Compilers

CS414-2015-02

Context-Free Grammars

David Galles

Department of Computer Science
University of San Francisco

02-0: Parsing

- Once we have broken an input file into a sequence of tokens, the next step is to determine if that sequence of tokens forms a syntactically correct program – parsing
- We will use a tool to create a parser – just like we used lex to create a parser
- We need a way to describe syntactically correct programs
  - Context-Free Grammars

02-1: Context-Free Grammars

- Set of Terminals (tokens)
- Set of Non-Terminals
- Set of Rules, each of the form: $<$Non-Terminal$>$ $\rightarrow$ $<$Terminals & Non-Terminals$>$
- Special Non-Terminal – Initial Symbol

02-2: Generating Strings with CFGs

- Start with the initial symbol
- Repeat:
  - Pick any non-terminal in the string
  - Replace that non-terminal with the right-hand side of some rule that has that non-terminal as a left-hand side

  Until all elements in the string are terminals

02-3: CFG Example

$$E \longrightarrow E + E$$
$$E \longrightarrow E - E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow E/E$$
$$E \longrightarrow \textsf{num}$$

02-4: CFG Example

$E \longrightarrow E + E$

$E \longrightarrow E - E$

$E \longrightarrow E * E$

$E \longrightarrow E/E$

$E \longrightarrow$ num

E

02-5: CFG Example

$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E/E$
$E \rightarrow \text{num}$

$E \quad \Rightarrow E + E$

02-6: CFG Example

$$E \longrightarrow E + E$$
$$E \longrightarrow E - E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow E/E$$
$$E \longrightarrow \text{num}$$

$$\begin{aligned} E \quad &\Rightarrow E + E \\ &\Rightarrow E * E + E \end{aligned}$$

02-7: CFG Example

$$E \longrightarrow E + E$$
$$E \longrightarrow E - E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow E/E$$
$$E \longrightarrow \text{num}$$

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E * E + E \\ &\Rightarrow \text{num} * E + E \end{aligned}$$

02-8: CFG Example

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E/E$

$E \rightarrow$ num

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E * E + E \\
&\Rightarrow \text{num} * E + E \\
&\Rightarrow \text{num} * \text{num} + E
\end{aligned}
$$

02-9: CFG Example

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$
$$E \rightarrow E/E$$
$$E \rightarrow \text{num}$$

$$\begin{aligned} E & \Rightarrow E + E \\ & \Rightarrow E * E + E \\ & \Rightarrow \text{num} * E + E \\ & \Rightarrow \text{num} * \text{num} + E \\ & \Rightarrow \text{num} * \text{num} + \text{num} \end{aligned}$$

02-10: CFG Example

$S \rightarrow NP \quad V \quad NP$

$NP \rightarrow$ the N

$N \rightarrow$ boy

$N \rightarrow$ ball

$N \rightarrow$ window

$V \rightarrow$ threw

$V \rightarrow$ broke

$S \Rightarrow NP \quad V \quad NP$

$\Rightarrow$ the $N \quad V \quad NP$

$\Rightarrow$ the boy $V \quad NP$

$\Rightarrow$ the boy threw NP

$\Rightarrow$ the boy threw the $N \Rightarrow$ the boy threw the ball

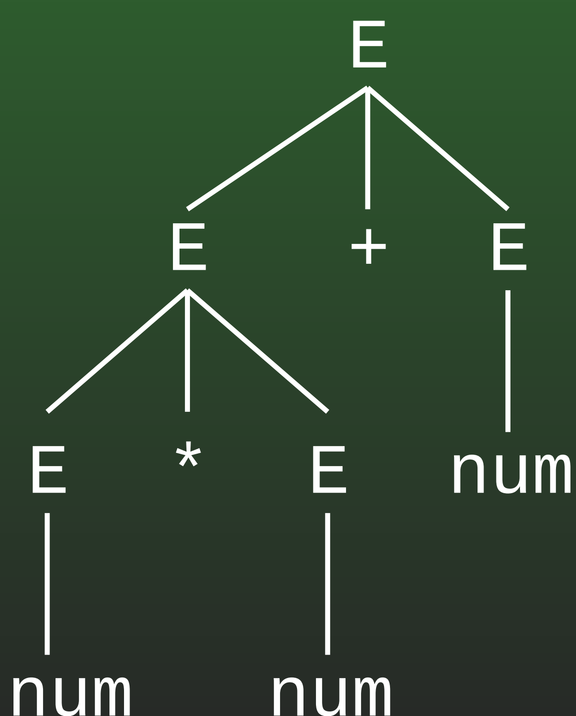
02-11: Derivations

- A derivation is a description of how a string is generated from a grammar
- A *Leftmost* derivation always picks the leftmost non-terminal to replace
- A *Rightmost* derivation always picks the rightmost non-terminal to replace
- Some derivations are neither rightmost nor leftmost

02-12: Parse Trees

A Parse Tree is a graphical representation of a derivation

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E * E + E \\ &\Rightarrow \text{num} * E + E \Rightarrow \text{num} * \text{num} + E \\ &\Rightarrow \text{num} * \text{num} + \text{num} \end{aligned}$$



02-13: Parse Trees & Derivations

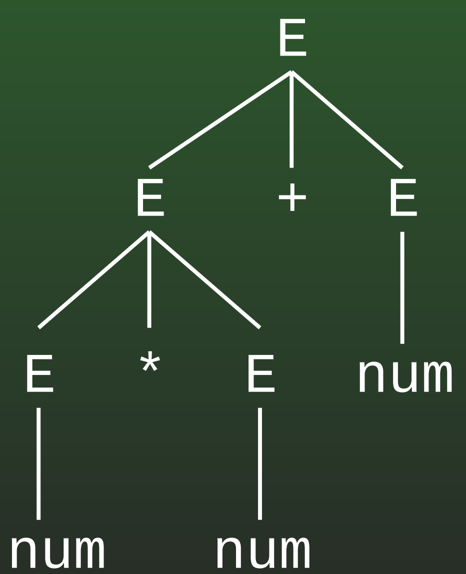
- A parse tree can represent > 1 different derivation (rightmost and leftmost, for example)
- There is a 1-1 correspondence between leftmost derivations and parse trees

02-14: Parse Trees & Meaning

- A parse tree represents some of the “meaning” of a string.
- For instance: 3 * 4 + 5
  - (3 * 4) + 5
  - 3 * (4 + 5)

02-15: Parse Trees & Meaning

- A parse tree represents some of the "meaning" of a string.
- For instance: 3 * 4 + 5
    - (3 * 4) + 5
    - ~~3 * (4 + 5)~~



02-16: Ambiguous Grammars

- A Grammar is *ambiguous* if there is at least one string with more than one parse tree
- The expression grammar we've seen so far is ambiguous

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$
$$E \rightarrow E / E$$
$$E \rightarrow \text{num}$$

02-17: Removing Ambiguity

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$
$$E \rightarrow E/E$$
$$E \rightarrow \text{num}$$

Step I: Multiplication over Addition:
(3 * 4) + 5 vs. 3 * (4 + 5)

02-18: Removing Ambiguity

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow T$$
$$T \rightarrow T * T$$
$$T \rightarrow T / T$$
$$T \rightarrow F$$
$$F \rightarrow \text{num}$$

Step II: Mandating Left-Associativity
(3 + 4) + 5 vs. 3 + (4 + 5) and

(3 - 4) - 5 vs. 3 - (4 - 5)

02-19: Adding Parentheses

$$E \to E + T$$
$$E \to E - T$$
$$E \to T$$
$$T \to T * F$$
$$T \to T/F$$
$$T \to F$$
$$F \to \text{num}$$

Allowing parenthesized expressions: (3 + 4) * 5

02-20: Expression Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow$ num

$F \rightarrow (E)$

02-21: CFG for Statements

- Expressions: id, num
- Function calls: id($<$input params$>$)
  - $<$input params$>$ are expressions separated by commas
- Block Statements { $<$ list of statements $>$ }
- While statements (C syntax)

All statements are terminated by a semi-colon ;

02-22: CFG for Statements

$S \rightarrow \text{id}(P);$

$S \rightarrow \{L\}$

$S \rightarrow \text{while } (E)\ S$

$E \rightarrow \text{id} \mid \text{num}$

$P \rightarrow \epsilon$

$P \rightarrow EP'$

$P' \rightarrow \epsilon$

$P' \rightarrow ,\ EP'$

$L \rightarrow \epsilon$

$L \rightarrow SL$

02-23: Bakus Naur Form

- Another term for Context-Free grammars is Bakus Naur Form, or BNF
- We will use CFG and BNF interchangeably for this class

02-24: Extended Bakus Naur Form

- Use regular expression notation (*, +, |, ?) in BNF (CFG) rules

  (1) $S \rightarrow$ { B }

  (2) $S \rightarrow$ print (id)

  (3) $B \rightarrow S$; C

  (4) $C \rightarrow S$; C

  (5) $C \rightarrow \epsilon$

- Rules (3) - (5) describe 1 or more statements, terminated by ;

02-25: Extended Bakus Naur Form

- Use regular expression notation (*, +, $|$, ?) in BNF (CFG) rules

  (1) $S \rightarrow$ { B }

  (2) $S \rightarrow$ print "(" id ")"

  (3) $B \rightarrow$ (S;)+

- Rules (3) describes 1 or more statements, terminated by ;

Extended Bakus Naur Form

- Pascal for statements:

(1) $S \rightarrow$ for id := E to E do S
(2) $S \rightarrow$ for id := E downto E do S

02-27: Extended Bakus Naur Form

- Pascal for statements:

(1) $S \rightarrow$ for id := E (to $|$ downto) E do S

- Why this is useful (other than just reducing typing) will be seen when we generate parsers