a) A comment begins with // and includes all characters until the end of that line.

b) A comment begins with /* and includes all characters through the next occurrence of the character sequence */.

Exercise **2.6.2**: Extend the lexical analyzer in Section 2.6.5 to recognize the relational operators <, <=, ==, !=, >=, >.

Exercise **2.6.3**: Extend the lexical analyzer in Section 2.6.5 to recognize floating point numbers such as 2., 3.14, and .5.

2.7 Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

From Section 1.6.1, the scope of a declaration is the portion of a program to which the declaration applies. We shall implement scopes by setting up a separate symbol table for each scope. A program block with declarations[8] will have its own symbol table with an entry for each declaration in the block. This approach also works for other constructs that set up scopes; for example, a class would have its own table, with an entry for each field and method.

This section contains a symbol-table module suitable for use with the Java translator fragments in this chapter. The module will be used as is when we put together the translator in Appendix A. Meanwhile, for simplicity, the main example of this section is a stripped-down language with just the key constructs that touch symbol tables; namely, blocks, declarations, and factors. All of the other statement and expression constructs are omitted so we can focus on the symbol-table operations. A program consists of blocks with optional declarations and "statements" consisting of single identifiers. Each such statement represents a use of the identifier. Here is a sample program in this language:

```
{ int x; char y; { bool y; x; y; } x; y; }          (2.7)
```

The examples of block structure in Section 1.6.3 dealt with the definitions and uses of names; the input (2.7) consists solely of definitions and uses of names.

The task we shall perform is to print a revised program, in which the declarations have been removed and each "statement" has its identifier followed by a colon and its type.

[8] In C, for instance, program blocks are either functions or sections of functions that are separated by curly braces and that have one or more declarations within them.

Who Creates Symbol-Table Entries?

Symbol-table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer. In this chapter, we have the parser create entries. With its knowledge of the syntactic structure of a program, a parser is often in a better position than the lexical analyzer to distinguish among different declarations of an identifier.

In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say **id,** along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.

Example 2.14: On the above input (2.7), the goal is to produce:

```
{ { x:int; yrbool; } x:int; y:char; }
```

The first x and y are from the inner block of input (2.7). Since this use of x refers to the declaration of x in the outer block, it is followed by `int`, the type of that declaration. The use of y in the inner block refers to the declaration of y in that very block and therefore has boolean type. We also see the uses of x and y in the outer block, with their types, as given by declarations of the outer block: integer and character, respectively. •

2.7.1 Symbol Table Per Scope

The term "scope of identifier really refers to the scope of a particular declaration of *x*. The term *scope* by itself refers to a portion of a program that is the scope of one or more declarations.

Scopes are important, because the same identifier can be declared for different purposes in different parts of a program. Common names like i and x often have multiple uses. As another example, subclasses can redeclare a method name to override a method in a superclass.

If blocks can be nested, several declarations of the same identifier can appear within a single block. The following syntax results in nested blocks when *stmts* can generate a block:

block -> '{' decls stmts '}'

(We quote curly braces in the syntax to distinguish them from curly braces for semantic actions.) With the grammar in Fig. 2.38, *decls* generates an optional sequence of declarations and *stmts* generates an optional sequence of statements.

Optimization of Symbol Tables for Blocks

Implementations of symbol tables for blocks can take advantage of the most-closely nested rule. Nesting ensures that the chain of applicable symbol tables forms a stack. At the top of the stack is the table for the current block. Below it in the stack are the tables for the enclosing blocks. Thus, symbol tables can be allocated and deallocated in a stack-like fashion.

Some compilers maintain a single hash table of accessible entries; that is, of entries that are not hidden by a declaration in a nested block. Such a hash table supports essentially constant-time lookups, at the expense of inserting and deleting entries on block entry and exit. Upon exit from a block *B*, the compiler must undo any changes to the hash table due to declarations in block *B*. It can do so by using an auxiliary stack to keep track of changes to the hash table while block *B* is processed.

Moreover, a statement can be a block, so our language allows nested blocks, where an identifier can be redeclared.

The *most-closely nested* rule for blocks is that an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears.

Example 2.15: The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

```
1)  {   int xi; int y3;
2)      {   int w2; bool y2; int z2;
3)          ... w2 ...; ... xi ... y2 ... z2 ...;
4)      }
5)      ... w0 ...; ... xi ... yi ...;
6)  }
```

The subscript is not part of an identifier; it is in fact the line number of the declaration that applies to the identifier. Thus, all occurrences of x are within the scope of the declaration on line 1. The occurrence of y on line 3 is in the scope of the declaration of y on line 2 since y is redeclared within the inner block. The occurrence of y on line 5, however, is within the scope of the declaration of y on line 1.

The occurrence of w on line 5 is presumably within the scope of a declaration of w outside this program fragment; its subscript 0 denotes a declaration that is global or external to this block.

Finally, z is declared and used within the nested block, but cannot be used on line 5, since the nested declaration applies only to the nested block. •

The most-closely nested rule for blocks can be implemented by chaining symbol tables. That is, the table for a nested block points to the table for its enclosing block.

Example 2.16: Figure 2.36 shows symbol tables for the pseudocode in Example 2.15. *B* is for the block starting on line 1 and is for the block starting at line 2. At the top of the figure is an additional symbol table *Bo* for any global or default declarations provided by the language. During the time that we are analyzing lines 2 through 4, the environment is represented by a reference to the lowest symbol table — the one for *B%.* When we move to line 5, the symbol table for *B^* becomes inaccessible, and the environment refers instead to the symbol table for *B*, from which we can reach the global symbol table, but not the table for B_2. •

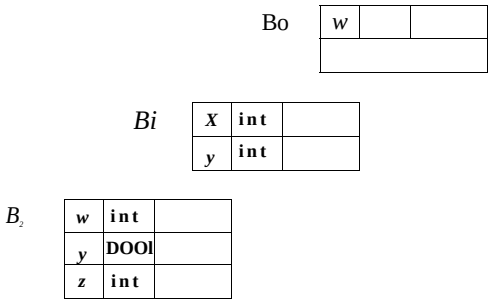

Figure 2.36: Chained symbol tables for Example 2.15

The Java implementation of chained symbol tables in Fig. 2.37 defines a class Env, short for *environment.*[9] Class Env supports three operations:

- *Create a new symbol table.* The constructor Env(p) on lines 6 through 8 of Fig. 2.37 creates an Env object with a hash table named table. The object is chained to the environment-valued parameter p by setting field next to p. Although it is the Env objects that form a chain, it is convenient to talk of the tables being chained.

- *Put* a new entry in the current table. The hash table holds key-value pairs, where:

  — The *key* is a string, or rather a reference to a string. We could alternatively use references to token objects for identifiers as keys.

  - The *value* is an entry of class Symbol. The code on lines 9 through 11 does not need to know the structure of an entry; that is, the code is independent of the fields and methods in class Symbol.

[9] "Environment" is another term for the collection of symbol tables that are relevant at a point in the program.

```
1) package symbols;                    // File Env.java
2) import java.util.*;
3) public class Env {
4)    private Hashtable table;
5)    protected Env prev;
6)    public Env(Env p) {
7)       table = new HashtableO; prev = p;
8)    >
9)    public void put(String s, Symbol sym) {
10)      table.put(s, sym);
11)   >
12)   public Symbol get(String s) {
13)      for( Env e = this; e != null; e = e.prev ) {
14)         Symbol found = (Symbol)(e.table.get(s));
15)         if( found != null ) return found;
16)      }
17)      return null;
18)   }
19) }
```

Figure 2.37: Class *Env* implements chained symbol tables

- *Get* an entry for an identifier by searching the chain of tables, starting with the table for the current block. The code for this operation on lines 12 through 18 returns either a symbol-table entry or null.

Chaining of symbol tables results in a tree structure, since more than one block can be nested inside an enclosing block. The dotted lines in Fig. 2.36 are a reminder that chained symbol tables can form a tree.

2.7.2 The Use of Symbol Tables

In effect, the role of a symbol table is to pass information from declarations to uses. A semantic action "puts" information about identifier *x* into the symbol table, when the declaration of *x* is analyzed. Subsequently, a semantic action associated with a production such as *factor* -> **id** "gets" information about the identifier from the symbol table. Since the translation of an expression *Ei* **op** E_2, for a typical operator **op,** depends only on the translations of *E*\ and E_2, and does not directly depend on the symbol table, we can add any number of operators without changing the basic flow of information from declarations to uses, through the symbol table.

Example 2.17: The translation scheme in Fig. 2.38 illustrates how class *Env* can be used. The translation scheme concentrates on scopes, declarations, and

uses. It implements the translation described in Example 2.14. As noted earlier, on input

```
program  ->•                          { top = null; }
           block

block    -»                           { saved = top;
                                        top = new Env(top);
                                        print("{ "); }
           decls stmts '}'            { top = saved;
                                        printO'} "); }

decls    —>  decls decl
         I   e

decl     -»•  type id ;               { s = new Symbol;
                                        s.type = type, lexeme
                                        top.put (id. lexeme, s); }

stmts    -»   stmts stmt

stmt     ->•  6/ocA;
          |   factor ;                 { print("; "); }

factor   ->   id                      { ˢ — top.get(id.lexeme);
                                        print (id. lexeme);
                                        print(":"); }
                                        print (s.type);
```

Figure 2.38: The use of symbol tables for translating a language with blocks

```
{ int x; char y; { bool y; x; y; } x; y; }
```

the translation scheme strips the declarations and produces

```
{ { x:int; y:bool; } x:int; y:char; }
```

Notice that the bodies of the productions have been aligned in Fig. 2.38 so that all the grammar symbols appear in one column, and all the actions in a second column. As a result, components of the body are often spread over several lines.

Now, consider the semantic actions. The translation scheme creates and discards symbol tables upon block entry and exit, respectively. Variable *top* denotes the top table, at the head of a chain of tables. The first production of

the underlying grammar is *program* ->• *block.* The semantic action before *block* initializes *top* to **null,** with no entries.

The second production, *block -» '{' decls stmts'}',* has actions upon block entry and exit. On block entry, before *decls,* a semantic action saves a reference to the current table using a local variable *saved.* Each use of this production has its own local variable *saved,* distinct from the local variable for any other use of this production. In a recursive-descent parser, *saved* would be local to the procedure for *block.* The treatment of local variables of a recursive function is discussed in Section 7.2. The code

top = **new** *Env (top);*

sets variable *top* to a newly created new table that is chained to the previous value of *top* just before block entry. Variable *top* is an object of class *Env;* the code for the constructor *Env* appears in Fig. 2.37.

On block exit, after '}', a semantic action restores *top* to its value saved on block entry. In effect, the tables form a stack; restoring *top* to its saved value pops the effect of the declarations in the block.[10] Thus, the declarations in the block are not visible outside the block.

A declaration, *decls* ->• **type id** results in a new entry for the declared identifier. We assume that tokens **type** and **id** each have an associated attribute, which is the type and lexeme, respectively, of the declared identifier. We shall not go into all the fields of a symbol object *s,* but we assume that there is a field *type* that gives the type of the symbol. We create a new symbol object *s* and assign its type properly by *s.type* — **type.***lexeme*. The complete entry is put into the top symbol table by *top.put(\d.lexeme, s).*

The semantic action in the production *factor* -> **id** uses the symbol table to get the entry for the identifier. The *get* operation searches for the first entry in the chain of tables, starting with *top.* The retrieved entry contains any information needed about the identifier, such as the type of the identifier. •

2.8 Intermediate Code Generation

The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program. In this section, we consider intermediate representations for expressions and statements, and give tutorial examples of how to produce such representations.

2.8.1 Two Kinds of Intermediate Representations

As was suggested in Section 2.1 and especially Fig. 2.4, the two most important intermediate representations are:

[10]Instead of explicitly saving and restoring tables, we could alternatively add static operations ***push*** and ***pop*** to class **Env.**