CS412/413
Introduction to Compilers
Radu Rugina

Lecture 5: Context-Free Grammars
04 Feb 04

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)

Syntax Analysis (Parsing)

Semantic Analysis

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 that 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
- Example: nested constructs (blocks, expressions, statements)
  - Language of balanced parentheses
    - \( \{ ( ) \} \)
    - \& We need unbounded counting!

Context-Free Grammars

- **Use Context-Free Grammars (CFG):**
  - Terminal symbols = token or \( \epsilon \)
  - Non-terminal symbols = syntactic variables
  - Start symbol \( S \) = special nonterminal
  - Productions of the form \( LHS \rightarrow RHS \)
    - \( LHS = \) 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:**
  - **\( S \rightarrow \{ \} S \)**
  - **\( S \rightarrow \epsilon \)**
  - 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:
      - \( S \rightarrow \{ \} S \rightarrow \{ \} \epsilon \rightarrow \{ \} \epsilon \rightarrow \{ \} \epsilon \)

Context-Free Grammars

- **Shorthand notation:** vertical bar for multiple productions
  - \( S \rightarrow a S a \mid T \)
  - \( T \rightarrow b T b \mid \epsilon \)

  - **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**
  - **Yes, if** \( s \in L(G) \)
  - **No, if** \( s \notin L(G) \)

  - **Syntax analyzers (parsers) =** CFG acceptors which also output the corresponding derivation when the token stream is accepted
    - Various kinds: LL(1), LR(1), SLR, LALR
RE is Subset of CFG

- Inductively build a grammar for each regular expression
  - $\epsilon \rightarrow \epsilon$
  - $a \rightarrow a$
  - $R_1R_2 \rightarrow S_1S_2$
  - $R_1 \rightarrow R_2 \rightarrow S_1 \rightarrow S_2$
  - $R_2 \rightarrow S_1S_2 \rightarrow \epsilon$

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:
  - $S \rightarrow E + S \mid \epsilon$
  - $E \rightarrow $ number $\mid (S)$

- Expanded:
  - 4 productions
  - 2 non-terminals: S E
  - 4 terminals: ( ) + number
  - start symbol $S$

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

Derivation Example

$S \rightarrow E + S \mid E$
$E \rightarrow $ number $\mid (S)$

Derive $(1 + 2 + (3 + 4)) + 5$:
- $S \rightarrow E + S \rightarrow (S) + S \rightarrow (E + S) + S$
  - $(1 + S) + S \rightarrow (1 + E + S) + S$
  - $(1 + 2 + S) + S \rightarrow (1 + 2 + E) + S$
  - $(1 + 2 + (3 + S)) + S$
  - $(1 + 2 + (3 + E)) + S$

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

- Parse Tree $\Rightarrow$ 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 A: αAγ → αAγ
- 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 → 1 + S
- Rightmost derivation: find right-most non-terminal...etc.
  E + S → E + E + S

Example

- S → E + S | E
  E → number | (S)
- Left-most derivation
  S → E + S → (E + S) → (1 + S) → (1 + S) + S → (1 + S + S) + S → (1 + 2 + (3 + 4)) + S → (1 + 2 + (3 + 4)) + S → (1 + 2 + (3 + 4)) + S → (1 + 2 + (3 + 4)) + S → (1 + 2 + (3 + 4)) + S → (1 + 2 + (3 + 4)) + S
- Right-most derivation
- 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 → E + S
- Consider another grammar:
  S → S + S | S * S | number
- Ambiguous grammar = different derivations produce different parse trees

Differing Parse Trees

S → S + S | S * S | number
- Consider expression 1 + 2 * 3
- Derivation 1: S → S + S → 1 + S → 1 + S * S → 1 + 2 * S → 1 + 2 * 3
- Derivation 2: S → S * S → S * 3 → S + S * 3 → S + 2 * 3 → 1 + 2 * 3

Impact of Ambiguity

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

= 7

= 9
Eliminating Ambiguity

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

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

- 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!)