CS412/413

Introduction to Compilers
Radu Rugina

Lecture 5: Context-Free Grammars
29 Jan 03

Outline

- Context-Free Grammars (CFGs)
- Derivations
- Parse trees and abstract syntax
- Ambiguous grammars

Lexical Analysis

- Translates the program (represented as a stream of characters) into a sequence of tokens
- Uses regular expressions to specify tokens
- Uses finite automata for the translation mechanism
- Lexical analyzers are also referred to as lexers or scanners

Where We Are

Source code (character stream)
if (b == 0) a = b;

Token stream
if (b == 0) a = b;

Abstract Syntax Tree (AST)

Semantic Analysis

Syntax Analysis Example

Source code
(token stream)

{ if (b == 0) a = b;
while (a != 1) { stdio.print(a);
 a = a - 1; }
}

Parsing Analogy

- Syntax analysis for natural languages: recognize whether a sentence is grammatically well-formed & identify the function of each component.

"I gave him the book"

sentence

subject: I verb: gave indirect object: him noun phrase article: the noun: book
Syntax Analysis Overview

- **Goal:** determine if the input token stream satisfies the syntax of the program

- **What we need for syntax analysis:**
  - An expressive way to describe the syntax
  - An acceptor mechanism that determines if the input token stream satisfies the syntax description

- **For lexical analysis:**
  - Regular expressions describe tokens
  - Finite automata = acceptors for regular expressions

---

Why Not Regular Expressions?

- Regular expressions can expressively describe tokens
  - easy to implement, efficient (using DFAs)
- Why not use regular expressions (on tokens) to specify programming language syntax?
- Reason: they don’t have enough power to express the syntax in programming languages
  - Language of balanced parentheses
    - {{}, {}, {((()))}, {(()), ()}}
    - We need unbounded counting!

---

Context-Free Grammars

- Use Context-Free Grammars (CFG):
  - Terminal symbols = token or ε
  - Non-terminal symbols = syntactic variables
  - Start symbol S = special nonterminal
  - Productions of the form LHS → RHS
  - LHS = a single nonterminal
  - RHS = a string of terminals and non-terminals
  - Specify how non-terminals may be expanded

- **Language** generated by a grammar = the set of strings of terminals derived from the start symbol by repeatedly applying the productions

  - L(G) denotes the language generated by grammar G

Example

- Grammar for balanced-parenthesis language:
  - S → {} S
  - S → ε
  - 1 nonterminal: S
  - 2 terminals “{” and “}”
  - Start symbol: S
  - 2 productions:
    - If a grammar accepts a string, there is a derivation of that string using the productions:
      - S → (S)S → (S) ε → {}(S) S ε → (S) ε → {}(S) ε → {}(ε) ε → ε

---

Context-Free Grammars

- Shorthand notation: vertical bar for multiple productions
  - S → a S a | T
  - T → b T b | ε

- Context-free grammars = powerful enough to express the syntax in programming languages
- Derivation = successive application of productions starting from S (the start symbol)
- The acceptor mechanism = determine if there is a derivation for an input token stream

---

Grammars and Acceptors

- Acceptors for context-free grammars

  - Syntax analyzers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted
  - Various kinds: LL(k), LR(k), SLR, LALR

<table>
<thead>
<tr>
<th>Context-Free Grammar</th>
<th>Token Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>s</td>
</tr>
<tr>
<td>Yes, if s ∈ L(G)</td>
<td>No, if s ∉ L(G)</td>
</tr>
</tbody>
</table>
RE is Subset of CFG

- Inductively build a grammar for each regular expression
  - $\varepsilon \rightarrow S \rightarrow E$
  - $a \rightarrow S \rightarrow a$
  - $R_1 R_2 \rightarrow S \rightarrow S_1 S_2$
  - $R_1 \mid R_2 \rightarrow S \rightarrow S_1 \mid S_2$
  - $R_1^* \rightarrow S \rightarrow S_1 S \mid \varepsilon$

where:
- $G_1$ grammar for $R_1$, with start symbol $S_1$
- $G_2$ grammar for $R_2$, with start symbol $S_2$

Sum Grammar

- Grammar:
  - $S \rightarrow E + S \mid E$
  - $E \rightarrow \text{number} \mid ( S )$

- Expanded:
  - 4 productions
  - 2 non-terminals: $S, E$
  - 4 terminals: $(\ ) \mid \text{number}$
  - Start symbol $S$

- Example accepted input:
  - $(1 + 2 + (3+4)) + 5$

Derivation Example

- $S \rightarrow E + S \mid E$
  - $E \rightarrow \text{number} \mid ( S )$

Derive $(1 + 2 + (3+4)) + 5$:

- $S \rightarrow E + S \rightarrow (E + S) + S$
- $\rightarrow (1 + S) + S \rightarrow (1 + E + S) + S$
- $\rightarrow (1 + 2 + S) + S \rightarrow (1 + 2 + E) + S$
- $\rightarrow (1 + 2 + (3 + S)) + S$
- $\rightarrow (1 + 2 + (3 + E)) + S$
- $\rightarrow (1 + 2 + (3 + 4)) + E$
- $\rightarrow (1 + 2 + (3 + 4)) + S$

Constructing a Derivation

- Start from $S$ (start symbol)
- Use productions to derive a sequence of tokens from the start symbol
- For arbitrary strings $\alpha, \beta$ and $\gamma$ and for a production $A \rightarrow \beta$
  - a single step of derivation is:
    - $\alpha A \gamma \Rightarrow \alpha \beta \gamma$
    - (i.e., substitute $\beta$ for an occurrence of $A$)

- Example:
  - $S \rightarrow E + S$
  - $(S + E) + E \rightarrow (E + S + E) + E$

Derivation $\Rightarrow$ Parse Tree

- $E \rightarrow S$
- $S \rightarrow S_1\ S_2$
- $S \rightarrow S_1\ S_2$
- $E \rightarrow S_1\ S_2$
- $E \rightarrow S_1\ S_2$

- Parse Tree = tree representation of the derivation
  - Leaves of tree are terminals
  - Internal nodes: non-terminals
  - No information about order of derivation steps

Parse Tree vs. AST

- Parse tree also called "concrete syntax"

- Discards (abstracts) unneeded information
Derivation order

- Can choose to apply productions in any order; select any non-terminal: $\alpha \beta \gamma \Rightarrow \alpha \beta \gamma$
- Two standard orders: left- and right-most -- useful for different kinds of automatic parsing
- Leftmost derivation: In the string, find the left-most non-terminal and apply a production to it $E + S \rightarrow 1 + S$
- Rightmost derivation: find right-most non-terminal...etc. $E + S \rightarrow E + E + S$

Example

- $S \rightarrow E + S \mid E$
  $E \rightarrow \text{number} \mid (S)$
- Left-most derivation
  $S \rightarrow E + S \rightarrow E \rightarrow (E + S) \rightarrow (E + S) + S \rightarrow (1 + S) + S \rightarrow (1 + 2 + 3 + 4) + S \rightarrow (1 + 2 + 3 + 4) + 5$
- Right-most derivation
  $S \rightarrow E + S \rightarrow E \rightarrow (E + S) \rightarrow (E + E + S) \rightarrow (E + E + S) + S \rightarrow (E + E + (E + S)) + S \rightarrow (E + E + (E + S)) + 5 \rightarrow (E + E + (E + S)) + 5$
- Same parse tree: same productions chosen, diff. order

Parse Trees

- In example grammar, left-most and right-most derivations produced identical parse trees
- + operator associates to right in parse tree regardless of derivation order

An Ambiguous Grammar

- + associates to right because of right-recursive production $S \rightarrow E + S$
- Consider another grammar:
  $S \rightarrow S + S \mid S \star S \mid \text{number}$
- Ambiguous grammar = different derivations produce different parse trees

Differing Parse Trees

$S \rightarrow S + S \mid S \star S \mid \text{number}$

- Consider expression $1 + 2 \star 3$
- Derivation 1: $S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + 1 \star 3$
  $\rightarrow 1 + 2 \star 3 \rightarrow 1 + 2 \star 3$
- Derivation 2: $S \rightarrow S \star S \rightarrow S \star 3 \rightarrow S \star 3 \rightarrow S \star 2 \star 3 $
  $\rightarrow S + 2 \star 3 \rightarrow 1 + 2 \star 3$

Impact of Ambiguity

- Different parse trees correspond to different evaluations!
- Meaning of program not defined

$\begin{cases} 1 + 2 \star 3 = 7 \\ 1 + 2 \star 3 = 9 \end{cases}$
Eliminating Ambiguity

- Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left

\[
S \rightarrow S + T \mid T \\
T \rightarrow T \ast \text{num} \mid \text{num}
\]

- \(T\) non-terminal enforces precedence
- Left-recursion : left-associativity

Context Free Grammars

- Context-free grammars allow concise syntax specification of programming languages
- A CFG specifies how to convert token stream to parse tree (if unambiguous!)