CFG1: Intro to CFGs

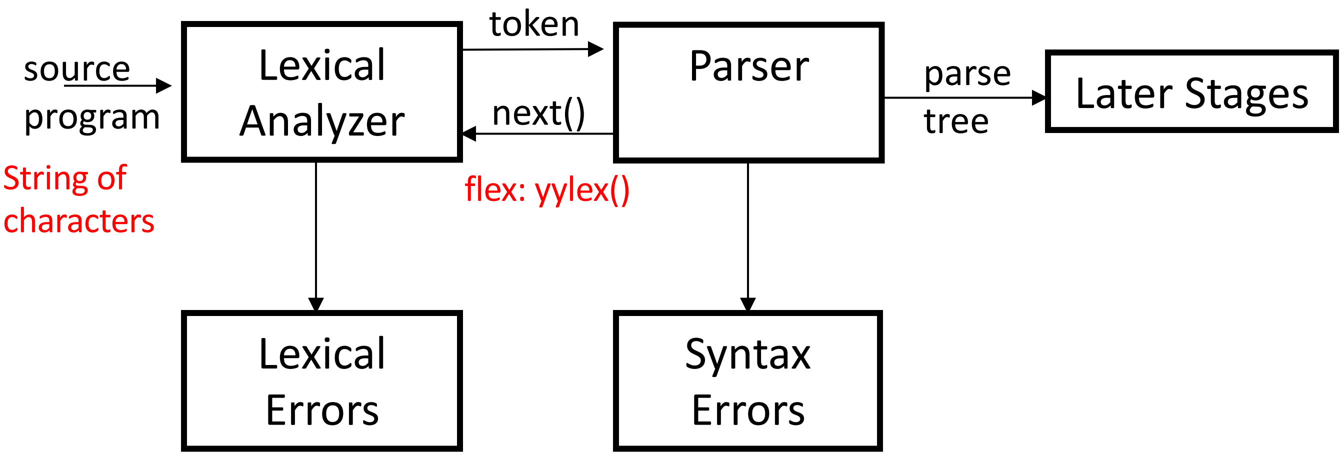
Context-Free Grammars

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Parsing



source program
String of characters
Lexical Analyzer
token
next()
flex: yylex()
Parser
parse tree
Later Stages
Lexical Errors
Syntax Errors

Parsing

- Every possible token sequence is not a valid program
- Parser distinguishes between valid and invalid programs
- We need
  - A language for describing valid sequence of tokens
  - A method for distinguishing valid from invalid programs
  - Provide the program structure for a valid token sequence

Context-free Grammars (CFGs)

- Programming languages have recursive structure
- An EXP is …

```
if EXP then
      EXP
else
      EXP
```

```
while EXP do
      EXP
end
```

$(_{if}$ if EXP then
  $(_{while}$ while EXP do
    $(_{if}$ if EXP then
      $(_{while}$ while EXP do
        EXP
      $)_{while}$ end
    $)_{if}$ else
        EXP
  $)_{while}$ end
$)_{if}$ else
    EXP

- Context Free Grammars are natural notation for the recursive structures we find in programming languages
- Finite state automata cannot handle nested parentheses

Context-free Grammars (CFGs)

- A CFG consists of
  - A set of terminals: T (input symbols)
  - A set on non-terminals: N
  - A start symbol: $S \in N$
  - A set of rules/productions: $\overbrace{X}^{\text{LHS}} \rightarrow \overbrace{Y_1 \ldots Y_n}^{\text{RHS}}$

    $X \in N$

    $Y_i \in N \cup T \cup \{\varepsilon\}$

Rule application:

Replace LHS with RHS

Context-free Grammars (CFGs)

L = $\{ (^{i})^{i} \mid i \geq 0 \}$

Q: Does the string "()(())" belong to this language?

Q: Modify this CFG to use the alphabet { '(', ')', '{', '}', '[', ']' } where opening and closing parentheses must of the same type. So "({[()]})" is valid but "(}" is invalid.

CFG Rules:

Non-deterministic choice of S rule

S → '(' S ')'

S → ε

N = {S}

T = { '(', ')' }

Context-free Grammars (CFGs)

1. Begin with string that has only start symbol S
2. Replace any non-terminal X in the string by the right-hand side of some production $X \rightarrow Y_1 \ldots Y_n$
3. Repeat (2) until there is no non-terminals

r1: $S \rightarrow (\, S \,)$
r2: $S \rightarrow \varepsilon$

$$S \Rightarrow^{r1} (\, S \,) \Rightarrow^{r1} (\, (\, S \,) \,) \Rightarrow^{r2} (\, (\quad) \,)$$

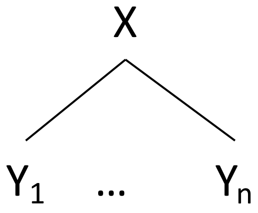
Non-deterministic choice of S rule

Derivation and Parse Tree

- A derivation is a sequence of rule applications

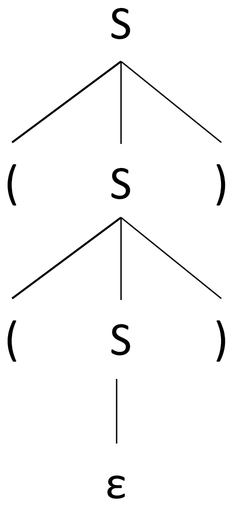
$$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \ldots$$

- A derivation can be drawn as a parse tree
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X



Derivation and Parse Tree

$S \Rightarrow^{r1} (S) \Rightarrow^{r1} ((S)) \Rightarrow^{r2} ((\ \ ))$



Closure of $\Rightarrow$

$S \Rightarrow^{*} (( ))$

Q: Write down the derivation and parse tree for the input string "({[]})" using your grammar for question on slide 6

Language of CFGs

Let G be a context free grammar with start symbol S, and terminals T

The language L(G) of G is:

$$\{\alpha_1 \dots \alpha_n | \forall_i \; \alpha_i \in T \; and \; S \Rightarrow^* \alpha_1 \dots \alpha_n\}$$

r1: S → (S)
r2: S → ε

L(G) = {ε, (), (()), ((())), ...}

Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- $E \rightarrow - E$
- $E \rightarrow \textbf{id}$

Derivation for
id + id * id

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow id$

$E \Rightarrow E + E$

$\Rightarrow \mathbf{id} + E$

$\Rightarrow \mathbf{id} + E * E$

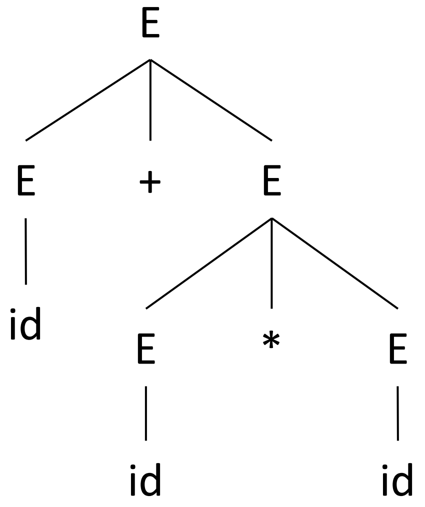
$\Rightarrow \mathbf{id} + \mathbf{id} * E$

$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

Notation:
$E \Rightarrow^* id + id * id$

Leaf nodes: terminals

Interior nodes: non-terminals



Leftmost derivation for **id + id * id**

Parse tree disambiguates operator precedence:

(id+id)*id vs id+(id*id)

$E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to - E$
$E \to id$

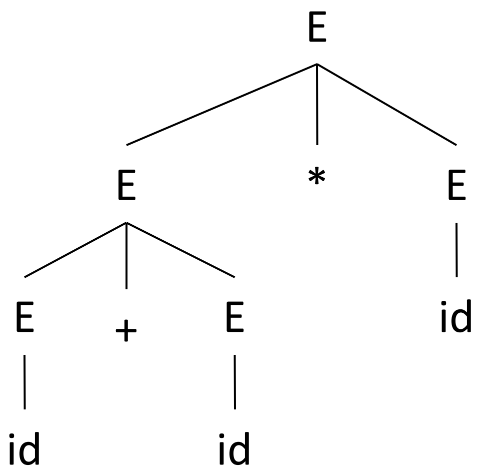
$E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow \mathbf{id} + E * E$

$\Rightarrow \mathbf{id} + \mathbf{id} * E$

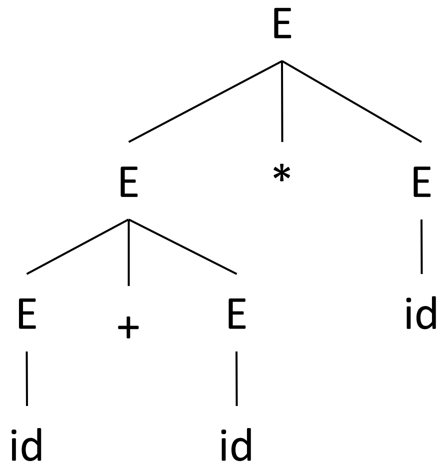
$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$



Rightmost derivation for **id + id * id**

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow id$

$E \Rightarrow E * E$
$\Rightarrow E * \mathbf{id}$
$\Rightarrow E + E * \mathbf{id}$
$\Rightarrow E + \mathbf{id} * \mathbf{id}$
$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$



Q: Write down the rightmost derivation for same grammar and input to get the parse tree in slide 12

Rightmost vs. Leftmost Derivation

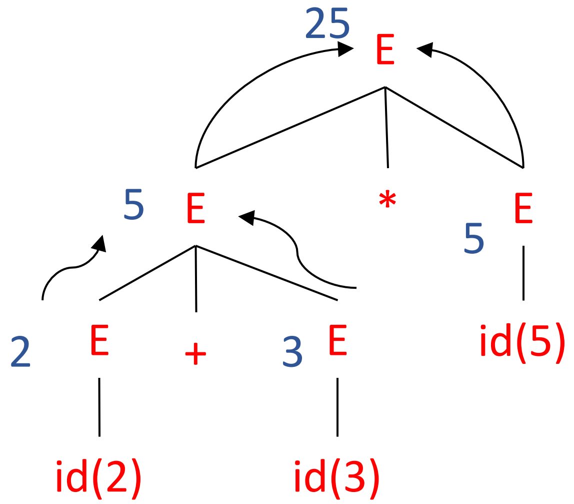
- Rightmost and leftmost derivations have the same parse tree
  - Every parse tree has a *rightmost derivation*
  - And every parse tree has an equivalent *leftmost derivation*
  - *Leftmost* / *Rightmost derivations* are important in resolving ambiguity

Writing a CFG for a programming language

- First write (or read) a reference grammar of what you want to be valid programs
- For now, we only worry about the structure, so the reference grammar might choose to over-generate in certain cases
  - e.g. `bool x = 20;`
- Convert the reference grammar to a CFG
- Use actions for each CFG rule to produce the output

Actions in a CFG: Arithmetic Expressions

- E → E + E { $$ = $1 + $3 }
- E → E * E { $$ = $1 * $3 }
- E → (E) { $$ = $2 }
- E → - E { $$ = -1 * $2 }
- E → **id** { $$ = $1 }



Q: Draw the parse tree and calculate the output value using the above CFG & actions for - (2+3)

CFG Notation

- Normal CFG notation

  $E \to E * E$

  $E \to E + E$

- Backus Naur notation

  E : E * E | E + E ;

  (an or-list of right-hand sides)

  Also:

  E = E "*" E | E "+" E .