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

Abstract Syntax Tree (AST)
Parsing Analogy

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

```
  sentence
    subject: I verb: gave indirect object: him
    noun phrase: 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 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
  - 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 = {Q}

Context-Free Grammars

- **Shorthand notation**: vertical bar for multiple productions
  - S → a Sa | 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
  
  Context-free Grammar
  
  $\text{G} \rightarrow \text{Accepter}$
  
  Token Stream
  
  $s \rightarrow \{ \text{Yes, if } s \in L(G) \}$
  
  $\{ \text{No, if } s \notin L(G) \}$

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

RE is Subset of CFG

- Inductively build a grammar for each regular expression
  
  $\varepsilon \rightarrow \varepsilon$
  
  $a \rightarrow a$
  
  $R_1 \rightarrow S_1, S_2$
  
  $R_1 \rightarrow S_1 | S_2$
  
  $R_1 \rightarrow S_1 | \varepsilon$

  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:
  
  $S \rightarrow E + S | E$
  
  $E \rightarrow \text{number } | (S)$

  Expanded:
  
  $S \rightarrow E + S$
  
  $2$ non-terminals ($S$, $E$)
  
  $E \rightarrow \text{number } | S$
  
  $4$ terminals: ( ), +, number
  
  start symbol $S$

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

Derivation Example

$S \rightarrow E + S | E$

$E \rightarrow \text{number } | (S)$

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

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

$\rightarrow (1 + S) + S \rightarrow (E + S) + S$

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

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

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

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

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

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 $\Rightarrow$ Parse Tree

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

$\Rightarrow (1 + 2 + (3 + 4)) + S$
Parse Tree vs. AST

- Parse tree also called "concrete syntax"

```
S
E + S
E

Parse Tree
(Concrete Syntax)
```

Abstract Syntax Tree

```
+ 5
1 2
3 4
```

Derivation order

- Can choose to apply productions in any order; select any non-terminal: $\alpha \gamma \Rightarrow \alpha \beta$
- 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

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

Ambiguous Grammars

- In example grammar, left-most and right-most derivations produced identical parse trees
  
  $$+ \text{ operator associates to right in parse tree regardless of derivation order}$$
  
  $$(1 + 2 + (3 + 4)) + 5 \Rightarrow 5$$

An Ambiguous Grammar

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

$$S \rightarrow S + S | S * S | \text{number}$$

- Consider expression $1 + 2 * 3$
  
  **Derivation 1:**
  
  $$S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S * S \rightarrow 1 + 2 * S \rightarrow 1 + 2 * 3$$
- **Derivation 2:**
  
  $$S \rightarrow S * S \rightarrow S * 3 \rightarrow S + S * 3 \rightarrow S + 2 * 3 \rightarrow 1 + 2 * 3$$
**Impact of Ambiguity**

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

\[
\begin{align*}
    1 & \ast 2 & \ast 3 = 7 \\
    1 & \ast 2 & 3 = 9
\end{align*}
\]

**Eliminating Ambiguity**

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

\[
\begin{align*}
    S & \rightarrow S + T | T \\
    T & \rightarrow T * \text{num} | \text{num}
\end{align*}
\]

\[
\begin{align*}
    S & \rightarrow S + T \\
    T & \rightarrow T * 3
\end{align*}
\]

- 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