CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- **Recognizing identifiers and keywords (Ch 2.6.4)**
  - **Still in the Ch. 2.6 Lexical Analysis**
- **A lexical analyzer (Ch. 2.6.5)**
- **Incorporating a symbol table (Ch 2.7)**
- **Symbol Table per Scope (Ch 2.7.1)**
- **The Use of Symbol Tables (Ch 2.7.2)**
- **Intermediate Code Generation (Ch 2.8)**
- **Two kinds of Intermediate Representations (Ch 2.8.1)**
- **Construction of (Abstract) Syntax Trees (Ch 2.8.2)**
- **Static checking, Dynamic checking and Type checking**

Recognizing identifiers and keywords

- Why we need symbol tables?
  - To identify identifiers (id) and keywords
- For example, considering the C statement
  - sum = sum + x;
  - contains 6 tokens. The scanner (aka lexer; aka lexical analyzer) will convert the input into, id = id + id ;
  - Although there are three id tokens, the first and second represent the lexeme sum
  - However, the third represents an x !
  - Two different lexemes must be distinguished.

Recognizing identifiers and keywords

- Keywards, i.e. “then”, in C language, are syntactically the same as identifiers.
- The way C language does is the symbol table”.
- It is used to accomplishes both distinctions.
- We assume (as do most modern languages) that the keywords are *reserved*, i.e., cannot be used as program variables.
- Symbol table is used to contain all these reserved words and mark them as keywords
- Our previous example, x<y v.s x<=y

Recognizing identifiers and keywords

- When the < is read, the scanner needs to read “another character” to see if it is an “=“ ?
- But if that second character is y, the current token is < and the y must be pushed back onto the input stream so that the configuration is the same after scanning < as it is after scanning <=

A lexical analyzer

- A Java program is given in the book
- Scanner converts digits into num(s). We can shorten the grammar mentioned earlier.
- Here is the shortened version before the elimination of left recursion.
- Note that the value attribute of a num is its numerical value.
- Before vs. After

```
expr → expr + term { print('+') }
expr → expr - term { print('-') }
expr → term
term → 0           { print('0') }
. . .
term → 9           { print('9') }
```

```
expr    → expr + term     { print('+') }
expr    → expr - term     { print('-') }
expr    → term
term    → num             { print(num.value) }
```

A lexical analyzer

- We now think about the precedence for + and – computations
- In anticipation of other operators with higher precedence, we introduced "factor" and, parentheses for overriding the precedence. Our grammar would then become:

```
expr   → expr + term    { print('+') }
expr   → expr - term    { print('-') }
expr   → term
term   → factor
factor → ( expr ) | num { print(num,value) }
```

- The factor() procedure follows the familiar recursive descent pattern
- Note that we are now able to consider constants of more than one digit.

Incorporating a symbol table

- The table is primarily used to store and retrieve <lexeme, token> pairs.
  - Why retrieve? For checking purposes! Like we mentioned earlier

Symbol Table per Scope

- The idea for a language with nested scopes is that, when entering a block, a new symbol table is created.
  - Each such table points to the one immediately outer scope
  - This kind of structure supports the *most-closely nested* rule for symbols
    - A symbol, is in the scope of most-closely nested declaration
  - Interfacing
    - Create table: A new table is created and points to the immediately outer table
    - Insert entry (in the current table).
    - Retrieve entry (from the most-closely nested table in which it appears, <lexeme, Token> pairs

Symbol Table per Scope

- For Reserved keywords, the ways to process them:
  - We simply insert them into the symbol table prior to examining any input.
  - Then they (reserved words) can be found when used correctly.
  - Since their corresponding token will not be id (id is a reserved words), any use of them where an identifier is required can be flagged.

The Use of Symbol Tables

- Below is the example of grammar. The language consists just of nested blocks, a weird mixture of C- and ada-style declarations

```
program → block
block   → { decls stmts }     -- { } are terminals not actions
decls   →  decls decl | ε     -- study this one
decl    → type : id ;
stmts   → stmts stmt | ε      -- same idea, a list
stmt    → block | factor ;    -- enables nested blocks
factor  → id
```

- One possible program in this language is:
    - Everything is nested-blocks

```
{ int : x ;  float : y ;
  x ; y ;
  { float : x ;
    x ; y ;
  }
  { int : y ;
    x ; y;
  }
  x ; y ;
}
```

The Use of Symbol Tables

- To show that we have correctly parsed the input and obtained its meaning (i.e., performed semantic analysis), we present a translation scheme that digests the declarations and translates the statements so that the above example becomes
- { int; float; { float; float; } { int; int; } { int; float; } }

```
{ int : x ;  float : y ;
  x ; y ;
  { float : x ;
    x ; y ;
  }
  { int : y ;
    x ; y;
  }
  x ; y ;
}
```

The Use of Symbol Tables

- The use of symbol table translation, in book P.90
  - Looks kind of weird at this moment
  - To fully understand the details, you must read the book
  - 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. (see next page)
  - A new Env initializes a new symbol table;
  - top.put inserts into the symbol table of the current environment top;
  - top.get retrieves from that symbol table.

The Use of 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

Production	Action
Program → block	{top = null}
block → {	{ saved = top; top = new Env(top); print ("{ "); }
decls stmts }	{ top = saved; print ("} "); }
decls → decls decl | ε	
decl → type id ;	{ s = new Symbol; s.type = type.lexeme; top.put(id.lexeme,s); }
stmts → stmts stmt | ε	
stmt → block	
| factor ;	{ print("; "); }
factor → id	{ s = top.get(id.lexeme); print(s.type); }

Semantic Actions

The Use of Symbol Tables

- 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

Intermediate Code Generation

- The front end of a compiler constructs an intermediate representation (IR) of the source program
- However, the back end generates the target program

Two kinds of Intermediate Representations

- Trees, including parse trees and (abstract) syntax trees.
- Linear representations, especially “three-address code.“
- **Very** roughly speaking, (abstract) syntax trees are parse trees reduced to their essential components, and three address code looks like assembler without the concept of registers.
- Abstract-syntax trees, or simply syntax trees, were introduced in Section 2.5.1,
  - During parsing, syntax-tree nodes are created to represent significant programming constructs.
  - The choice of attributes depends on the translation to be performed.

Construction of Syntax Trees

- Consider the production
  - while-stmt → while (expr) stmt ;
  - The parse tree would have a node called while-stmt with 6 children: while, (, expr,), stmt, and ;
  - The essence of the while statement is that:
    - The system repeatedly executes stmt until expr is false.
    - Thus, the (abstract) syntax tree has a node (most likely called while) with two children, the syntax trees for expr and stmt.
  - To construct the while node, we execute the following pseudo-code
    - where x and y are the already constructed (synthesized attributes!) nodes for expr and stmt.

Construction of Syntax Trees

- **Syntax Trees for Statements**
  - The book has an SDD on page 94 for several statements.
  - The part for while reads:

  stmt → while (expr) stmt1   { stmt.n = new While(expr.n, stmt1.n); }
  - The n attribute gives the syntax tree node.
- **Representing Blocks in Syntax Trees**
  - Fairly easy
  - stmt → block   { stmt.n = block.n }
  - block → { stmts }   { block.n = stmts.n }

Construction of Syntax Trees

- These two just use the syntax tree for the statements constituting the block as the syntax tree for the block when it is used as a statement.

```
while ( x == 5 ) {
    blah
    blah
    more
  }
```

- Gives the while node of the abstract syntax tree two children:
  - The tree for x==5.
  - The tree for blah blah more.

Construction of Syntax Trees

- **Syntax trees for Expressions**
- For example, when parsing, we need to distinguish between + and * to insure that 3+4*5 is parsed correctly, reflecting the higher precedence of *
- Once parsed, the precedence is reflected in the tree itself (the node for + has the node for * as a child).
- For the rest of the "terms", the compiler treats + and * largely the same.
- So, it is common to use the same node label, say OP, for both of them.
- We "new" an OP (to wrap up term1 and factor)
  - term $\rightarrow$ $term_1$ * factor { term.n = new Op('*', $term_1$.n, factor.n); }

Construction of Syntax Trees

- Note, however, that the SDD (Figure 2.39, in page 94, Compilers 2nd Ed.) essentially constructs **both** the parse tree (expressions)and the syntax tree (statements).
- That latter is constructed as the attributes in the former.
  - Like the bottom of the previous slide!

Static checking, Dynamic checking and Type checking

- *Static checking* refers to checks performed during compilation; whereas, *dynamic checking* refers to those performed at run time
- Examples of static checks:
  - Syntactic checks, such as avoiding multiple declarations of the same identifier in the same scope. This check would not be enforced by the grammar.
  - Type checks.
- *Type checking* assure that the type of the operands are expected by the operator.
  - In addition to flagging errors (if there is any), this activity includes
    - *Coercions*. The automatic conversion of one type to another.
    - *Overloading*. In Java, Ada, and other languages, the same symbol can have different meanings depending on the types of the operands. **Static checks** are used to determine the correct operation, or signal an error if none exists

We are done with the Chapter 2! Wow!