Comp 411
Principles of Programming Languages
Lecture 3
Parsing

Corky Cartwright
January 29, 2021

Top Down Parsing

- What is a context-free grammar (CFG)?
  - A recursive definition of a set of strings; it is *identical* in format to recursive data definitions of algebraic types (as in Ocaml or Haskell) *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 the grammar. In other words, it designates the string 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 construction of a generated tree is manifest in the structure of the tree.
  - The construction 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 *ambiguous* and it may be costly in the general case ($O(n^3)$). Fortunately, parsing the language for *deterministic* (LL(*k*), LR(*k*)) grammars is linear.

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 LL(*k*) grammar, there is an *equivalent* LR(1) grammar. LR(*k*) is more general than LL(*k*) parsing but less friendly in practice. The dominant parser generators for Java, ANTLR and JavaCC are based on LL(*k*) grammars. (Conjecture: the name ANTLR is a contraction of anti-LR.) For more information, take Comp 412.
- Data definition of abstract syntax corresponding to preceding sample grammar

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

  - Note that the syntax of the preceding production is nearly identical to what we use for CFGs but we interpret infix terminal symbols like **+** as the name of binary tree node constructor. For tree node constructors that are not binary we typically use prefix notation. Only one terminal can appear within a variant on the right-hand-side (RHS) of a production in a tree grammar.
  - Why is the data definition simpler that the corresponding CFG? Because the nesting structure of program phrases is built-in to the definition of abstract syntax but must be explicitly encoded using parentheses or multiple productions (encoding the precedence hierarchy) in CFGs.

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 look-ahead. For every LL(*k*) grammar, there is an *equivalent* LR(1) grammar. LR(*k*) is more general than LL(*k*) parsing but less friendly in practice. The dominant parser generators for Java, ANTLR and JavaCC are based on LL(*k*) grammars. (Conjecture: the name ANTLR is a contraction of anti-LR.) For more information, take Comp 412.

- Data definition of abstract syntax corresponding to preceding sample grammar

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

  - Note that the syntax of the preceding production is nearly identical to what we use for CFGs but we interpret infix terminal symbols like **+** as the name of binary tree node constructor. For tree node constructors that are not binary we typically use prefix notation. Only one terminal can appear within a variant on the right-hand-side (RHS) of a production.
  - Why is the data definition simpler that the corresponding CFG? Because the nesting structure of program phrases is built-in to the definition of abstract syntax but must be explicitly encoded using parentheses *or* multiple productions in CFGs. The former approach is embodied in "Lisp-like" languages.

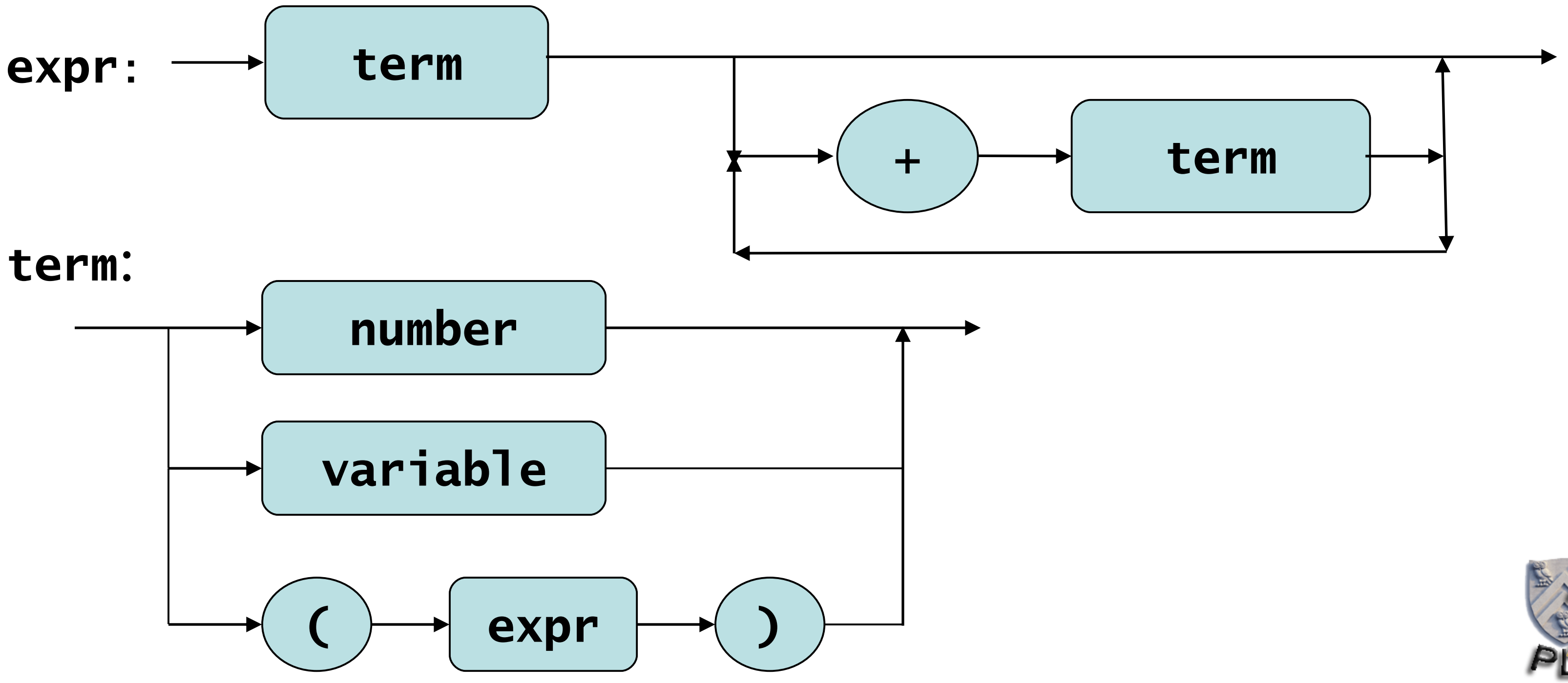
Top Down Parsing cont.

- The parser returns the abstract syntax tree (AST) for the input program. In the literature on parsing, parsers often return parse trees (containing irrelevant non-terminal nodes) which must be converted to ASTs.
- Consider the following example: `5-10+7`
  - What is the corresponding abstract syntax tree? It depends on the implicit associativity of `+` and `-` : `(5-10)+7` or `5-(10+7)`
  - In a Lisp-like language, we must write

    `(+ (- 5 10) 7)` or `(- 5 (+ 10 7))`
- Are strings (unless they are written in Lisp-like syntax) a good data representation for programs? Is a Lisp-like string as good an internal representation as a tree? (Note: such a string representation must still be "parsed" to extract subtrees.)
- Why do we use external string representations for source programs? Humans find such representations more intelligible perhaps because this convention is followed in mathematics.

Parsing algorithms

- Top-down (predictive) parsing: use *k* token look-ahead to determine next syntactic category.
- Simplest description uses *syntax diagrams* which actually support a slightly more general framework than LL(*k*) parsing because they can have iterative loops which correspond to both left-associative and right-associative operators; the parser designer can decide for such iterative loop whether to use left-association or right-association. The former is typically chosen. In addition, the longest possible match is chosen when parsing using syntax diagrams which can eliminate ambiguity in the corresponding CFG. For more about LL(*k*) grammars and syntax diagrams, see http://www.bottlecaps.de/rr/ui



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 the **`expr`** diagram based on current token
  - Digest the current token to match **`+`** and read the next token **`10`**
  - 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 and Syntax Diagrams for Top-Down Parsing

- Many different grammars and syntax diagrams generate the same language (set of strings of symbols):
- Requirement for any efficient parsing technique: determinism of (non-ambiguity) of the *grammar* or *syntax diagrams* defining the language. In addition, the precedence of operations must be correctly represented in parse trees (or the abstract syntax implied by syntax diagrams). This information is *not* captured in the concept of "language equivalence".
- For deterministic *top-down* parsing using a grammar or syntax diagram, we must design the grammar or syntax diagram 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 grammars].

Designing Grammars and Syntax Diagrams for Top-Down Parsing (cont.)

To create such a grammar or syntax diagram:

- Eliminate left recursion; use right recursion (in an LL(*k*) grammar) or iteration (in syntax diagrams) instead. A syntax diagram is more expressive in practice because iteration naturally corresponds to left associativity (using iteration in a recursive descent parser).
- Factor out common prefixes (standard practice in a syntax diagrams)
- In extreme cases, hack the lexer to split token categories based on local context.
  - Example: in DrJava, we introduced >> and >>> as extra tokens when Java 5 was introduced because >> 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 an LL(*k*) 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 using an LL(*k*) grammar or a syntax diagram is simple and intuitive, but it is not as powerful (in terms of the set of languages [strings] 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 LR(*k*) and has an equivalent LR(1) grammar but many LR(1) grammars do not equivalent LL(*k*) grammars 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 originally were based on LR(*k*) parsing (or some bottom-up restriction of LR(*k*) such as SLR(*k*) or LALR(*k*)). But more recent 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.