1 Part VI: Name Analysis

- Programming languages are not context-free
- Context rules for Jex
- Representation of Contexts in a Compiler
- Skeleton Specification of Visibility rules
- Optimisation

2 Programming Languages are not context-free

- Counter example: Every identifier needs to be declared
- *being declared* is a property that depends on *context*
- In theory, the syntax of 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.

3 Context Rules for Jex

- Jex has standard *block structured* visibility rules for identifiers
- 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.
  - the whole program is a block

4 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 extends from the point of its definition to the end of the enclosing block.
- 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.
- However, 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 the outer.

5 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* (or entry)
- Since Jex has nested blocks, the symbol table should be structured in the same way.
- The symbol table can be represented as a stack of blocks, with the current innermost block on top.

6 Symbols

- A *symbol* is a data structure, which contains all the information about a defined identifier a compiler needs to know.
- 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), we will have more than one subclass.

7 Symbols (2)

- We have for every occurence of an identifier a field `sym` in the abstract syntax tree, which gets 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`.

8 Symbols (3)

- It is sometimes necessary to step through all symbols of a scope in the sequence they were defined.
- Therefore they are linked linearly with a `next` field.

```
class Symbol {
    int pos;
    String name;
    Symbol next;
    public Symbol(int pos, String name) { ... }
}
```

9 Scopes

- Symbols are grouped together in Scopes.
- Scopes represent areas of visibility.
- A Scope is a data structure which refers to all identifiers declared in it.
- Scopes are nested; therefore it is convenient to keep a field outer in a scope, which refers to the enclosing scope.
- This leads to the following class fragment:

```
class Scope {
/** a list of all symbols in this scope
  */
        Symbol first;
/** the enclosing scope
  */
        Scope outer;
```



```
/** create new scope
  */
        public Scope(Scope outer) {
                this.outer = outer;
        }
/** lookup a name in the current or an enclosing scope;
  *  returns null, if no matching symbol is found
  */
        public Symbol lookup(String name) { ... }
/** enter a symbol into this scope
  */
        public void enter(Symbol sym) { ... }
}
```

- Scopes refer to the first symbol declared in the scope (first),
- Other symbols are accessed via the next field in class Symbol

Exercise: Write implementations for lookup and enter.

10 Symbols in Jex

- In Jex unqualified names (names that do not follow a dot), can refer to different kind of things:
  - local variables
  - global variables
  - functions
  - java classes
- It is the purpose of the name analysis to find for every unqualified name the point of definition and set the sym - field in the tree correspondingly
- At the definition point we construct a JexSymbol object and put it into the symbol table.
- At a usage point we look it up in the symbol table.

11 Symbols in Jex(2)

We implement this by a class JexSymbol and 4 subclasses

- LocalSym GlobalSym FunSym ClassSym

We use the class JexSymbol to implement functionality that is common to some symbols, but is Jex specific an not in class Symbol.

- Example: **boolean** isVariable(), which asks whether the symbol is a LocalSym or GlobalSym.

The information we need to store in the symbols are

- The types of the defined names
- Run-time information
  - the number of local variables

12 Types

Types in Jex are the classes or primitive types of Java. So we can use the class Class from Java to store the type of a defined entity.

- LocalSym, GlobalSym and ClassSym have a field Class cls.
- FunSym has a field Class resClass and a field Class[ ] argClasses.

```
class JexSymbol extends Symbol {
    static public class FunSym extends JexSymbol {
        Tree.FunDef tree;
        int localCount;
        Class resClass;
        Class[ ] argClasses;

    }
```



```
    static public class LocalSym extends JexSymbol {
        int adr;
        Class cls;
    }
    static public class GlobalSym extends JexSymbol {
        JexValue val;
        Class cls;
    }
    static public class ClassSym extends JexSymbol {
        Class cls;
    }
}
```


13 How it hangs together

```
import java.lang.*;
import java.io.*;
String z;
z = "hello";
int f(string s, int y) {
    int x;
    x = 0;
    while (x < y) {
        int y;
        y = x + x;
        System.out.println(z);
        System.out.println(s);
        System.out.println(y);
        x = x + 1;
    }
}
```


14 Imports

- If a name does not refer to a defined identifier, it might refer to a Java class.
- Java has a static method Class Class.classForName(String) which yields an object representing the class.
- But when looking for a class by name we also have to consider the imports.
- Imports are represented by a class Import

```
class Import {
    String prefix;
    String suffix;  // null for Star
    Import next;
    public Import(String prefix, String suffix) { ... }
    public Class classForName(String s) { ... }
}
```

- The next field again makes a linear list.

15 ImportScope

- We have a class ImportScope:

```
class ImportScope {
        Import first;
        public ImportScope() {
                this.first = new Import("", null); // import *;
        }
        public void enter(Import imp) { ... }
        public Class lookup(String s) { ... }
}
```

- enter puts a new import directive into the import scope.
- lookup tries to find a class with name s, using all the import directives in the import scope.

16 Optimisation

- The current scheme uses a linear search for identifiers
- In a production compiler this is far to slow
- Better schemes:
  - additionally link entries as a binary tree and use that for searching
  - Use a hash table for each block
  - Use a global hash table (fastest)

17 Hash tables

- A hash table is a fast implementation for tables.
- A table here is set of pairs (key, value).
- 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.

18 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 there exist classes Hashtable and HashMap, which implement hashtables.