply declared
  - to check for a use of an undeclared name, and
  - to link a use with the corresponding symbol-table entry
- do **what must be done** when a new scope is entered
  - in the type checking this operation will be `newScope(Env) = Env•[]`
- do **what must be done** when a scope is exited
  - in the type checking this operation will be `exitScope(Env•Env') = Env`

SYMBOL TABLE, FORMALLY = ENVIRONMENT

formally, a scope is a map from variables to something

in `SimpLan` , "something" is an object of `STentry`

- `STentry` contains the infos about the type
- and other infos

- a scope is denoted by the Greek letter Γ, and it is called **environment**
- the simplest symbol table consisting of one scope is therefore an environment
- also **stacks of scopes** will be called **environments**

FORMAL DEFINITION OF Γ

Definition: environment

the environment Γ is a **finite partial map** $\texttt{Id} \rightarrow \texttt{STentry}$

- that takes identifiers (variables and function symbols) and returns STentries
- Γ **may be extended**: for example we want to extend Γ with the binding $[\texttt{z} \mapsto \texttt{T}]$
  - $\Gamma[\texttt{z} \mapsto \texttt{T}]$ means the environment Γ' such that

$$\Gamma'(\texttt{u}) = \begin{cases} \texttt{T} & \texttt{if u = z} \\ \Gamma(\texttt{u}) & \texttt{otherwise} \end{cases}$$

Γ may be formalized by **explicitating the list of the bindings**

- **example**: let $\Gamma = [\texttt{x} \mapsto \texttt{T1}, \texttt{y} \mapsto \texttt{T2}]$ then
  - $\Gamma(\texttt{x}) = \texttt{T1}$ and $\Gamma(\texttt{y}) = \texttt{T2}$
- $\varnothing$ is the **empty map** (that may be also noted with **[ ]**)
- the notation $\Gamma[\texttt{x} \mapsto \texttt{T3}]$ also means **updating**:
  - $\Gamma[\texttt{x} \mapsto \texttt{T3}] = [\texttt{x} \mapsto \texttt{T3}, \texttt{y} \mapsto \texttt{T2}]$

STACK OF ENVIRONMENTS

stacks of environments are denoted $\Gamma \bullet \Gamma'$ (Γ' is the top-envitonment)

abuse of notation! Γ indicates a single environment and a sequence of environments!

stack of environments are called **environments** and are denoted with Γ

- **lookup**: $\Gamma \bullet \Gamma'(x) = \begin{cases} \Gamma'(x) & \text{if } x \in \text{dom}(\Gamma') \\ \Gamma(x) & \text{otherwise} \end{cases}$
- the **add** operation $(\Gamma \bullet \Gamma')[x \mapsto T]$ becomes $\Gamma \bullet (\Gamma'[x \mapsto T])$, i.e. add x in the top environment
- if Γ is a sequence of environments, $\text{top}(\Gamma)$ returns the top environment
- the **add-new-scope** operation is $\Gamma \bullet [\,]$

ASSUMPTIONS

from now on, assume that our language:

- uses **static scoping**
- requires that **all names be declared** before they are used
- **does not allow multiple declarations** of a name in the same scope
  - even for different kinds of names
- does allow **the same name to be declared in multiple nested scopes**
  - but only once per scope
- uses the same scope for function's/method's parameters and for the local variables declared at the beginning of the method

SYMBOL TABLE IMPLEMENTATION

assume that the symbol table will be used to answer two questions (additional simplification):

1. given a declaration of a name, **is there already a declaration of the same name in the current scope** ?
2. given a use of a name, **to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared**?

remark: point 2 is also relevant when you generate code because you need to store an offset in the activation record

- therefore, the symbol table, or part of it, must be kept till the end of the compilation

we keep "part of it"

TWO POSSIBLE SYMBOL TABLE IMPLEMENTATIONS

1. a **list of (hash)tables**
2. a **(hash)table of lists**

for each approach, we will consider

- what must be done when processing a declaration,
- when processing a use, and
- when entering and exiting a scope.

simplification: assume each symbol-table entry includes only

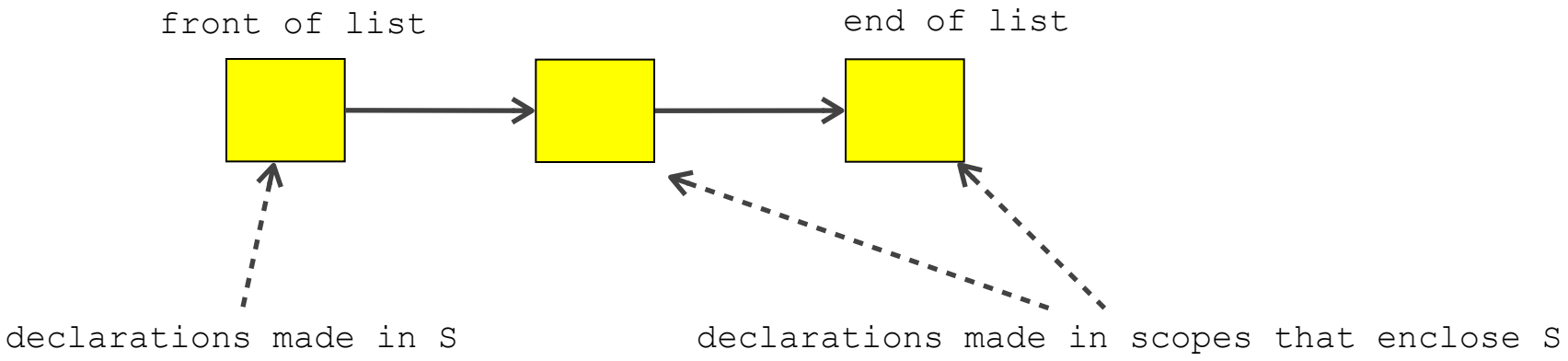
- the symbol name
- its type
- the nesting level of its declaration

IMPLEMENTATION 1: LIST OF HASHTABLES

the idea:

- the symbol table **is** a list of hashtables
- one hashtable **for each currently visible scope**

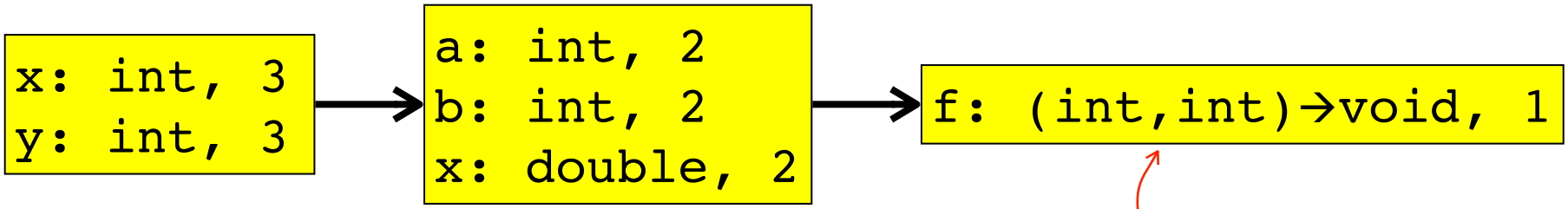
when processing a scope `S`:



EXAMPLE:

```
void f(int a, int b) {
    double x;
    while (...) { int x, y; ...  ★     }
}
void g() { f(4,5); }
```

at ★ the symbol table is :



method g has still not been parsed

LIST OF HASHTABLES: THE OPERATIONS

1. on scope entry:
   - increment the **current level number** and **add** a new empty hashtable to the front of the list
2. to process a declaration of `x`:
   - **lookup** `x` in the first table in the list
   - if it is there, then issue a "`multiply declared variable`" error
   - otherwise, add `x` to the first table in the list
3. to process a use of `x`:
   - lookup `x` starting in the first table in the list
   - if it is not there, then look up `x` in each successive table in the list
   - if it is not in any table then issue an "`undeclared variable`" error
4. on scope exit:
   - **remove** the first table from the list and **decrement** the current level number

REMEMBER

function/method names belong to the hashtable for the outermost scope

- not to the same table as the method's variables
- **for instance**, in the example above:
  - the function name `f` is in the symbol table for the outermost scope
  - name `f` is not in the same scope as parameters `a` and `b`, and variable `x`
  - therefore, when the use of name `f` in method `g` is processed, the name is found in an enclosing scope's table

THE COMPUTATIONAL COMPLEXITY OF OPERATIONS

1. scope entry
   a) time to initialize a new, empty hashtable
   b) probably proportional to the size of the hashtable
2. process a declaration
   a) using hashing, constant expected time (O(1))
3. process a use:
   a) using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined
4. scope exit
   a) time to remove a table from the list, which should be O(1)

EXERCISE

assume to have `Java?` (an **imaginary language** different from `Java`) that allows a function to have both a parameter and a local variable with the same name

- any use of the name in the body of the function **refers to the local variable**

consider the code

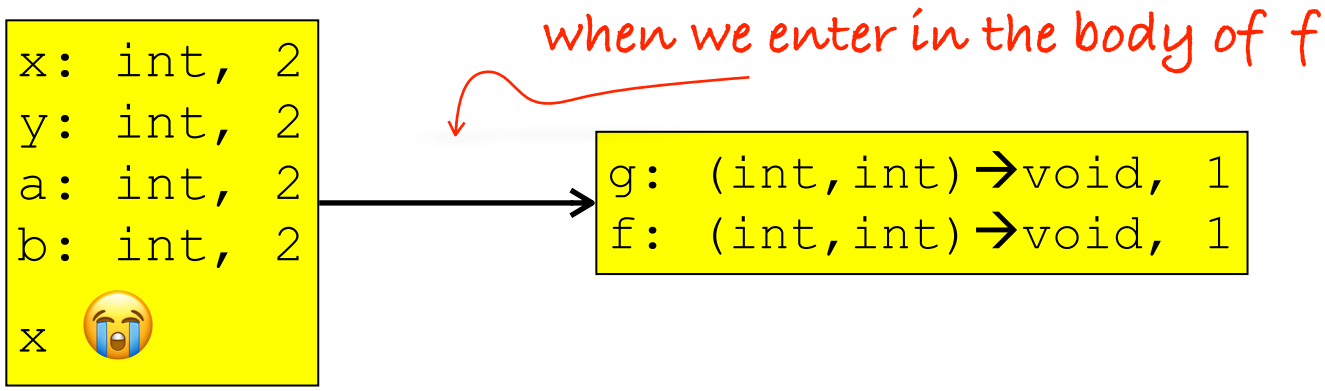
```
void g(int x, int a) { }
void f(int x, int y, int z) { int a, b, x; ... }
```

- draw the symbol table as it would be after processing the declarations in the body of `f` under:
  - the scoping rules we have been assuming
  - `Java?` scoping rules

EXERCISE

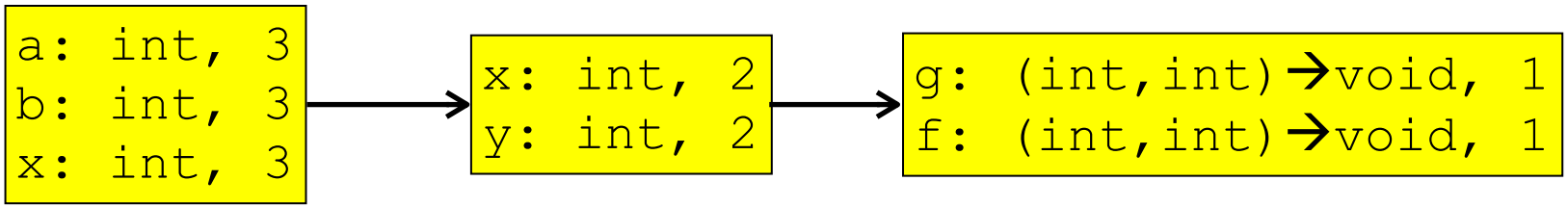
```
void g(int x, int a) { }
void f(int x, int y) { int a, b, x; ... }
```

- symbol table with **Java** scoping rules:



ERROR!

- **Java?** scoping rules

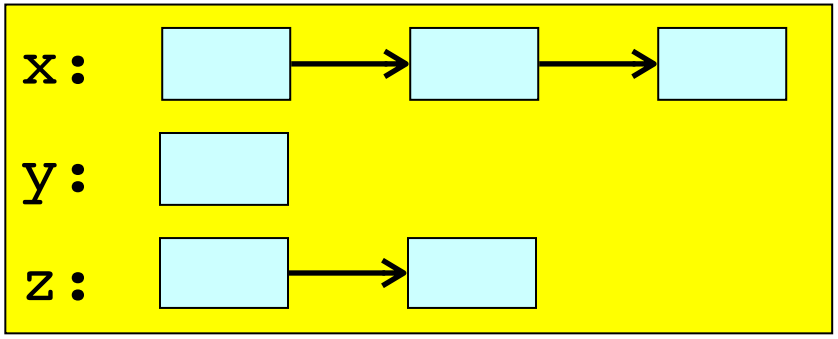


NO ERROR!

IMPLEMENTATION 2: HASHTABLE OF LISTS

the idea:

- when processing a scope `S`, the structure of the symbol table is

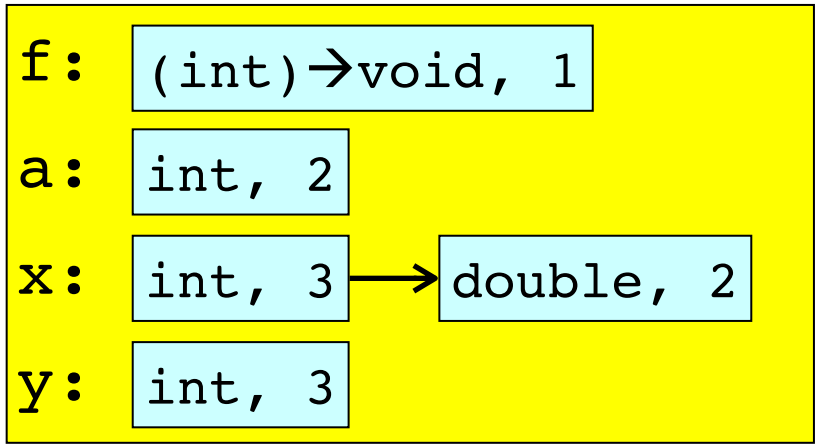


- there is just **one big hashtable**, containing an entry for each name for which there is
  - some declaration in scope `S` or
  - in a scope that encloses `S`
- each name has an associated list of symbol-table entries
  - the first item corresponds to the most closely enclosing declaration
  - the other list items correspond to declarations in enclosing scopes

EXAMPLE

```
void f(int a) {
        double x;
        while (...) { int x, y;★...        }
        void g() { f(); }
}
```

at ★ the symbol table is



the **nesting level information is crucial**

the level-number attribute stored in each list item enables us to determine whether the most closely enclosing declaration was made

- in the **current scope** or
- in an **enclosing scope**

HASHTABLE OF LISTS: THE OPERATIONS

1. on **scope entry**:
   - increment the **current level number**
2. to process a **declaration** of `x`:
   - look up `x` in the symbol table
   - if `x` is there, fetch the level number from the first list item
   - if that level number = the current level then **issue a** `"multiply declared variable"` error
   - otherwise, **add** a new item to the front of the list with the appropriate type and the current level number
3. to **process a use** of `x`:
   - look up `x` in the symbol table
   - if it is not there, then **issue an** `"undeclared variable"` error
4. on **scope exit**:
   - scan all entries in the symbol table, looking at the first item on each list
   - if that item's level number = the current level number, then **remove it from its list** (and if the list becomes empty, remove the entire symbol-table entry)
   - finally, decrement the **current level number**

THE COMPUTATIONAL COMPLEXITY OF OPERATIONS

1. scope entry: time to increment the level number, O(1)

2. process a declaration: using hashing, constant expected time (O(1))

3. process a use: using hashing, constant expected time (O(1))

4. scope exit: time proportional to the number of names in the symbol table

EXERCISE

assume that the symbol table is implemented using a **hashtable of lists**

draw pictures to show how the symbol table changes as each declaration in the following code is processed

```
void g(int x, int a) {
   double d;
   while (...) { int d, w;
                 double x, b;
                 if (...) { int a,b,c; }
   }
   while (...) { int x,y,z;
   }
}
```

SCOPING

the **scope rules of a language**:

- determine which declaration of an identifier corresponds to each occurrence of the identifier
- i.e., scoping rules map identifier occurrences to their declarations

`C++` and `Java` use **static scoping**:

- mapping from uses to declarations is made at compile time
- `C++` uses the "**most closely nested**" rule
  - an occurrence of `x` matches the declaration in the most closely enclosing scope such that the declaration precedes the use
  - a deeply nested variable `x` hides `x` declared in an outer scope
- in `Java`:
  - inner scopes cannot define variables defined in outer scopes

SCOPE LEVELS

each function in languages like `Java` and `C` has one or more scopes:

- **one** for the parameters **and** for the function body
- and possibly additional scopes in the function (**for each for loop** and **for each nested block** delimited by curly braces)

SCOPE LEVELS

example:

```
void f(int k) {
      int y = 0;
      int x = 3;
      while (y) {
          int x = 1;   // another local var x
      }                // (legal in C++, not legal in Java)
}
```

- the outermost scope includes just the name "`f`"
- function `f` itself has two (nested) scopes in `Java` (e in `C++`):
  1. the first one includes `k`, `y` and `x`
  2. the innermost scope is for the body of the while loop, and includes the variable `x` that is initialized to `1`

EXERCISE

this is a C++ program

- match each var-occurrence to its declaration, or say when an occurrence is undeclared

```
class Foo {
    int k=10, x=20;
    void foo(int k) {
       int a = x; int x = k; int b = x;
        while (...) {
                        int x=11;
                        if (x == k) {
                               int k, y;
                               k = (y = x);
                        }
                        if (x == k) { int x, y; }
        }
    }
}
```

SUMMING UP

in order to manage symbol tables you need 4 functions

1. `SymbolTable` **`newScope`**`(SymbolTable st)`
   `// extends the st with a new scope (called add in SimpLan)`

2. `SymbolTable` **`add`**`(SymbolTable st, String id, Type t)`
   `// if there is no clash of names, adds id ⟼ t to st`

3. `Type` **`lookup`**`(SymbolTable st, String id)`
   `// looks for the type of id in st, if any`

4. `SymbolTable` **`remove`**`(SymbolTable st)`
   `// exits the current scope`

it is crucial to take care of exceptions!

USED BEFORE DECLARED?

can a name be used before it is defined?

- `Java`: a method or field name can be used before the definition appears
- not true for a variable!

example:

```
class Test {
  void f() {
        val = 0;  // field val has not yet been declared -- OK
        g();      // method g has not yet been declared -- OK
        x = 1;    // var x has not yet been declared -- ERROR
        int x;
    }
    void g() {}
    int val;
}
```

SCOPING: EXAMPLE

`Java`: you may use a same name for

- a class,
- field of the class,
- a method of the class, and
- a local variable of the method

example: legal `Java` program:

```
class Test {
        int Test;
        Test( ) { double Test; }
}
```

SCOPING: OVERLOADING

`Java` and `C++` (but not in `Pascal` or `C`):

- can use the same name for more than one method as long as the number and/or types of parameters are unique

example:

```
int add(int a, int b);
float add(float a, float b);
```

SNIPPETS OF THE SIMPLAN COMPILER

the `ProgLetInNode.java` has the following method for defining symbol tables:

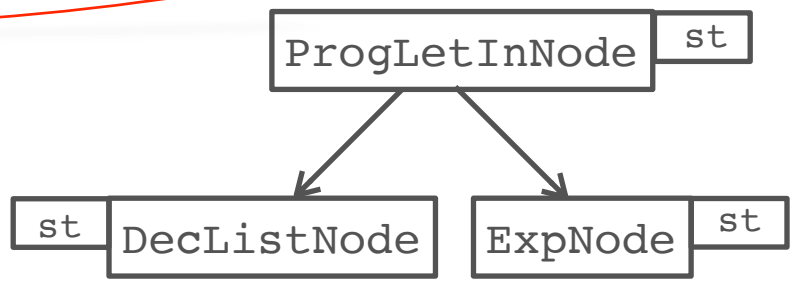
```
public ArrayList<SemanticError> checkSemantics(SymbolTable ST, int _nesting) {
        nesting = _nesting + 1 ;
        HashMap<String,STentry> H = new HashMap<String, STentry>();
        ST.add(H);

        //declare resulting list
        ArrayList<SemanticError> errors = new ArrayList<SemanticError>();

        for (Node d : dec) {
           errors.addAll(d.checkSemantics(ST, nesting)) ;
        }

        //check semantics in the exp body
        errors.addAll(exp.checkSemantics(ST, nesting)) ;

        //clean the scope, we are leaving a let scope
        ST.remove();

        //return the result
        return errors;
}
```

the nesting level is increased!

hashtable

list of hashtables

the environment is removed!



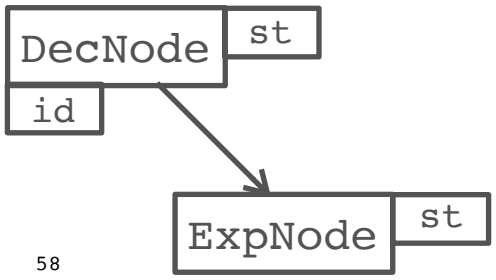
SNIPPETS OF THE SIMPLAN COMPILER

the **`DecNode.java`** has the following method for defining symbol tables:

```
public ArrayList<SemanticError> checkSemantics(SymbolTable ST, int _nesting) {
        ArrayList<SemanticError> errors = new ArrayList<SemanticError>();

        nesting = _nesting ;

       errors.addAll(exp.checkSemantics(ST, nesting));

       if (ST.top_lookup(id) == true)
           errors.add(new SemanticError("Var id " + id + " already declared"));
       else ST.insert(id, (Type) type, nesting, "") ;

       return errors ;
}
```

verify that id is not in the top-level hash table

buid an entry with right nesting-level and type ("" because there is no label ...)



IN FUNNODE.JAVA OF THE SIMPLAN COMPILER

```
public ArrayList<SemanticError> checkSemantics(SymbolTable ST, int _nesting) {
    ArrayList<SemanticError> errors = new ArrayList<SemanticError>();
    nesting = _nesting ;
    if (ST.lookup(id) != null)
         errors.add(new SemanticError("Identifier " + id + " already declared"));
    else {
         HashMap<String,STentry> HM = new HashMap<String,STentry>() ;
         ArrayList<Type> partypes = new ArrayList<Type>() ;

         ST.add(HM);

         for (ParNode arg : parlist){
              partypes.add(arg.getType());
              if (ST.top_lookup(arg.getId()))
                 errors.add(new SemanticError("Parameter id "+ arg.getId() +" already declared")
              else ST.insert(arg.getId(), arg.getType(), nesting+1, "") ;
         }

         type = new ArrowType(partypes, returntype) ;
         ST.increaseoffset() ; // offset aumentato per il return value: SEE CODE GENERATION
         for (Node dec : declist)
              errors.addAll(dec.checkSemantics(ST, nesting+1));

         errors.addAll(body.checkSemantics(ST, nesting+1));
         ST.remove();

         flabel = SimpLanlib.freshFunLabel() ;
         ST.insert(id, type, nesting, flabel) ;
    }
    return errors ;
}
```

adds an empty environment ahead

adds the formal parameters in the top environment: notice the nesting level!

buid an entry with right nesting-level and type (offset is used in the code generation)

generate the label for the function in the bytecode!

look for comments in the code!

SNIPPETS OF THE SIMPLAN COMPILER

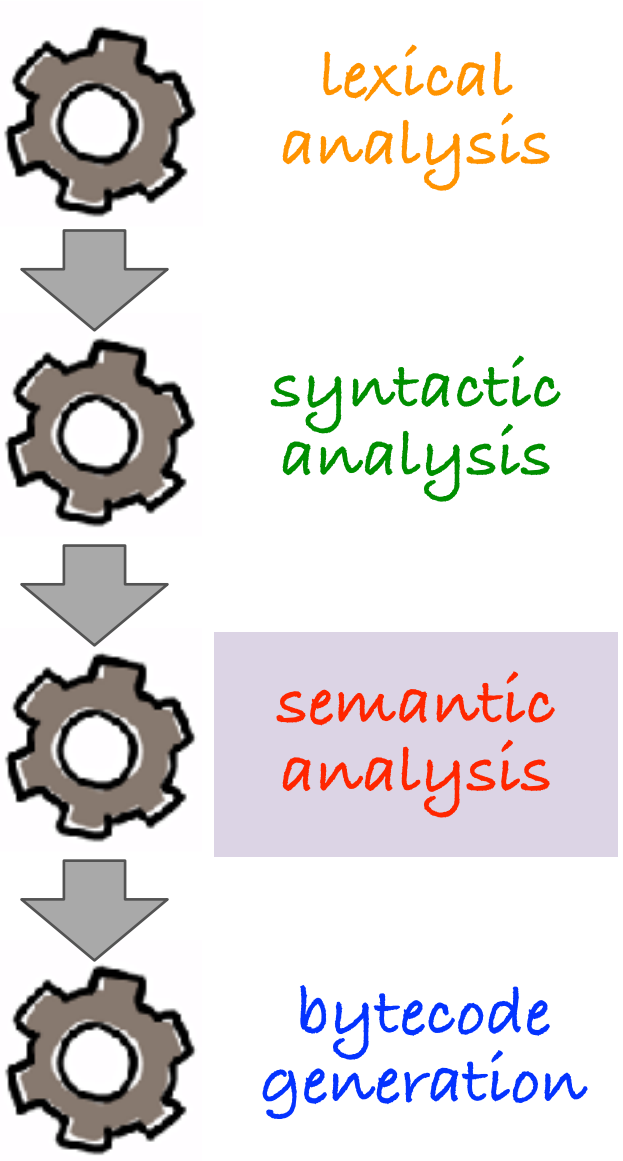
the **`IdNode.java`** has the following method for defining symbol tables:

```
public ArrayList<SemanticError> checkSemantics(SymbolTable ST, int _nesting) {
        ArrayList<SemanticError> errors = new ArrayList<SemanticError>();

        nesting = _nesting ;

        STentry st_type = ST.lookup(id) ;
        if (st_type == null)
             errors.add(new SemanticError("Id " + id + " not declared"));
        else type = st_type ;

        return errors;
    }
```

take the type of id (and other infos...)

error if there is no type

NEXT LECTURE



- scopes and symbol tables
- type checking

the `SimpLan` interpreter