Reading Assignment

Get and read Chapter 8 of Crafting a Compiler featuring Java.

(Available from DoIt Tech Store)

Symbol Tables in CSX

CSX is designed to make symbol tables easy to create and use.

There are three places where a new scope is opened:

- In the class that represents the program text. The scope is opened as soon as we begin processing the **`classNode`** (that roots the entire program). The scope stays open until the entire class (the whole program) is processed.
- When a **`methodDeclNode`** is processed. The name of the method is entered in the top-level (global) symbol table. Declarations of parameters and locals are placed in the method's symbol table. A method's symbol table is closed after all the

statements in its body are type checked.

- When a **`blockNode`** is processed. Locals are placed in the block's symbol table. A block's symbol table is closed after all the statements in its body are type checked.

CSX Allows no Forward References

This means we can do type-checking in *one pass* over the AST. As declarations are processed, their identifiers are added to the current (innermost) symbol table. When a use of an identifier occurs, we do an ordinary block-structured lookup, always using the innermost declaration found. Hence in

```
int i = j;
int j = i;
```

the first declaration initializes **`i`** to the nearest non-local definition of **`j`**.

The second declaration initializes **`j`** to the current (local) definition of **`i`**.

Forward References Require Two Passes

If forward references are allowed, we can process declarations in two passes.

First we walk the AST to establish symbol tables entries for all local declarations. No uses (lookups) are handled in this passes.

On a second complete pass, all uses are processed, using the symbol table entries built on the first pass.

Forward references make type checking a bit trickier, as we may reference a declaration not yet fully processed.

In Java, forward references to fields within a class are allowed.

Thus in


```
class Duh {
 int i = j;
 int j = i;
}
```

a Java compiler must recognize that the initialization of **i** is to the **j** field and that the **j** declaration is incomplete (Java forbids uninitialized fields or variables).

Forward references do allow methods to be mutually recursive. That is, we can let method **a** call **b**, while **b** calls **a**.

In CSX this is impossible!

(Why?)

Incomplete Declarations

Some languages, like C++, allow incomplete declarations.

First, part of a declaration (usually the header of a procedure or method) is presented.

Later, the declaration is completed.

For example (in C++):

```
class C {
  int i;
 public:
  int f();
};
int C::f(){return i+1;}
```


Incomplete declarations solve potential forward reference problems, as you can declare method headers first, and bodies that use the headers later.

Headers support abstraction and separate compilation too.

In C and C++, it is common to use a **#include** statement to add the headers (but not bodies) of external or library routines you wish to use.

C++ also allows you to declare a class by giving its fields and method headers first, with the bodies of the methods declared later. This is good for users of the class, who don't always want to see implementation details.

Classes, Structs and Records

The fields and methods declared within a class, struct or record are stored within a individual symbol table allocated for its declarations.

Member names must be unique within the class, record or struct, but may clash with other visible declarations. This is allowed because member names are qualified by the object they occur in.

Hence the reference **x.a** means look up **x**, using normal scoping rules. Object **x** should have a type that includes local fields. The type of **x** will include a pointer to the symbol table containing the field declarations. Field **a** is looked up in that symbol table.

Chains of field references are no problem.

For example, in Java

System.out.println

is commonly used.

System is looked up and found to be a class in one of the standard Java packages (**java.lang**). Class **System** has a static member **out** (of type **PrintStream**) and **PrintStream** has a member **println**.

Internal and External Field Access

Within a class, members may be accessed without qualification. Thus in

```
class C {
  static int i;
  void subr() {
     int j = i;
  }
}
```

field **i** is accessed like an ordinary non-local variable.

To implement this, we can treat member declarations like an ordinary scope in a block-structured symbol table.

When the class definition ends, its symbol table is popped and members are referenced through the symbol table entry for the class name.

This means a simple reference to **i** will no longer work, but **C.i** will be valid.

In languages like C++ that allow incomplete declarations, symbol table references need extra care. In

```
class C {
  int i;
 public:
  int f();
};
int C::f(){return i+1;}
```

when the definition of **`f()`** is completed, we must restore **`C`**'s field definitions as a containing scope so that the reference to **`i`** in **`i+1`** is properly compiled.

Public and Private Access

C++ and Java (and most other object-oriented languages) allow members of a class to be marked **`public`** or **`private`**.

Within a class the distinction is ignored; all members may be accessed.

Outside of the class, when a qualified access like **`C.i`** is required, only **`public`** members can be accessed.

This means lookup of class members is a two-step process. First the member name is looked up in the symbol table of the class. Then, the **`public`**/**`private`** qualifier is checked. Access to **`private`** members from outside the class generates an error message.

C++ and Java also provide a **`protected`** qualifier that allows access from subclasses of the class containing the member definition.

When a subclass is defined, it "inherits" the member definitions of its ancestor classes. Local definitions may hide inherited definitions. Moreover, inherited member definitions must be **`public`** or **`protected`**; **`private`** definitions may not be directly accessed (though they are still inherited and may be indirectly accessed through other inherited definitions).

Java also allows "blank" access qualifiers which allow **`public`** access by all classes within a package (a collection of classes).

Packages and Imports

Java allows packages which group class and interface definitions into named units.

A package requires a symbol table to access members. Thus a reference

```
java.util.Vector
```

locates the package **`java.util`** (typically using a **`CLASSPATH`**) and looks up **`Vector`** within it.

Java supports **`import`** statements that modify symbol table lookup rules.

A single class import, like

```
import java.util.Vector;
```

brings the name **`Vector`** into the current symbol table (unless a definition of **Vector** is already present).

An "import on demand" like

import java.util.*;

will lookup identifiers in the named packages after explicit user declarations have been checked.

Classfiles and Object Files

Class files (".class" files, produced by Java compilers) and object files (".o" files, produced by C and C++ compilers) contain internal symbol tables.

When a field or method of a Java class is accessed, the JVM uses the classfile's internal symbol table to access the symbol's value and verify that type rules are respected.

When a C or C++ object file is linked, the object file's internal symbol table is used to determine what external names are referenced, and what internally defined names will be exported.

C, C++ and Java all allow users to request that a more complete symbol table be generated for debugging purposes. This makes internal names (like local variable) visible so that a debugger can display source level information while debugging.

Overloading

A number of programming languages, including Java and C++, allow method and subprogram names to be *overloaded*.

This means several methods or subprograms may share the same name, as long as they differ in the number or types of parameters they accept. For example,

```
class C {
  int x;
  public static int sum(int v1,
                        int v2) {
     return v1 + v2;
  }
  public int sum(int v3) {
     return x + v3;
  }
}
```

For overloaded identifiers the symbol table must return a *list* of valid definitions of the identifier. Semantic analysis (type checking) then decides which definition to use.

In the above example, while checking

```
(new C()).sum(10);
```

both definitions of **`sum`** are returned when it is looked up. Since one argument is provided, the definition that uses one parameter is selected and checked.

A few languages (like Ada) allow overloading to be disambiguated on the basis of a method's result type. Algorithms that do this analysis are known, but are fairly complex.

Overloaded Operators

A few languages, like C++, allow operators to be overloaded.

This means users may add new definitions for existing operators, though they may not create new operators or alter existing precedence and associativity rules.

(Such changes would force changes to the scanner or parser.)

For example,

```
class complex{
  float re, im;
  complex operator+(complex d){
    complex ans;
    ans.re = d.re+re;
    ans.im = d.im+im;
    return ans;
} }
complex c,d; c=c+d;
```


During type checking of an operator, all visible definitions of the operator (including predefined definitions) are gathered and examined.

Only one definition should successfully pass type checks.

Thus in the above example, there may be many definitions of **+**, but only one is defined to take **`complex`** operands.

Contextual Resolution

Overloading allows multiple definitions of the same kind of object (method, procedure or operator) to co-exist.

Programming languages also sometimes allow reuse of the same name in defining different kinds of objects. Resolution is by context of use.

For example, in Java, a class name may be used for both the class and its constructor. Hence we see

```
C cvar = new C(10);
```

In Pascal, the name of a function is also used for its return value.

Java allows rather extensive reuse of an identifier, with the same identifier potentially denoting a class (type), a class constructor, a package name, a method and a field.

For example,

```
class C {
   double v;
  C(double f) {v=f;}
 }
 class D {
   int C;
   double C() {return 1.0;}
  C cval = new C(C+C());
}
```

At type-checking time we examine all potential definitions and use that definition that is consistent with the context of use. Hence new `C()` must be a constructor, `+C()` must be a function call, etc.

Allowing multiple definitions to co-exist certainly makes type checking more complicated than in other languages.

Whether such reuse benefits programmers is unclear; it certainly violates Java's "keep it simple" philosophy.