More Parsing

Lecture 3
CS 212 – Fall 2007

Recall

- A language (computer or human) has
  - An alphabet
  - Tokens (i.e., words)
  - Syntax (i.e., structure)
  - Semantics
- We know the alphabet
- The tokens are simple
- Syntax??
  - Syntax can be described by a *Context Free Grammar*
  - A grammar uses *productions* of the form V → w
  - V is a single *nonterminal* (i.e., it's not a token)
  - w is word made from both *terminals* (i.e., tokens) and nonterminals

Compiling Overview

- Compiling a program
  - Lexical analysis
    - Break program into tokens
  - Parsing
    - Analyze token arrangement
    - Discover structure
  - Code generation
    - Create code
- What you'll be doing
  - Lexical analysis
    - This will be given to you
  - Parsing
    - Recursive Descent Parsing
    - Build an Abstract Syntax Tree (AST)
  - Code generation
    - Use the AST to create code

An Extended Example

- A simple computer language
- Just 3 variables: x, y, z
- Just two statement types: assignment and do

```
x = 1; y = 1;
do 5:
  x = x * y;
  y = y + 1;
  end;
end.
```

- We can invent a grammar to describe legal programs
  - We need rules for building *expressions*, *statements*, and *programs*
  - Context Free Grammars are just what's needed to describe these rules

The Grammar

program → statement* **end .**

statement → name **=** expression **;**

statement →
 do expression **:** statement* **end ;**

expression → part [(**+** | **-** | ***** | **/**) part]

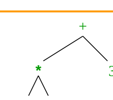
part → (name | number | **(** expression **)**)

name → (**x** | **y** | **z**)

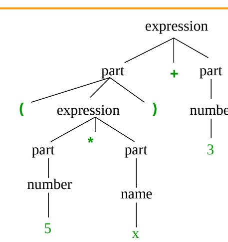
- Notation:
  - * indicates zero or more occurrences
  - [ ] indicates zero or one occurrence
  - (| |) indicates choice
- What is the parse tree for the expression (5 * x) + 3?

Abstract Syntax Tree

- We can build a parse tree, but an AST (*Abstract Syntax Tree*) is more useful
  - Idea is to show less grammar and more meaning



Abstract Syntax Tree

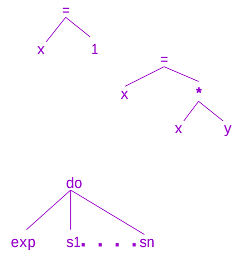


Parse Tree

Designing the AST

- We can decide how the AST should look for each of our language constructs

```
x = 1; y = 1;
do 5:
  x = x * y;
  y = y + 1;
  end;
end.
```



Recursive Descent Parsing

- Idea: Use the grammar to design a recursive program that builds the AST
- To parse a do-statement, for instance
  - We look for each terminal (i.e., token)
  - Each nonterminal (e.g., expression, statement) can handle itself—recursively
- The grammar tells how to write the program

```
public ASTNode parseDo {
  Make sure there is a "do" token;
  exp = parseExpression();
  Make sure there is a ":" token;
  while (not "end" token) {
        s = parseStatement();
        stList.add(s);
        }
  Make sure there is an "end" token;
  Make sure there is a ";" token;
  return DoNode(exp, stList);
}
```

In Practice

- We define a parent class ASTNode
- DoNode can be a subclass
- Each possible node in the AST will have its own subclass of ASTNode
- Some of the grammar's nonterminals don't correspond to nodes in the AST
  - E.g., statement, expression, part
- For these we don't want to create classes
  - But we do need recursive methods to parse these nonterminals

Does Recursive Descent Always Work?

- There are some grammars that cannot be used as the basis for recursive descent
  - A trivial example (causes infinite recursion):
    - S -> b
    - S -> Sa
- Can rewrite grammar
  - S -> b
  - S -> bA
  - A -> aA
- For some constructs Recursive Descent is hard to use
  - Can use a more powerful parsing technique (there are several, but not in this course)

Code Generation

- The same kind of recursive viewpoint can drive our code generation
  - This time we recurse on the AST instead of the grammar
  - Write the code for the root node; the subtrees (e.g., exp) can take care of themselves

```
class AssignmentStatement extends
    ASTNode {

    String var; ASTNode exp;

    public AssignmentNode (var, exp) {
          this.var = var;
          this.exp = exp;
    }

    public void generate ( ) {
          exp.generate( );
          // Exp result is left on stack
          Generate code to move top
          of stack into mem-location of
          var;
    }
}
```