Lexical Analysis and Parsing

Lecture 2
CS 212 - Fall 2007

Recall
• Compiling Java
  • Java Program
  • Java Compiler
  • Java Byte Code (JBC)
  • JVM Interpreter
  • Compiling Bali
  • Bali Program
  • Bali Compiler (you write this)
  • Sam-Code

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 → words → sentences → paragraphs → chapters → book
• Computer language
  • alphabet → tokens → statements → program
• Both types of language have
  • Syntax
  • Structural rules
  • Semantics
  • Meaning

Syntax
• Remember diagramming sentences? This was syntax!
  • sentence → noun-phrase verb-phrase
  • noun-phrase → article (adjective) noun verb-phrase → verb direct-object
  • direct-object → 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
  - Parsing
  - Code generation
  - Create code
- For a computer language, each phase can be completed before the next one begins
- For human language, there is feedback between parsing and understanding

Lexical Analysis

- Goal: divide program into tokens
- Tokens: individual units or words of a language
- Tokens can be specified using regular expressions
  - a* = repeat a zero or more times
  - a+ = repeat a one or more times
  - [abc] = choose one of a, b, or c
  - . = matches any one character
- Examples:
  - operator = [ + - * / ]
  - integer = [0123456789]+

For Bali, 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,…

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; parentheses are used for grouping; | indicates choice; * indicates zero-or-more occurrences)
  - E → T ( ( '+' | '-' ) T )*
  - T → F ( ( '*' | '/' ) F )*
  - F → n | w | '(' E ')'

Example syntax charts

E:

T:

F:

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 → w
- V is a single nonterminal (i.e., it’s not a token)
- w is word made from terminals (i.e., tokens) and nonterminals
- In simple examples, uppercase is used for nonterminals, lowercase for terminals
- Example (c represents the empty string):
  - A → c
  - A → aB
- A grammar defines a language
- Language of example: all strings of the form a^n b^n for n ≥ 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 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 8x/5 is a valid Expression (E) by building a parse tree
  - E → T ( ( '*' | '/' ) T )*
  - T → F ( ( '+' | '-' ) F )*
  - F → 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 descendants of W
- A collection of trees is called a ??

Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
  \[ S \rightarrow A \mid \text{all} \]
  \[ A \rightarrow \epsilon \mid aAb \]
  \[ B \rightarrow \epsilon \mid aB \mid bB \]
- 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 ’fi’)
  - Other methods (e.g., Python uses amount of indentation)
- We try to avoid syntactic ambiguity in Bali