CS412/413

Introduction to Compilers
Radu Rugina

Lecture 5: Context-Free Grammars
30 Jan 02

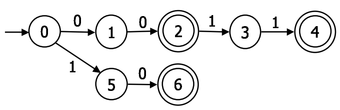
Outline

- JLex clarification

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

JLex: Clarification

- JLex tries to find the longest matching sequence
- Problem: what if the lexer goes past a final state of a shorter token, but then doesn't find any other longer matching token later?
- Consider R = 00 | 10 | 0011 and input w = 0010



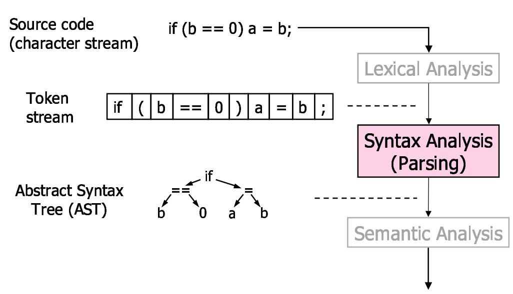
- We reach state 3 with no transition on input 0!
- Solution: record the last accepting state

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

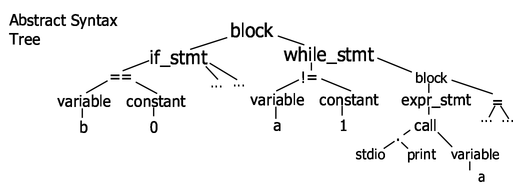


Syntax Analysis Example

Source code
(token stream)

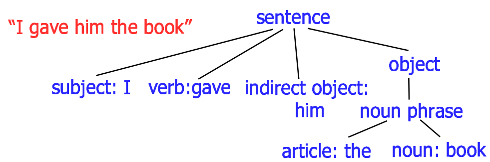
```
{
  if (b == (0)) a = b;
  while (a != 1) {
    stdio.print(a);
    a = a - 1;
  }
}
```

Abstract Syntax
Tree



Parsing Analogy

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

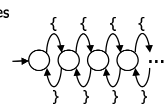


Syntax Analysis Overview

- 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 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

S → a S a
S → T
T → b T b
T → ε

- 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
  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) ε = {{S} S} ε = {{ε} ε} ε = {{}}

Context-Free Grammars

- Shorthand notation: vertical bar for multiple productions

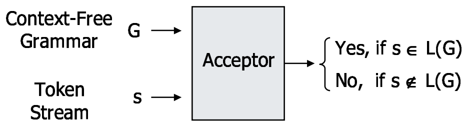
  S → a S a | T

  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

RE is Subset of CFG

- Inductively build a grammar for each regular expression

ε	$S \to \varepsilon$
a	$S \to \mathbf{a}$
$R_1 R_2$	$S \to S_1 S_2$
$R_1 \mid R_2$	$S \to S_1 \mid S_2$
R_1 *	$S \to S_1 S \mid \varepsilon$

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:
  $S \to E + S \mid E$
  $E \to$ number $\mid$ (S)
- Expanded:
  $S \to E + S$
  $S \to E$
  $E \to$ number
  $E \to (S)$

  4 productions
  2 non-terminals (S, E)
  4 terminals: (,), +, number
  start symbol S
- Example accepted input:
  (1 + 2 + (3+4)) + 5

Derivation Example

S → E + S | E
E → number | (S)

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

S → **E** + S → (**S**) + S → (**E** + S)+ S
→ (1 + **S**)+S → (1 + **E** + S)+S
→ (1 + 2 + **S**)+S → (1 + 2 + **E**)+S
→ (1 + 2 + (**S**))+S→ (1 + 2 + (**E** + S))+S
→ (1 + 2 + (3 + **S**))+S
→ (1 + 2 + (3 + **E**))+S
→ (1 + 2+ (3+4))+**S**
→ (1 + 2+ (3+4))+**E**
→ (1 + 2+ (3+4))+5

replacement string
non-terminal being expanded

Constructing a Derivation

- Start from S (start symbol)
- Use productions to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β and γ and for a production

$$A \to \beta$$

a single step of derivation is:

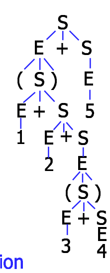
$$\alpha A \gamma \Rightarrow \alpha\beta\gamma$$

(i.e., substitute β for an occurrence of A)

- Example:
  S → E + S
  (**S** + E) + E → (**E + S** + E)+E

Derivation ⇒ Parse Tree

Parse Tree



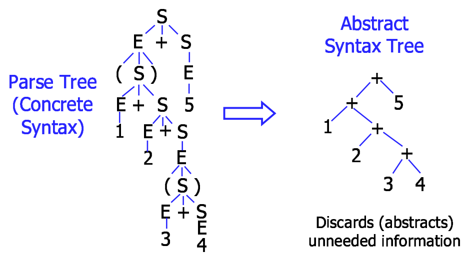
- Parse Tree = tree representation of the derivation
- Leaves of tree are terminals
- Internal nodes: non-terminals
- No information about order of derivation steps

Derivation

S → **E** + S → (**S**) + S → (**E** + S) + S → (1 + **S**)+S → (1 + **E** + S) + S → … → (1 + 2 + (**S**)) + S→ (1 + 2 + (**E** + S))+S → … → (1 + 2 + (3 + **E**))+S → … → (1 + 2+ (3+4))+5

Parse Tree vs. AST

- Parse tree also called “concrete syntax”



Derivation order

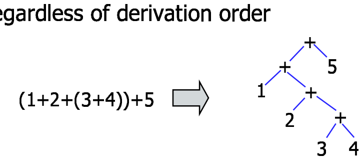
- Can choose to apply productions in any order; select any non-terminal A: $\alpha A \gamma \Rightarrow \alpha\beta\gamma$
- Two standard orders: left- and right-most -- useful for different kinds of automatic parsing
- Leftmost derivation: In the string, find the left-most non-terminal and apply a production to it
  E + S → **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 →(**S**) + S → (**E** + S)+ S → (1 + **S**)+S → (1+**E**+S)+S → (1+2+**S**)+S → (1+2+**E**)+S → (1+2+(**S**))+S → (1+2+(**E**+S))+S → (1+2+(3+**S**))+S → (1+2+(3+**E**))+S → (1+2+(3+4))+**S** → (1+2+(3+4))+**E** → (1+2+(3+4))+5
- Right-most derivation
  S →E+**S** →E+**E** → **E**+5 → (**S**)+5 → (E+**S**)+5 → (E+E+**S**)+5 → (E+E+**E**)+5 → (E+E+(**S**))+5 → (E+E+(E+**S**))+5 → (E+E+(E+**E**))+5 → (E+E+(**E**+4))+5 → (E+**E**+(3+4))+5→ (**E**+2+(3+4))+5 → (1+2+(3+4))+5
- Same parse tree: same productions chosen, diff. order

Ambiguous Grammars

- 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:

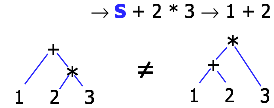
  S → S + S | S * S | number

- Ambiguous grammar = different derivations produce different parse trees

Differing Parse Trees

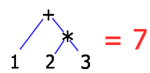
S → S + S | S * S | number

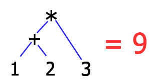
- Consider expression 1 + 2 * 3
- Derivation 1: **S** → **S** + S → 1 + **S** → 1 + **S** * S → 1 + 2 * **S** → 1 + 2 * 3
- Derivation 2: **S** → S * **S** → **S** * 3 → S + **S** * 3 → **S** + 2 * 3 → 1 + 2 * 3



Impact of Ambiguity

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

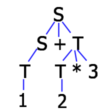




Eliminating Ambiguity

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

S → S + T | T
T → T * **num** | **num**



- T non-terminal enforces precedence
- Left-recursion : left-associativity

CFGs

- Context-free grammars allow concise syntax specification of programming languages
- CFGs specifies how to convert token stream to parse tree (if unambiguous!)
- Read Appel 3.1, 3.2