1 Part VI: Name Analysis

- Programming languages are not really context-free.
- Representation of contexts in a compiler.
- Symbol tables and symbols.

2 Programming Languages are not really context-free

- Counter example: Every identifier needs to be declared.
- *being declared* is a property that depends on *context*.
- In theory, a programming language could be specified completely in a context-dependent grammar.
- But in practice, we define a context-free superset of the language in EBNF, and then we weed out illegal programs with further rules.
- Those rules typically need access to an identifier's declaration. (For example to know the declared type in type-checking)

3 Purpose of Name Analysis

- The purpose of name analysis is to find out, which use of an identifier refers to which definition.
- It also finds out if there are uses of an identifier which are not defined, and if there are illegal double definitions.

```
class A {
    int i,j;
    void m() {
        i = 7;
        int i;
        i = 3;
    }
    void n(int j, int k) {
        if (j == 5) {
            int i;
            i = 4;
        } else {
            i = 6;
        }
    }
}
```

4 Block Structured Languages

- Most programming languages have *block structured* visibility rules for identifiers.
- This also holds for Java.
- For the purpose of this discussion, a *block* is
  - anything enclosed in braces {},
  - the area consisting of a functions parameter-list to the end of its body.

```
class A {
    int i,j;
    void m() {
        i = 7;
        int i;
        i = 3;
    }
    void n(int j, int k) {
        if (j == 5) {
            int i;
            i = 4;
        } else {
            i = 6;
        }
    }
}
```

```
x = 3 * 4;
y = x + x;
z = x * y;
```

5 Scope

- Every defined identifier has its *scope*, i.e. an area of program text, in which it can be referred to.
- The scope of an identifier typically extends from the point of its definition to the end of the enclosing block.
- In Java methods and fields can be accessed before their definition, but we do not consider this here.
- It is illegal to refer to an identifier outside its scope.
- It is illegal to declare two identifiers with the same name in the same block.
- It is legal to declare an identifier in a nested block, which is also declared in an enclosing block.
- In this case the inner declaration hides (shadows) the outer.
- As a special rule, Java forbids shadowing parameters and local variables, but we do not consider this here.

6 Representation of Context in a Compiler

- We represent context by a global data structure which stores for every visible identifier data about its declaration.
- The data structure is called a *symbol table* and the information is called a *symbol table entry*.
- If a language has nested blocks, the symbol table should be structured in the same way.

7 Symbol Table Entries

- A *symbol table entry* is a data structure, which contains all the information about a defined identifier a compiler needs to know.
- We use a class Symbol for symbol table entries.
- Symbols have a name field, and a pos field, to indicate the point of definition.
- Because we usually want to store additional information we will build subclasses of Symbol.
- Additional information are for example the declared types or the number of local variables for a function.
- Because we define different things (variables, functions, classes), we will have more than one subclass.

8 class Symbol

```
class Symbol {
        /** position of the symbols definition
          */
        int pos;
        /** the name of the symbol
          */
        String name;
        /** create a new symbol
          */
        public Symbol(int pos, String name) {
                this.pos = pos;
                this.name = name;
        }
}
```

9 Symbols

- We have for every occurence of an identifier a field `sym` in the abstract syntax tree, which is set in the name analysis.
- The purpose of name analysis is to determine for every occurrence (usage or definition) of an identifier in the source code the corresponding symbol.
- At the definition point of an identifier, we construct a new symbol for it set the field `sym`, and enter it into the symbol table.
- At a usage point of an identifier we look it up in the symbol table and store it into the field `sym`.

10 Scopes

- Scopes represent areas of visibility.
- Symbols are grouped together in Scopes.
- For each block in the source program we have one Scope.
- We put the symbols for identifiers that are declared in a block into the corresponding scope.
- So, a Scope is a data structure which refers to all identifiers declared in it.
- Scopes are nested (as are blocks); therefore it is convenient to keep a field outer in a scope, which refers to the enclosing scope.
- This leads to the following class:

```
class Scope {
    /** map from identifier to symbol entries
     */
    public Map map;
    /** outer scope
     */
    public Scope outer;
    /** construct a new scope
     */
    public Scope(Scope outer) {
        this.outer = outer;
        this.map = new HashMap();
    }
```

```
/** enter a symbol in scope */
public void enter(Symbol sym) {
    if (map.containsKey(sym.name)) {
        error(); // double definition
    } else {
        map.put(sym.name, sym);
    }
}
```

```
        /** lookup a symbol */
        public Symbol lookup(String name) {
            if (map.containsKey(name)) {
                return (Symbol)map.get(name);
            } else if (outer != null) {
                return outer.lookup(name);
            } else {
                return null;
            }
        }
    }
```

11 Example (1)

$x = 3 * 4;$
$y = x + x;$
$z = x * y;$

12 Example (2)

```
class A {
    int i,j;
    void m() {
        i = 7;
        int i;
        i = 3;
    }
    void n(int j, int k) {
        if (j == 5) {
            int i;
            i = 4;
        } else {
            i = 6;
        }
    }
}
```

13 A Visitor for Name Analysis

For a definition:

- It has to construct the Symbols.
- It has to set the sym-field of the definition in the tree to the new Symbol.
- It has to enter the new Symbol into the Scope.
- It has to give an error for a double definition.

For a use:

- It has to lookup the Symbol for the identifier in the Scope.
- It has to set the sym-field in the tree to the looked up Symbol.
- It has to give an error for an undefined identifier.

```
public class Analyzer implements Tree.Visitor {

        /** current scope
         */
        Scope scope;

        /** construct a new semantic analyzer
         */
        public Analyzer() {
                this.scope = null;
        }

        /** the main name analysis method
         */
        public static void analyzeTree(Tree tree) {
                tree.apply(new Analyzer());
        }
```

```
/** analysis method for recursion
  * we use the visitor reuse version
  */
protected void analyze(Tree tree) {
        tree.apply(this);
}

/** open a new nested scope
  */
protected void openScope() {
        scope = new Scope(scope);
}

/** close a nested scope
  */
protected void closeScope() {
        scope = scope.outer;
}
```

14 Variable Declaration

StatOrDecl = VARDECL Type String
| ...

```
public void caseVarDecl(VarDecl tree) {
    analyze(tree.type);
    tree.sym = new VarSymbol(tree.pos, tree.name,
                             tree.type);
    scope.enter(tree.sym);
}
```

15 Variable Usage

```
Expression = IDENT String
           | ...

    public void caseIdent(Ident tree) {
        tree.sym = scope.lookup(tree.name);
        if (sym == null) {
            error();
        }
    }
```

16 Literals

Expression = STRINGLIT String
| ...

```
public void caseStringLit(StringLit tree) {
}
```

17 Blocks

StatOrDecl = BLOCK { StatOrDecl }
| ...

```
public void caseBlock(Block tree) {
    openScope();
    for (int i = 0; i < tree.stats.length; i ++) {
        analyze(tree.stats[i]);
    }
    closeScope();
}
```

18 Function Declarations

StatOrDecl = FUNDECL Type String { Parameter } StatOrDecl
| ...

```
public void caseFunDecl(FunDecl tree) {
        analyze(tree.type);
        tree.sym = new FunSymbol(tree.pos, tree.name,
                                 tree.type);
        scope.enter(tree.sym);
        openScope();
        for (int i = 0; i < tree.params.length; i ++) {
                analyze(tree.params[i]); // puts parameters in Scope
        }
        analyze(tree.body);
        closeScope();
}
```

19 Hash tables

- A hash table is a fast implementation for tables.
- A table here is set of pairs (key, value), with no double keys
- We have two operations on tables:
  - enter a pair (key, value): `put(key, value)`
  - find the corresponding value for a given key: `get(key)`
- Idea: Use a function `hash(key)` which maps each key to an integer, then store values in an array under the computed index.
- An example of a hash-function on strings would be the sum of all characters.

20 Hash tables (2)

- But: `hash` might yield the same integer for different keys!
- We use an array of linked lists.
- To enter a pair, we compute the integer `i` and enter the pair into the corresponding linked list `a[i]`.
- To lookup a key, we compute the integer `i` and look up the key in the corresponding linked list `a[i]`.
- If the table is big enough, the lists are typically very short (often 0 or 1 element).
- Then access is very fast.
- Choosing a good hash-function is essential for performance (taking the first character doesn't work well).
- In Java `HashMap` implements hashtables.