Laboratorio di Compilatori

Michele Peroli

REGIS

REsearch Group in Information Security
Department of Computer Science
University of Verona, Italy

23 MAGGIO 2013

Declarations define *names* for objects (variables, functions, types).

This is called *binding*.

The *scope* of a declaration is the portion of the program where the name is visible.

It may happen that the same name is declared in several *nested* scopes. This is the case for languages with nested *blocks*.

```
{int x;  int y;
         { int w; bool y; int z;
         ... w... ; ... x... ; ... z... ;
          }
... w... ; ... x... ; ... y...;
}
```

Nested Blocks

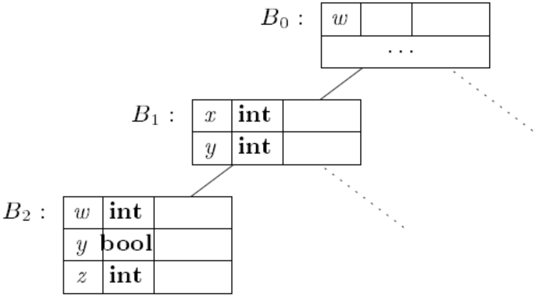

Figure 2.36: Chained symbol tables for Example 2.15

Scoping based on the structure of the syntax tree is called *static* or *lexical* binding. The *most-closely nested* rule for blocks: x is in the scope of the most-closely nested declaration of x.

Compilers use symbol tables (or environments) to keep track of names and the objects these are bound to.
For languages with blocks we shall implement scopes by setting up a separate symbol table for each scope.

We need a number of operations on symbol tables:

- empty : a symbol table in which no name is defined;
- bind : for linking a name to a piece of information;
- look up : to find a name and the information it is bound to;
- enter : to mark a new scope;
- exit ; to reestablish the environment to what it was before the scope was entered.

Symbol Tables

```
package symbols;                    // File Env.java
import java.util.*;
public class Env {
   private Hashtable table;
   protected Env prev;

   public Env(Env p) {
      table = new Hashtable(); prev = p;
   }

   public void put(String s, Symbol sym) {
      table.put(s, sym);
   }

   public Symbol get(String s) {
      for( Env e = this; e != null; e = e.prev ) {
         Symbol found = (Symbol)(e.table.get(s));
         if( found != null ) return found;
      }
      return null;
   }
}
```

Figure 2.37: Class *Env* implements chained symbol tables

The role of symbol tables is to pass information from declarations to uses.
When a declaration of x is analysed, a *semantic action* of a translation scheme puts information about x into the symbol table.
Later, a semantic action associated with a production

$$factor \rightarrow \mathbf{id}$$

gets the information from the symbol table.
A translation scheme illustrating a possible use of the class `Env` is given below.

Usando la classe `Env` implementare le operazioni:

```
empty
bind
lookup
enter
exit
```

mediante uno schema di traduzione per il linguaggio a blocchi definito dalla seguente grammatica.

Esercizio I: Linguaggio

```
program   -->     block

block    -->      '{'  decls stmts '}'

decls    -->    decls decl   |   eps

decl    -->       type id ;

stmts    -->    stmts stmt   |    eps

strnt     -->    block   |    factor ;

factor     -->    id
```

Scrivere, utilizzando LEX e YACC, un compilatore per il linguaggio `Simple` definito in Figura 1.
Il compilatore prende in ingresso un file contenete un programma `Simple` e genera il corrispondente programma in *C* (o Java).

Esercizio II (ctd.)

```
program  -->  preamble body
preamble  -->   DECLARATIONS declarations
body    -->   BEGIN_PROGRAM  cmd_seq  END_PROGRAM
declarations  -->  eps   | INTEGER id_seq IDENTIFIER
id_seq  -->  eps  |  id_seq  IDENTIFIER,
cmd_seq  -->  eps  | cmd_seq command;
command   -->  eps | SKIP | READ IDENTIFIER |
                  WRITE exp | IDENTIFIER := exp |
                  IF bool_exp THEN cmd_seq ELSE cmd_seq FI |
                  WHILE bool_exp DO cmd_seq OD
exp   -->   exp + term | exp - term | term
term   -->  term * factor | term / factor | factor
factor  -->  NUMBER | IDENT
bool_exp  -->  exp = exp | exp < exp | exp > exp
```

Figure : Il linguaggio Simple