The Symbol Table

Working with Identifiers

• We have seen a simple implementation of type checking for assignment statements, for example for "x = 123":

```
class AssignmentStatement extends Statement {
    Identifier leftSide;
    Expression rightSide;
}

Type-checking an assignment statement s:
    var leftType = getTypeOfExpression(s.leftSide);
    var rightType = getTypeOfExpression(s.rightSide);
    if(!leftType.equals(rightType))
        throw new TypeErrorException();
```

Before we discuss how to implement getTypeOfExpression we have to first solve another problem: How do we know whether the variable "x" has been declared and what its type is?

The Symbol Table (step 1)

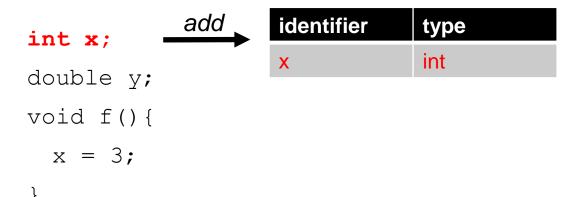
- The symbol table is a data structure (for example a hash map with the identifier as key) that stores information about all identifiers and their type
- Example in C:

int x;	
double y;	
void f(){	
x = 3;	
1	

identifier	type
-	-

At the beginning the symbol table is empty

The Symbol Table (step 2)



When we traverse the AST, every time we encounter a declaration of a variable or function, we add it to the symbol table

The Symbol Table (step 3)

```
int x;
double y; add
void f() {
    x = 3;
}

identifier type

x int
y double
```

The Symbol Table (step 4)

```
int x;
double y;
void f() {
    x = 3;
}
```

identifier	type
x	int
у	double
f	()→void

Using the symbol table

 During type checking, we can access the symbol table to retrieve the type of a variable or function an identifier is referring to:

```
identifier
                                                          type
                                                          int
HashMap<String, Type> symbolTable;
                                              Χ
                                                          double
                                              У
Type getTypeOfExpression(Expression e) {
                                                          ()→void
    if(e instanceof Identifier id) {
        if (symbolTable.contains(id.name))
            return symbolTable.get(id.name);
        else
          throw new UnknownIdentifierException();
    else if ...more complex expressions...
```

- Again, better use visitor pattern instead of instanceof
- If you prefer, pass symbolTable as argument to getTypeOfExpression

Problem with forward references

Here is another example in C:

```
int x;
double y;
void f() {
void g() {
```

identifier	type
X	int
У	double
f	()→void

Symbol table when analyzing the AST of function f

Problem: When traversing the AST of function f() we have not yet added g() to the symbol table

Handling forward references

Two solutions to the problem of forward references:

Solution 1: How C does it: prototypes

Solution 2: Do multiple passes over the AST

- First pass: collect the types of global variables, functions,...
 Result: complete symbol table
- Second pass: Traverse the AST for type checking etc.

Scopes

A more complex example:

```
int x;
double y;

The declaration of the
   parameter y hides
   ("shadows") the global
        variable y

void f(int y, int z) {
        Inside the function f the
        identifier y refers to the
        parameter y, not to the
        global variable y
```

- The part of the code in which a variable is visible under a specific name is called the *scope* of that name-variable binding
- In this example, we have different scopes:
 - the scope of the global variables x and y: the entire program
 - the scope of the parameters y and z: only inside f

Implementing multiple scopes

We can implement multiple scopes by linked symbol tables

```
identifier
                                             type
int x;
                                             int
                               X
                                                            Symbol table T_1 outside
double y;
                                                            function f
                                             double
                                             (int,int)→void
                                                                Link to previous table
void f(int y, int z){
                               identifier
                                             type
  x = y;
                                             int
                               У
                                                           Symbol table T_2 inside
                                                           function f
                                             int
                               Ζ
```

- Inside function f, we create a new symbol table that is linked to the first table
- \blacksquare To check an identifier inside f, we first look in T_2
- If we cannot find the identifier in T_2 , we look in T_1

Implementing multiple scopes (2)

Possible implementation:

```
class SymbolTable {
    SymbolTable previousTable;
    HashMap<String, Type> entries;
    SymbolTable(SymbolTable prev) { previousTable = prev; }
void checkTypes(Function f, SymbolTable globalTable) {
    // new symbol table linked to global table
    var localTable = new SymbolTable(globalTable);
    // add parameters of the function to local table
    localTable.add(f.parameters);
    // use local table when type-checking body of f
    checkTypes(f.body, localTable);
    // note that we didn't modify globalTable. Outside f, the parameters
    // of f are not visible
```

Nesting scopes (step 1)

Most languages allow to nest an unlimited number of scopes

```
int x;
void f(int y) {
   for(int i=0;i<10;i++) {
      int z = 10;
   }
   int t = 2;
}</pre>
```

identifier	type
X	int
f	(int)→void

Symbol table outside f

Nesting scopes (step 2)

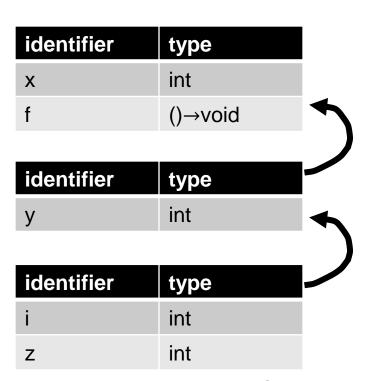
```
int x;
void f(int y) {
   for(int i=0;i<10;i++) {
      int z = 10;
   }
   int t = 2;
}</pre>
```

identifier	type	
x	int	
f	()→void	1
identifier	type	
у	int	

Symbol table inside f

Nesting scopes (step 3)

```
int x;
void f(int y) {
    for(int i=0;i<10;i++) {
        int z = 10;
    }
    int t = 2;
}</pre>
```



Symbol table inside for-loop

Nesting scopes (step 4)

```
int x;
void f(int y) {
    for(int i=0;i<10;i++) {
        int z = 10;
    }
    // we are here
    int t = 2;
}</pre>
```

identifier	type
X	int
f	()→void

identifier	type
у	int

Symbol table when leaving for-loop

Nesting scopes (step 5)

```
int x;
void f(int y) {
    for(int i=0;i<10;i++) {
        int z = 10;
    }
    int t = 2;
}</pre>
```

identifier	type
X	int
f	()→void

identifier	type
у	int

identifier	type
t	int

This is like starting a new block { int t=2; }

Nesting scopes (step 6)

```
int x;
void f(int y) {
    for(int i=0;i<10;i++) {
        int z = 10;
    }
    int t = 2;
}
// we are here</pre>
```

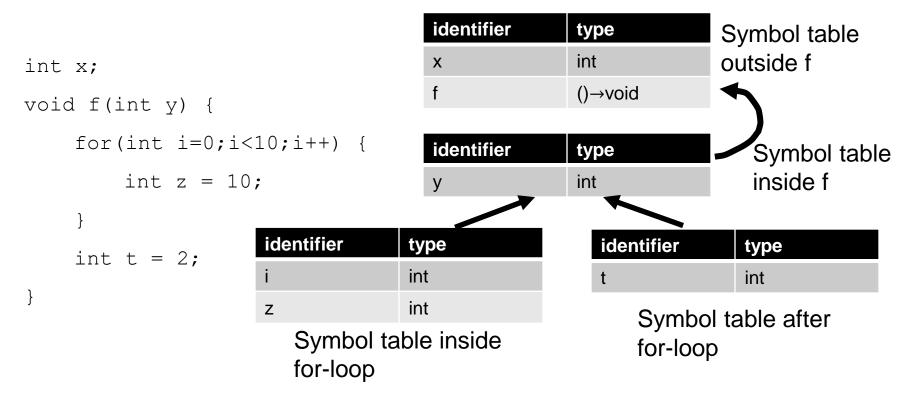
identifier	type
x	int
f	()→void

Symbol table after leaving function f

- You can understand the symbol table as a stack
 - When we enter a "{} block" (function, for-loop,...), we push a new local table onto the stack
 - When we leave a "{}" block, we pop the local table from the stack

Spaghetti stack

- Handling the symbol table like a stack has a draw back:
 - If the compiler has to do multiple traversals through the AST (for type checking, for optimization, etc.), we have to recreate the symbol table at each traversal
- In that case, it can be more efficient to create and keep the linked table. Resulting data structure is a spaghetti stack:



Dynamic scopes

- On the previous slides, our definition of scopes was purely lexical,
 i.e., they were defined by the static structure of the source code
 - Only during compile time. Not relevant during program execution.
- In some programming languages (e.g., Lisp, not Java or C), the scope depends on the dynamic behavior of the program

```
int x;
void g() {
    x = 10;
}
void f() {
    int x;
    g();
    // local variable x now has the value 10
}
```

- Programs with dynamic scope are more difficult to understand
- Less efficient: we have to consult the symbol table during execution of the program, not just during compilation

Conclusion

- As you can see, designing a programming languages involves many decisions about "how identifiers" work, for example:
 - What is the scope of a variable name?
 - Do we allow nested scopes?
 - Do we allow shadowing?
 - Do we allow forward referencing of function names?
 - Static vs dynamic scopes?
- In the examples in this course, we mostly follow Java's rules
- Actually, Java's scoping rules are even more complex than what we have seen here:
 - classes define scopes for their fields and methods
 - fields and methods of super-classes are visible
 - packages define scopes,...