Comp 411
Principles of Programming Languages
Lecture 3
Parsing

Corky Cartwright
January 13, 2017
Top Down Parsing cont.

- We restrict our attention to LL(k) grammars because they can be parsed deterministically using a top-down approach. Every LL(k) grammar is LR(k). LR(k) grammars are those that can be parsed deterministically bottom-up using k-symbol lookahead. For every LR(k) grammar, there is an equivalent LR(1) grammar. For more information, take Comp 412.

- Data definition of abstract syntax corresponding to preceding sample grammar:
  \[ \text{Expr} = \text{Expr} + \text{Expr} \mid \text{Number} \mid \text{Variable} \]

  Note that the preceding defines a set of trees not strings.

- Why is the data definition simpler? (Why did we introduce the syntactic category \texttt{<term>} in the CFG but not in the data definition?)

- The parser returns the abstract syntax tree (AST) for the input program. In the literature, parsers often return parse trees which must be converted to ASTs.

- Consider the following example: \texttt{5+10+7}

- Are strings a good data representation for programs?

- Why do we use external string representations for source programs?
Parsing algorithms

- Top-down (predictive) parsing: use $k$ token look-ahead to determine next syntactic category.
- Simplest description uses *syntax diagrams*; see http://www.bottlecaps.de/rr/ui

```
expr:
  term
  +    expr

term:
  number
  variable
  (expr)
```
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: parsing \( 5+10 \) as an expr
  - 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

- Parsing is fundamentally recursive because syntactic rules are recursive
Structure of Recursive Descent Parsers

• The parser includes a method/procedure for each non-trivial non-terminal symbol in the grammar.

• For trivial non-terminals (like `number`) that correspond to individual tokens, the token (or the corresponding object in the AST definition) is the AST so we can directly construct the AST making a separate procedure unnecessary.

• The procedure corresponding to a non-terminal may take the first token of the text corresponding to a non-terminal as an argument; this choice is natural *if that token has already been read*. It is cleaner coding style to omit this argument if the token has not already been read.

• Most lexers support a `peek` operation that reveals the next token without actually reading it (consuming it from the input stream). In some cases, this operation can be used to cleanly avoid reading a token beyond the syntactic category being recognized. The class solution does not always follow this strategy; perhaps it should.
Designing Grammars for Top-Down Parsing

• Many different grammars generate the same language (set of strings):
• Requirement for any efficient parsing technique: determinism of (non-ambiguity) of the grammar defining the language.
• For deterministic top-down parsing, we must design the grammar so that we can always tell what rule to use next starting from the bottom (leaves) 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.

Example: in DrJava, we introduced \(\gg\) and \(\gg\gg\) as extra tokens when Java 5 was introduced because \(\gg\) can either be an infix right shift operator or consecutive closing pointy brackets in a generic type. With this change to the lexer, it was easy to revise a top-down Java 4 (1.4) parser to create a Java 5 parser. Without this change to the lexer, top-down parsing of Java 5 looked really ugly, possibly requiring unbounded look-ahead, which our parser generator (javacc) did not support.
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 not as powerful (in terms of the set of grammars it accommodates) as bottom-up deterministic 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 and ANTLR 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 a bit 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 details and mechanics of parsing, take Comp 412.