Week 3
More Parsing

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 $V \rightarrow w$
  - $V$ 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

Building a Parse Tree
- Grammars can be used in two ways
  - A grammar defines a language (i.e., the set of properly structured sentences)
  - A grammar can be used to parse a sentence (thus, checking if the sentence is in the language)
- For us,
  - We will give you the grammar for Bali
  - The sentence is a Bali program
- You can show a sentence is in a language by building a parse tree (much like diagramming a sentence)
- Example: Show that $8 + x/5$ is a valid Expression (E) by building a parse tree
  - $\text{E} \rightarrow \text{T} \{ ( '+' | '-' ) \text{E} \}$
  - $\text{T} \rightarrow \text{F} \{ ( '*' | '/' ) \text{T} \}$
  - $\text{F} \rightarrow \{ ( n | w | '(' \text{E} ')' ) \}$
  - $()$ indicates 0 or more occurrences
  - $( | )$ indicates choice
  - $n$ is a number
  - $w$ is a word

Tree Terminology
- $M$ is the root of this tree
- $G$ is the root of the left subtree of $M$
- $B$, $H$, $J$, $N$, and $S$ are leaves
- $P$ is the parent of $N$
- $M$ and $G$ are ancestors of $D$
- $P$, $N$, and $S$ are descendents of $W$
- A collection of trees is called a ??

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

Designing the AST

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

Designing the AST

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
  - The grammar tells how to write the program

In Practice

- We define a parent class ASTNode
- DoStatement can be a subclass
  - The parseDo program can be used as the outline for the constructor
- 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 for these nonterminals
- One place to put such methods:
  - In the parent class (ASTNode)

Recursive Descent Parsing

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 → 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)
Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
  \[ S \rightarrow A | aAb \]
  \[ A \rightarrow \varepsilon | aAb \]
  \[ B \rightarrow \varepsilon | aB | bB \]
  - The string aabb can be parsed in two ways
- This kind of ambiguity sometimes shows up in programming languages
- if E1 then if E2 then S1 else S2

- This ambiguity actually affects the program’s meaning
- How do we resolve this?
  - Provide an extra non-grammar rule (e.g., the else goes with the closest if)
  - Modify the grammar (e.g., an if-statement must end with a ‘;’)
  - Other methods (e.g., Python uses amount of indentation)
- We try to avoid syntactic ambiguity in Bali

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 AssignmentStatement {
        var = variable on left;
        exp = expression on right;
    }
    public void generate () {
        exp.generate();
        // Exp result is left on stack
        Generate code to move top of stack into mem location of var;
    }
}