Comp 411
Principles of Programming Languages
Lecture 3
Parsing

Corky Cartwright
Swarat Chaudhuri

August 26, 2011

Top Down Parsing

- What is a context-free grammar (CFG)?
  A recursive definition of a set of strings; it is *identical* in format to the data definitions used in Comp 210/211 *except* for the fact that it defines sets of strings (using concatenation) rather than sets of trees (objects/structs) using tree construction. The *root symbol* of a grammar generates the language of he grammar. In other words, it designates the syntax of complete programs.
- Example. The language of expressions generated by **<expr>**

```
<expr> ::= <term>   |  <term> + <expr>
<term> ::= <number>   |  <variable>   |  ( <expr> )
```

- Some sample strings generated by this CFG

  5 **5+10** **5+10+7** **(5+10)+7**

- What is the fundamental difference between generating strings and generating trees?
  - The derivation of a generated tree is manifest in the structure of the tree.
  - The derivation of a generated string is *not* manifest in the structure of the string; it must be *reconstructed* by the parsing process. This reconstruction may be *amibiguous* and it may be costly in the general case (O(n^3)). Fortunately, *deterministic* (LR(*k*)) parsing is linear.

Top Down Parsing cont.

- Data definition corresponding to sample grammar:

```
Expr  = Expr + Expr  |  Number  |  Variable
```

- Why is the data definition simpler? (Why did we introduce the syntactic category `<term>` in the CFG?)

- Consider the following example:

```
5+10+7
```

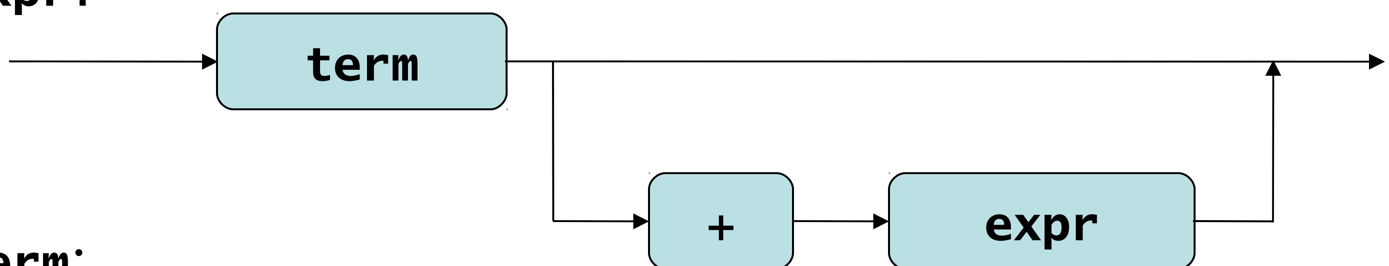
- Are strings a good data representation for programs?

- Why do we use string representations for programs?

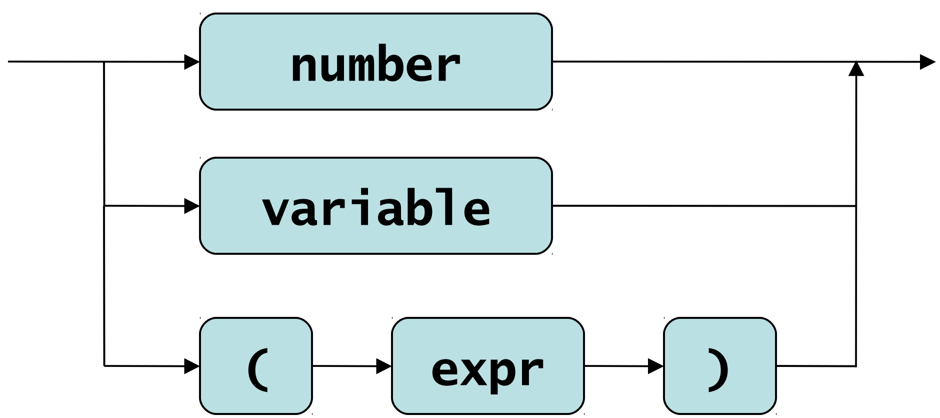
Parsing algorithms

- Top-down (predictive) parsing: use *k* token look-ahead to determine next syntactic category.
- Simplest description uses *syntax diagrams*

expr:



term:



Key Idea in Top Down Parsing

- Use *k* token look-ahead to determine which direction to go at a branch point in the current syntax diagram.
- Example: **`5+10`**
  - Start parsing by reading first token **`5`** and matching the syntax diagram for **`expr`**
  - Must recognize a **`term`**; invoke rule (diagram) for **`term`**
  - Select the **`number`** branch (path) based on current token **`5`**
  - Digest the current token to match **`number`** and read next token **`+`**; return from **`term`** back to **`expr`**
  - Select the **`+`** branch in **`expr`** diagram based on current token
  - Digest the current token to match **`+`** and read the next token **`10`**
  - Must recognize an **`expr`**; invoke rule (diagram) for **`expr`**
  - Must recognize a **`term`**; invoke rule (diagram) for **`term`**
  - Select the **`number`** branch based on current token **`10`**
  - Digest the current token to match **`number`** and read next token **`EOF`**
  - Return from **`term`**; return from **`expr`**

Designing Grammars for Top-Down Parsing

- Many different grammars generate the same language (set of strings):
- Requirement for any efficient parsing technique: determinism (non-ambiguity)
- For deterministic *top-down* parsing, we must design the grammar so that we can always tell what rule to use next starting from the root of the parse tree by looking ahead some small number (*k*) of tokens (formalized as LL(*k*) parsing).
- For top down parsing
  - Eliminate left recursion; use right recursion instead
  - Factor out common prefixes (as in syntax diagrams)
  - Use iteration in syntax diagrams instead of right recursion where necessary
  - In extreme cases, hack the lexer to split token categories based on local context

Other Parsing Methods

When we parse a sentence using a CFG, we effectively build a (parse) tree showing how to construct the sentence using the grammar. The root (start) symbol is the root of the tree and the tokens in the input stream are the leaves.

Top-down (predictive) parsing is simple and intuitive, but is is not as powerful a deterministic parsing strategy as bottom-up parsing which is much more tedious. Bottom up deterministic parsing is formalized as LR(k) parsing.
Every LL(k) grammar is also LR(1) but many LR(1) grammars are not LL(k) for any k.

No sane person manually writes a bottom-up parser. In other words, there is no credible bottom-up alternative to recursive descent parsing. Bottom-up parsers are generated using parser-generator tools which until recently were almost universally based on LR(k) parsing (or some bottom-up restriction of LR(k) such as SLR(k) or LALR(k)). But some newer parser generators like javacc are based on LL(k) parsing. In DrJava, we have several different parsers including both recursive descent parsers and automatically generated parsers produced by javacc.

Why is top-down parsing making inroads among parser generators? Top-down parsing is much easier to understand and more amenable to generating intelligible syntax diagnostics. Why is recursive descent still used in production compilers? Because it is straightforward (if tedious) to code, supports sensible error diagnostics, and accommodates *ad hoc* hacks (e.g., use of state).to get around the LL(k) restriction.

If you want to learn about the mechanics of bottom-up parsing, take Comp 412.