Announcements

- Part 1 (both Compiler & GBA) is due on Friday
- Sections have been split
  - GBA sections
    - There is a GBA section at each section time:
      - M12:20, M7:30, W7:30
    - Location for all GBA sections: Hollister 401
  - Compiler sections
    - Using rooms originally assigned to sections
      - M12:20 Olin Hall 245
      - M7:30 Upson 205
      - W7:30 Upson 205

Compilers

- Basically, a compiler
  - Translates one language (e.g., Java)
  - Into another (e.g., JBC: Java Byte Code)
- Why do this?
  - Idea is to translate a language that is easy for humans to understand into one that is easy for a computer to understand
  - This idea was initially controversial!
- Typical compiler phases
  - Lexical analysis: Breaking input into tokens
  - Parsing: Understanding program's structure
  - Optimization: Making the code more efficient (e.g., precomputing constant expressions, avoid recomputing)
  - Code Generation: Creating code in a simpler language (e.g., JBC, machine code)

Parts of a Language

- Human language
  - alphabet \(\rightarrow\) words \(\rightarrow\) sentences \(\rightarrow\) paragraphs \(\rightarrow\) chapters \(\rightarrow\) book
- Computer language
  - alphabet \(\rightarrow\) tokens \(\rightarrow\) statements \(\rightarrow\) program
- Both types of language have
  - Syntax
    - Structural rules
  - Semantics
    - Meaning

Syntax

- Remember diagramming sentences? This was syntax!
  - sentence \(\rightarrow\) noun-phrase verb-phrase
    - noun-phrase \(\rightarrow\) article (adjective) noun
    - verb-phrase \(\rightarrow\) verb direct-object
    - direct-object \(\rightarrow\) noun-phrase
  - The hungry mouse ate the cheese.
  - The shiny elbow drank the automobile.

Syntax vs. Semantics

- Syntax = structure
  - Semantics = meaning
- Legal syntax does not imply valid meaning
- Examples of semantic rules for a programming language
  - Variables must be declared before use
  - Division by zero causes an error
  - The then-clause is executed only if the if-expression is True
- It's relatively easy to define valid syntax (especially if we get to invent the language)
- It's harder to specify semantics
- How can we specify semantics?
  - Formally, using logic (axiomatic semantics)
  - Informally, using explanations in English
  - By reference to a canonical implementation
Compiling Overview

- Compiling a program
  - Lexical analysis
  - Break program into tokens
  - Parsing
  - Analyze token arrangement
  - Discover structure
  - Code generation
  - Create code

- For a computer language, each phase can be completed before the next one begins

Understanding a sentence

- Lexical analysis
  - Break sentence into words
- Parsing
  - Analyze word arrangement
  - Discover structure

Understanding

- Understand the sentence

For human language, there is feedback between parsing and understanding

Lexical Analysis

- Goal: divide program into tokens
- Tokens can be specified using regular expressions
  - $a^*$: repeat a zero or more times
  - $a^+$: repeat a one or more times
  - $\{a,b,c\}$: choose one of $a$, $b$, or $c$

- Examples
  - operator = $\{+,-,*,/\}$
  - integer = $\{0123456789\}+$

- For the Compiler Project, we give you the lexical analyzer (or tokenizer)

Building a Tokenizer

- For tokens, can tell what to do next by checking a few characters (usually 1 character) ahead
  - Example: If it starts with a letter, it's a word; the word ends when you reach a non-alphanumeric character
  - Example: If it starts with a digit, it's a number; if you reach a decimal point, it's a floating point number...

- Java has a class (introduced in Java 5) `Java.util.Scanner`

- Early computer languages were not parsed based on tokens

Java.util.Scanner

- Can recognize identifiers, numbers, quoted strings, and various comment styles
- This is more useful than the earlier (Java 1.0) `java.io.StreamTokenizer`

Specifying Syntax

- How do we specify syntax?
  - Can use a grammar
  - Can use a syntax chart

- Example grammar

  - Anything in single-quotes is a token; `n` and `w` represent a number token and a word token, respectively; `()` are used for grouping; `|` indicates choice; `*` indicates zero-or-more occurrences

  - $E \rightarrow T \left( (\ + \ | \ - \ ) \ T \right)^*$
  - $T \rightarrow F \left( (\ * \ | \ / \ ) \ F \right)^*$
  - $F \rightarrow n \ | \ w \ | \ ( \ E \ )$

- Example syntax charts

- Language of example: all strings of the form $anbn$ for $n \geq 0$

- CS 381 for more detail

Grammars

- The rules in a grammar are called productions
- Syntax rules can be specified using a Context-Free grammar

  - All productions are of the form $V \rightarrow w$
  - $V$ is a single nonterminal (i.e., it's not a token)
  - $w$ is word made from terminals (i.e., tokens) and nonterminals

  - Example: $E \rightarrow T \left( (\ + \ | \ - \ ) \ T \right)^*$

- In simple examples, uppercase is used for nonterminals, lowercase for terminals

- Example: $a^*$ represents the empty string:
  - $A \rightarrow a$
  - $A \rightarrow aB$

- A grammar defines a language

  - Example: all strings of the form $ab^n$ for $n \geq 0$

  - CS 381 for more detail

Building a Parse Tree

- Grammars can be used in two ways

  - A grammar defines a language
  - A grammar can be used to parse a sentence (thus, checking if the sentence is in the language)

  - For the Compiler Project:
    - We 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 $8\times5$ is a valid Expression ($E$) by building a parse tree:

    - $E \rightarrow T \left( (\ * \ | \ / \ ) \ T \right)^*$
    - $T \rightarrow F \left( (\ * \ | \ / \ ) \ F \right)^*$
    - $F \rightarrow n \ | \ w \ | \ ( \ E \ )$
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 ??

Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
- This string can be parsed in two ways
- This kind of ambiguity sometimes shows up in programming languages

An Extended Example

- A simple computer language
- Each variable is a single letter
- Just two statement types: assignment and do

We can invent a grammar to describe legal programs

- We need rules for building expressions, statements, and programs
- We create a Context Free Grammar for our simple language

The Grammar

- program → statement* end .
- statement → name = expression ;
- statement → do expression : statement* end ;
- expression → part \[ ( part | part + part | part - part | part * part | part / part ) \]
- part → ( name | number | ( expression ) )
- name → singleLowercaseLetter

- 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 invent how the AST should look for each of our language constructs
- Abstract Syntax Tree
- Parse Tree

Notation:

\[ x = 1; y = 1; \]
\[ do 5: \]
\[ x = x * y; \]
\[ y = y + 1; \]
\[ end. \]
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
  + A -> a
- 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 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;
    }
}