Syntax Analysis

Prof. James L. Frankel
Harvard University

Version of 6:06 PM 11-Feb-2025

Context-Free Grammar (CFG)

- terminals — basic symbols from which strings are formed; also called *tokens*
- non-terminals — syntactic variables that denote sets of strings
- start symbol — strings denoted by the start symbol is the language generated by the grammar
- productions — <head> → <body>

<head>	→	<body>
or	*or*	*or*
<left side>	::=	<right side>

Context-Free Grammar for Simple Expressions

- E → E + T | E – T | T
  T → T * F | T / F | F
  F → (E) | **id**
- E, T, and F are *non-terminals*
- E is the *start symbol*
- +, –, *, /, (,), and **id** are *terminals*
- E is an abbreviation for Expression (sometimes referred to as Expr)
- T is an abbreviation for Term
- F is an abbreviation for Factor
- Vertical bar (|) indicates alternate right sides and is equivalent to writing multiple productions, as shown below for F → (E) | **id**
  - F → (E)
  - F → **id**
- The right arrow (→) can be read as "can have the form of"

CFG Notational Conventions (1 of 3)

- Terminals
  Lowercase letters early in alphabet (a, b, c, …)
  Operators
  Punctuation
  Digits
  Boldface string denoting terminal symbols (such as **id** and **if**)
- Non-terminals
  Uppercase letters early in alphabet (A, B, C, …)
  S is usually the start symbol
  Lowercase italic names (*expr*, *stmt*, …)
  E, T, F

CFG Notational Conventions (2 of 3)

• Grammar symbols (either non-terminals or terminals)	Uppercase letters late in the alphabet (X, Y, Z)
• Possibly empty strings of terminals	Lowercase letters late in the alphabet (u, v, …, z)
• Possibly empty strings of grammar symbols	Lowercase Greek letters (α, β, γ, …)

CFG Notational Conventions (3 of 3)

- Set of productions with a common head: $A \to \alpha_1$, $A \to \alpha_2$, …, $A \to \alpha_k$ are called A-productions and may be rewritten as
  $A \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$
- Unless otherwise specified, the head of the first production is the start symbol

CFG Derivation

- *Derivation* is the process of using productions as rewriting rules

- If $A \rightarrow \gamma$ is a production, then

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

  where $\Rightarrow$ means "derives in one step"

- $\Rightarrow^{*}$ means "derives in zero or more steps"
- $\Rightarrow^{+}$ means "derives in one or more steps"

Leftmost and Rightmost Derivations

- In *leftmost derivations*, the leftmost non-terminal in each sentential is always chosen.
- This is denoted by

$$\alpha \Rightarrow_{lm} \beta$$

- In *rightmost derivations*, the rightmost non-terminal in each sentential is always chosen.
- These are also called *canonical derivations*
- This is denoted by

$$\alpha \Rightarrow_{rm} \beta$$

Leftmost & Rightmost Derivation of –(**id**+**id**)

- Given the grammar: $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \textbf{id}$, derive –(**id**+**id**)

- $E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E) \Rightarrow_{lm} -(\textbf{id} + E) \Rightarrow_{lm} -(\textbf{id} + \textbf{id})$

- $E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E) \Rightarrow_{rm} -(E + \textbf{id}) \Rightarrow_{rm} -(\textbf{id} + \textbf{id})$

Ambiguity in Leftmost Derivation of **id+id*id**

- Given the grammar: $E \to E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$,
  derive **id+id*id**

- Two distinct leftmost derivations exist:

- $E \Rightarrow_{lm} E + E \Rightarrow_{lm} \mathbf{id} + E \Rightarrow_{lm} \mathbf{id} + E * E \Rightarrow_{lm} \mathbf{id} + \mathbf{id} * E \Rightarrow_{lm} \mathbf{id} + \mathbf{id} * \mathbf{id}$
- $E \Rightarrow_{lm} E * E \Rightarrow_{lm} E + E * E \Rightarrow_{lm} \mathbf{id} + E * E \Rightarrow_{lm} \mathbf{id} + \mathbf{id} * E \Rightarrow_{lm} \mathbf{id} + \mathbf{id} * \mathbf{id}$

- Therefore, this grammar is ambiguous

Translation of a regex into a CFG

- The regex:

  ba*bba

  and the context-free grammar:

  $A_0 \rightarrow bA_1$
  $A_1 \rightarrow aA_1 \mid A_2$
  $A_2 \rightarrow bba$

  derive the same language

Balanced parentheses

- The context-free grammar:

  $$A \rightarrow (A) A \mid \varepsilon$$

  derives any number of balanced parentheses

- This cannot be derived using a regex
- Colloquially, we say that “a finite automata cannot count”

Ambiguous **else** matching

- stmt → **if** expr **then** stmt |
  if expr **then** stmt **else** stmt |
  other
- The above grammar is ambiguous

- **if** E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3
- The above sentence's derivation is ***un***ambiguous

- **if** E_1 **then if** E_2 **then** S_1 **else** S_2
- The above sentence's derivation is ambiguous
- With which **then** does the **else** match?

Rewritten **if-then-else** grammar

- stmt → matched_stmt | open_stmt
- matched_stmt → **if** expr **then** matched_stmt **else** matched_stmt | other
- open_stmt → **if** expr **then** stmt | **if** expr **then** matched_stmt **else** open_stmt

- Idea: stmt between **then** and **else** cannot end with an unmatched (or open) **then**
- The above grammar is unambiguous

Elimination of Left Recursion

- $A \rightarrow A\alpha \mid \beta$

  can be rewritten as

  $A \rightarrow \beta A'$
  $A' \rightarrow \alpha A' \mid \varepsilon$

Left Factoring

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

  can be rewritten as

  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1 \mid \beta_2$

- This refactoring defers the decision so it can be made when the input symbol is available
  - Useful in predictive, or top-down, or recursive descent parsers

Top-Down Parsing

- Starting with the left-recursive grammar:

  E → E + T | T
  T → T * F | F
  F → (E) | **id**

- After applying the elimination of left recursion transformation we have:

  E → T E'
  E' → + T E' | ε
  T → F T'
  T' → * F T'| ε
  F → (E) | **id**

- This is suitable for a top-down parser or for producing a leftmost derivation

Example of Recursive-Descent Parser

- Go over recursiveDescentParser.c

- Show the parse trees of

  1 – 2 * 3
  and
  1 – 2 + 3

  using the grammars from before and after applying the “elimination of left recursion” transformation