Outline

- Recap
  - Introduction (Section 2.1)
  - Syntax definition (Section 2.2)
  - Syntax directed translation (Section 2.3)
  - Parsing (Section 2.4)
  - A translator for simple expressions (Section 2.5)
  - Lexical analysis (Section 2.6)
- A simple syntax-directed translator (Chapter 2)
  - The rest of Chapter 2
- Summary and homework
You should now be able to …

- Define language syntax using BNF grammar
- Parse sentences and detect syntax errors
- Use syntax-directed definition to perform language translation
- Understand lexical analysis
Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Intermediate code generation (Section 2.8)
  - Symbol tables (Section 2.7)
- Summary and homework
Two Kinds of Intermediate Representation

- Syntax tree representation
  - Expressions
    - $E_1 \text{ op } E_2$
  - Statements
    - $\text{while } (\text{expr}) \text{ stmt}$
    - $\text{do stmt while expr}$
  - Use a translation scheme
    - Semantic rules or semantic actions
- Three-address-code representation
A Translation Scheme for Constructing Syntax Tree

- Syntax tree for statements
- Syntax tree for expressions
Generating Syntax Tree: Syntax Tree for Statements

- **In Parser.java**

  Stmt stmts() throws IOException {
  if ( look.tag == '}') return Stmt.Null;
  else return new Seq(stmt(), stmts());
}

Expr x; Stmt s, s1, s2;
switch (look.tag) {
  case Tag.IF:
    match(Tag.IF); match('('); x = bool(); match(')'); s1 = stmt();
    if (look.tag != Tag.ELSE) return new If(x, s1);
    match(Tag.ELSE); s2 = stmt();
    return new Else(x, s1, s2);
...

Instructor: Dr. Liang Cheng  
CSE302: Compiler Design  
01/30/07
Generating Syntax Tree: Syntax Tree for Statements, Expressions

In Parser.java

Stmt stmt() throws IOException{
    Expr x; Stmt s, s1, s2;
    switch( look.tag ) {
        case Tag.WHILE:
            While whilenode = new While();
            match(Tag.WHILE); match('('); x = bool(); match(')'); s1 = stmt();
            whilenode.init(x, s1); return whilenode;
    ...

Expr expr() throws IOException {
    Expr x = term();
    while( look.tag == '+' || look.tag == '-' ) {
        Token tok = look;
        move();
        x = new Arith(tok, x, term());
    }
    return x;
}

Instructor: Dr. Liang Cheng

CSE302: Compiler Design

01/30/07
Translation to 3-Address Code

- Each node in the syntax tree may encapsulate various actions
  - Translation actions
  - Type checking actions
- Three-address code
  - \( x = y \, \text{op} \, z \) where \( x, y, \) and \( z \) are names (ids), constants, or compiler-generated temps
    - \( x = y, \, x[y] = z, \) or \( x = y[z] \)
  - Sequential execution except conditional or unconditional jumps
    - ifFalse \( x \) goto \( L \), ifTrue \( x \) goto \( L \), or goto \( L \)
Translation of Statements

- **Code layout for**
- **if** (expr) stmt

---

code to compute

expr into x

ifFlase x goto after

code for stmt

after

---

```java
class If extends Stmt {
    Expr E; Stmt S;
    public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
    public void gen() {
        Expr n = E.value();
        emit("ifFalse " + n.toString() + " goto " + after);
        S.gen();
        emit(after + ":");
    }
}
```
Expression Translation Guidelines

- No code is generated for ids and constants
  - They appear as addresses in instructions
- If a node $x$ of class Expr has an operator $\text{op}$, e.g. $i-j$, then an instruction is emitted
  - The value computed at node $x$ is stored at a temp: $t = i - j$
- Array accesses and assignments need to distinguish l-values and r-values
  - $2\times a[i]$ needs the r-value of $a[i]$
  - $a[i]=2$ needs the l-value of $a[i]$
3-Address Code Generation

- Perform tree traversal to generate three-address code
  - The function *gen* is called at the root of the syntax tree
  - All statement classes contain a function *gen*
Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Intermediate code generation (Section 2.8)
  - Symbol tables (Section 2.7)
- Summary and homework
Symbol Tables

- Hold info of source program constructs
  - Collected during analysis
  - Used for synthesis

- Support multiple declarations of the same identifier within a program
  - A separate symbol table for each scope
    - A program block
    - A class
Chained Symbol Tables for Nested Blocks

```c
{ bool w;
    { int x; int y;
        { int w; bool y; int z;
            ... w ...; ... x ...; ... y ...; ... z ...;
        }
        ... w ...; ... x ...; ... y ...;
    }
    ... w ...; ... x ...; ... y ...;
}
... w ...
```
Data Structure for Chained Symbol Tables

```java
public class Env {
    private Hashtable tab;
    protected Env prev;
    public Env(Env p) {table=new Hashtable(); prev=p; }
    public void put(String s, Symbol sm) {tab.put(s,sm); }
    public Symbol get(String s) {
        for (Env e=this; e!=null; e=e.prev) {
            Symbol found=e.table.get(s);
            if(found!=null) return found;
        }
        return null;
    }
}
```

Syntax-directed translation using this data structure
Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Intermediate code generation (Section 2.8)
  - Symbol tables (Section 2.7)
- Summary and homework