Lecture 2. Context Free Grammar

January 22, 2021

Overview

- what is a grammar
- what is a context free grammar (cfg)
- relations of strings and grammar
  - derivation: derive a string from a grammar
  - parsing: parse a string to determine if it comforms to the gramma
- parse tree and semantics
- ambiguity
- design grammar

What is a grammar

Specify Patterns in String

a program is a string that follows certain "patterns" and "rules"

the rules can be specified by:

- informal grammar – English description
- formal grammar
  1. what are the atoms? (**terminals**, **lexeme**, **token**), e.g., int a = 0; lexeme: int, a, =, 0, ; terminals (given in the grammar) identifier, numbers; tokens (terminal + link to the symbol table, for implementing compiler or interpreter)
  2. how to compose them to form a sentence?
  3. others, e.g., regular expressions

a program is **syntactically correct** (**valid**) if it follows the grammar of the language (rules) in which it is written

Grammar

- A language is the set of strings
- Rules to define a string in the language
- Every programming language has a grammar describing the syntax (the order or arrangement of terminals need to follow certain patterns, rules)
- Bases for parsing – given a program (string) and a grammar, validate the grammatical correctness of the string, i.e., if the string follows the rules of the language

Connection to Formal Language and Formal Grammars

Language types are determined by their grammars; the grammar types are determined by the patterns of their production rules

- Type-3 : regular language – a*b*
- Type-2 : context free languages – typical programs
- Type-1 : context-sensitive languages – $(a^n b^n c^n)$
- Type-0: recursively enumerable

Chomsky Hierarchy. COM S 331 Theory of Computing.

Formal Grammars Define Formal Languages

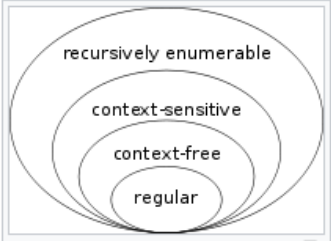
Chomsky Hierarchy

The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.

Grammar	Languages	Automaton	Production rules (constraints)*	Examples[3]
Type-0	Recursively enumerable	Turing machine	$\alpha A\beta \to \gamma$	$L = \{w \mid w$ describes a terminating Turing machine$\}$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A\beta \to \alpha\gamma\beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \to \alpha$	$L = \{a^n b^n \mid n > 0\}$
Type-3	Regular	Finite state automaton	$A \to \mathrm{a}$ and $A \to \mathrm{a}B$	$L = \{a^n \mid n \geq 0\}$

* Meaning of symbols:

- a = terminal
- A, B = non-terminal
- α, β, γ = string of terminals and/or non-terminals
  - α, β = maybe empty
  - γ = never empty



Set inclusions described by the Chomsky hierarchy

Context Free Grammar (CFG)

1. formal definitions
2. example
3. notations
4. string and grammar: derivation, parsing, design grammars

Context Free Grammar (CFG)

A CFG contains:

- A set of terminals or atoms in the language
- A set of non-terminals
- A set of (production rules) which describes how a non-terminal can be expanded/rewritten to a sequence of terminals and non-terminals

Context Free Grammar (CFG)

A CFG is a tuple $G = (\Sigma, V, S, P)$, where

- σ is a set of terminals
- V is a set of non-terminals such that $\Sigma \cap V = \emptyset$
- $S \in V$ is a start non-terminal
- P is a set of product rules, each of the form: $X \rightarrow \omega$, such that $X \in V$ and $\omega \in (\Sigma \cup V)^+$

Context Free Grammar (CFG): Example

G $= (\Sigma, V, S, P)$, where

- $\Sigma = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .$
- $V =$ unknown, part, digit, sign
- $S =$ unknown
- P:

  unknown $\rightarrow$ sign part . part|sign part
  sign $\rightarrow +|-$
  part $\rightarrow$ digit|digit part
  digit $\rightarrow 0|1|2|3|4|5|6|7|8|9$

What language is *unknown*?

Notations for Production Rules

- left hand side (LHS)$\rightarrow$ right hand side (RHS)
- if not specified, assume LHS of first production is the start symbol
- productions with the same LHS are combined with $|$
- Backus-Naur Form (BNF) production rules, e.g., $\langle A \rangle := \langle B \rangle$ "c" $\langle D \rangle$ for $A \rightarrow BcD$, by John Backus, Peter Naur, used for describing the grammar of Algol in 1962
- you will see slightly different notations in different systems

Strings and grammar

- L(G) is the language defined by grammar G:
  $L(G) = \{s \in \Sigma^* | S \Rightarrow^+ s\}$
  S is the start symbol of the grammar
  Σ is the set of terminals for that grammar
- L(G) contains all strings that can be derived from the start symbol via one or more derivations
- a program s is a string over a set of tokens (keywords, identifies, numbers, operators ...). s is said to be syntactically correct if and only if $s \in L(G)$, L(G) is the programming language in which s is written, and G is its grammar
- if $s \in L(G)$, we say that the grammar **accepts** the string

Strings and Grammar: generating strings from grammar

Rewriting/derivation: apply a sequence of production rules starting at the start symbol, each application is called a **rewrite**

Given grammar rules: $S \rightarrow 0S|1S|\epsilon$, generating a string 011 from G

Strings and Grammar: generating strings from grammar(contd.)

Rewriting/derivation: apply a sequence of production rules starting at the start symbol, each application is called a **rewrite**

Given grammar rules: $S \rightarrow 0S|1S|\epsilon$, generating a string 011 from G

Derivation:

$$\begin{aligned} S &\Rightarrow 0S \\ &\Rightarrow 0S \\ &\Rightarrow 01S \\ &\Rightarrow 011S \\ &\Rightarrow 011 \end{aligned}$$

Is ϵ a terminal?

Derivation Notations

$\Rightarrow$ indicates a derivation of one step

$\Rightarrow^+$ indicates a derivation of one or more steps

$\Rightarrow^*$ indicates a derivation of zero or more steps

- Example
  - $S \rightarrow 0S \mid 1S \mid \varepsilon$
- For the string 010
  - $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 010$
  - $S \Rightarrow^+ 010$
  - $010 \Rightarrow^* 010$

In-class exercises: derivation

(1) Given grammar rules: $S \rightarrow (S)|\epsilon$, can you generate (), (()), ()()?

(2) Given grammar: $\mathrm{G} = (\Sigma, V, S, P)$, where

- $\Sigma = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .$
- $S =$ unknown
- P:
  unknown $\rightarrow$ sign part . part|sign part
  sign $\rightarrow +|-$
  part $\rightarrow$ digit|digit part
  digit $\rightarrow 0|1|2|3|4|5|6|7|8|9$

What strings can *unknown* generate?

Leftmost and Rightmost Derivations

- Leftmost derivation: At each derivation point, the leftmost non-terminal is expanded
- Rightmost derivation: At each derivation point, the rightmost non-terminal is expanded

Leftmost and Rightmost Derivations

S is a start symbol
digit→ 0|1|2|3|4|5|...|9
part→ digit|digit part
S→ part|S+S |S-S|S*S|S/S

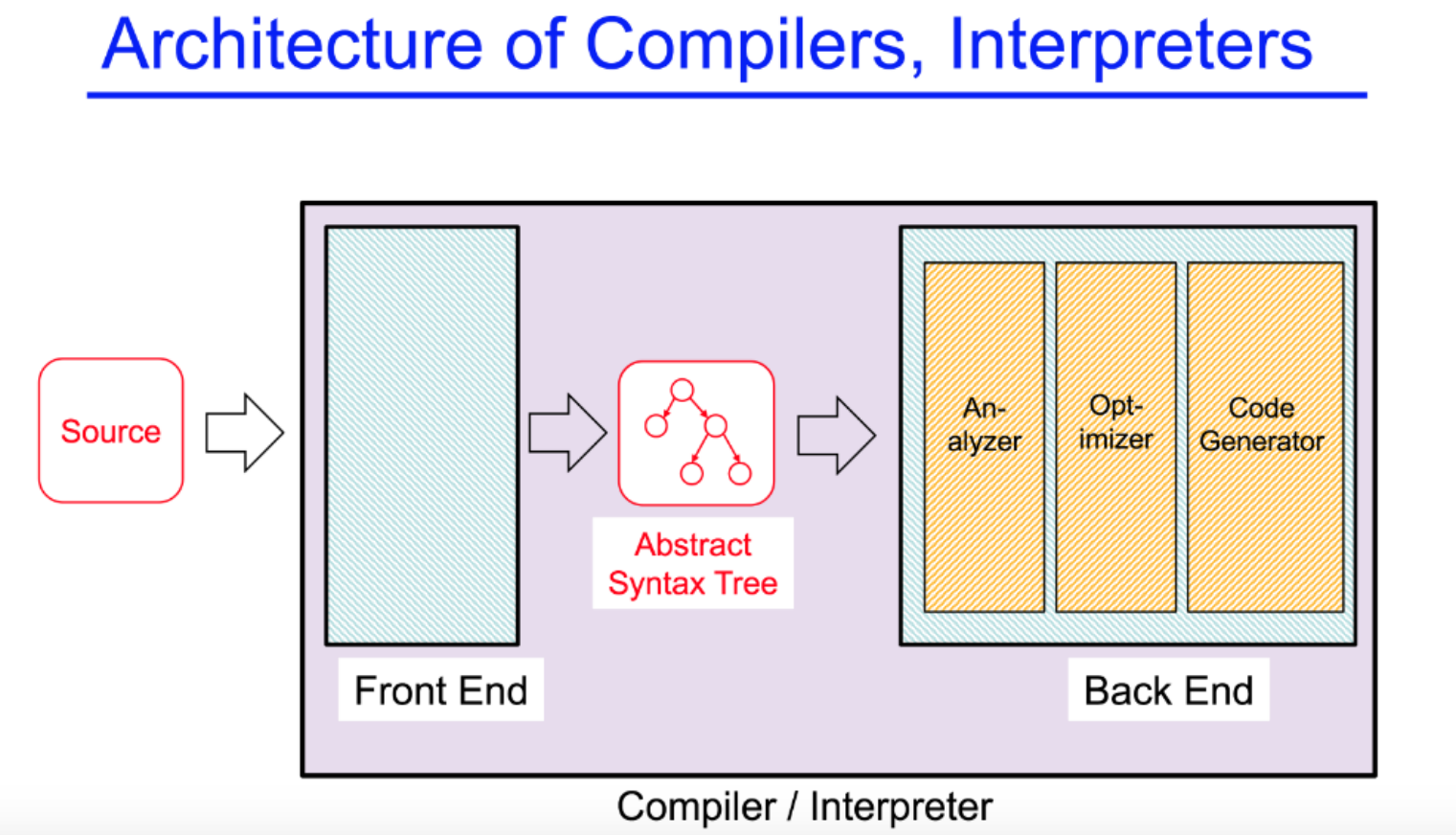
find the derivation for 1+2+3

Leftmost and Rightmost Derivations

Derivation of 1 + 2 + 3

Leftmost	Rightmost
S ⇒ S + S	S ⇒ S + S
⇒ part + S	⇒ S + part
⇒ digit + S	⇒ S + digit
⇒ 1 + S	⇒ S + 3
⇒ 1 + S + S	⇒ S + S + 3
⇒ 1 + part + S	⇒ S + part + 3
⇒ 1 + digit + S	⇒ S + digit + 3
⇒ 1 + 2 + S	⇒ S + 2 + 3
⇒ 1 + 2 + part	⇒ part + 2 + 3
⇒ 1 + 2 + digit	⇒ digit + 2 + 3
⇒ 1 + 2 + 3	⇒ 1 + 2 + 3

Parsing

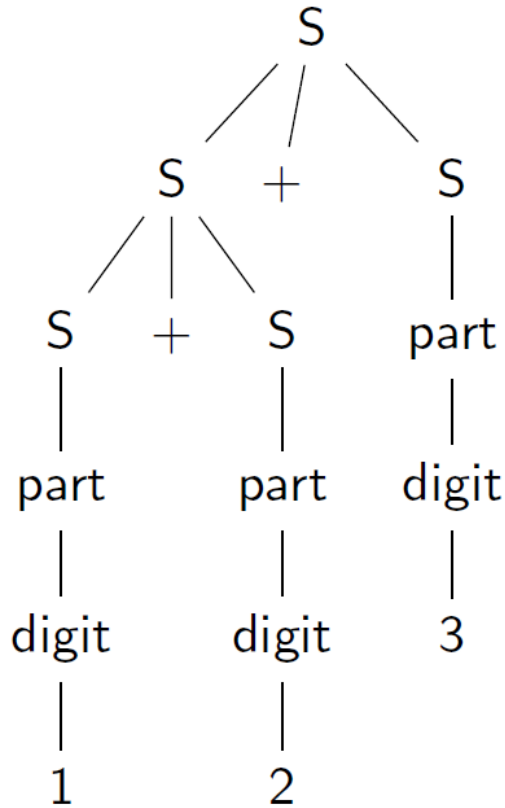

Architecture of Compilers, Interpreters
Source
Front End
Abstract
Syntax Tree
An-
alyzer
Opt-
imizer
Code
Generator
Back End
Compiler / Interpreter

Parsing

- **Parsing**: discovering the derivation and constructing a tree (called a **parse tree**) based on the derivation
- a parse tree shows the structure of an expressions it corresponds to a grammar
- Example
  digit→ 0|1|2|3|4|5|...|9
  part→ digit|digit part
  S→ part|S+S |S-S|S*S|S/S

  find the derivation for 1+2+3

Parse Tree



The same parse tree can be generated by left-most and right-most derivation.

Parse Tree

A parse tree results from the derivation sequences

- a terminal is a leaf node
- each node in the tree is either a terminal or non-terminal in the production rule
- each edge in the tree from a non-terminal results from the application of production on the non-terminal
- application of production rule always result in new nodes in the tree

In-class exercise: constructing parse tree

Grammar: $E \rightarrow a|b|c|d|E + E|E - E|E * E|(E)$ construct parse tree for

a, a*c, c*(b+d)

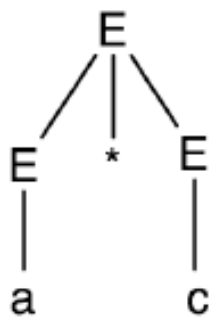
Parse Trees for Expressions

- A parse tree shows the structure of an expression as it corresponds to a grammar

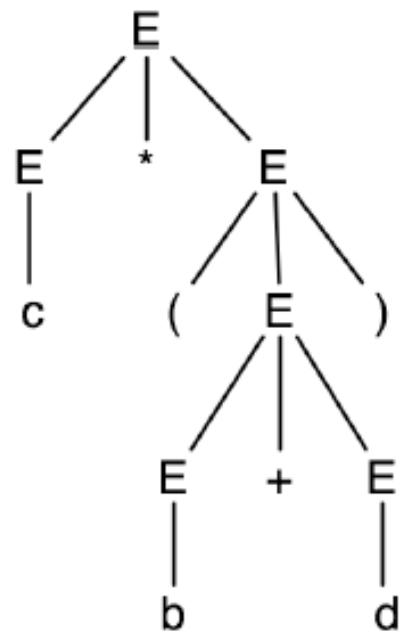
$$E \rightarrow a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E)$$

a

a*c



c*(b+d)



CMSC 330 Spring 2018

Practice

$E \rightarrow a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E)$

Make a parse tree for…

- a*b
- a+(b-c)
- d*(d+b)-a
- (a+b)*(c-d)
- a+(b-c)*d

Computing Values Using Parse Tree: Evaluating Semantics

- We call it *evaluate an expression* – get an value from the expression
- Semantics: meaning of a syntactically correct sentence
- How to automatically get a value from the string
  - Classify the terminals into operands and operator classes
  - Associate meaning with each operand.
  - Associate meaning with the application of operator(s) on operand(s).
  - Evaluate the semantics using parse tree

Computing Values Using Parse Tree: Evaluating Semantics

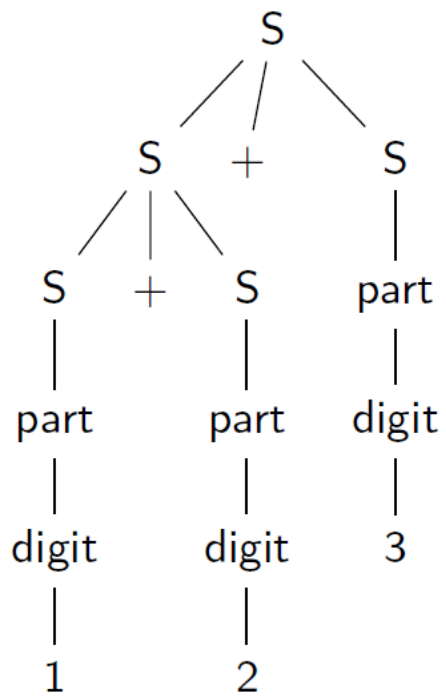
- Start from the leaf-nodes to create the operand and find their meanings.
- Apply the operators on the generated operands to obtain the meaning of the application of the operators.
- Continue until you reach the root-node.

Evaluating Semantics: an example

Given a grammar

digit→ 0|1|2|3|4|5|...|9
part→ digit|digit part
S→ part|S+S |S-S|S*S|S/S

Given a string 1+2+3



sem(+, sem(+, sem(1), sem(2)), sem(3))

Evaluating Semantics: an example continued

- Meaning of a number n can be n points.
- Meaning of Operators :+, -, *
- Follow the evaluation order given by
  sem(+, sem(+, sem(1), sem(2)), sem(3))
  to compute the final value

Ambiguity

A grammar is **ambiguous** if there exists a string that has at least two distinct parse trees

Given a grammar

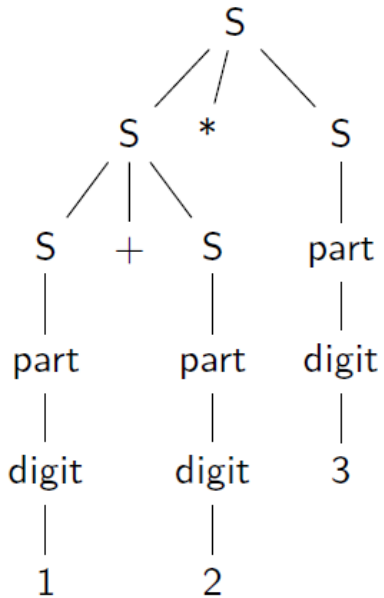
digit→ 0|1|2|3|4|5|...|9
part→ digit|digit part
S→ part|S+S |S-S|S*S|S/S

*Draw a parse tree for 1+2*3*

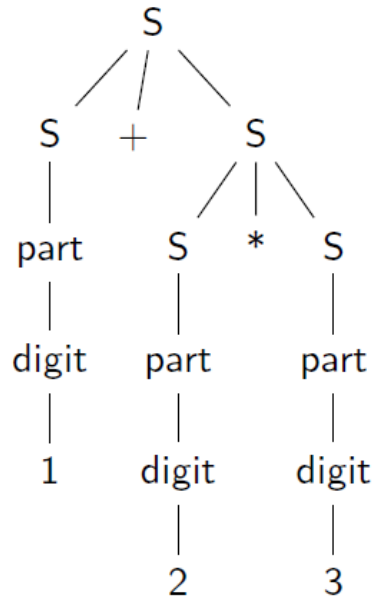
Ambiguity

- Ambiguity in Grammar can result in generating different values
- Ambiguity in Grammar can result in incorrect syntax-directed translation

Ambiguity



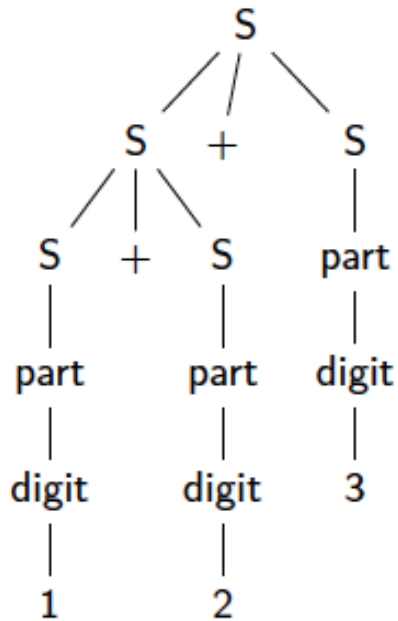
sem(*, sem(+, sem(1), sem(2)), sem(3))

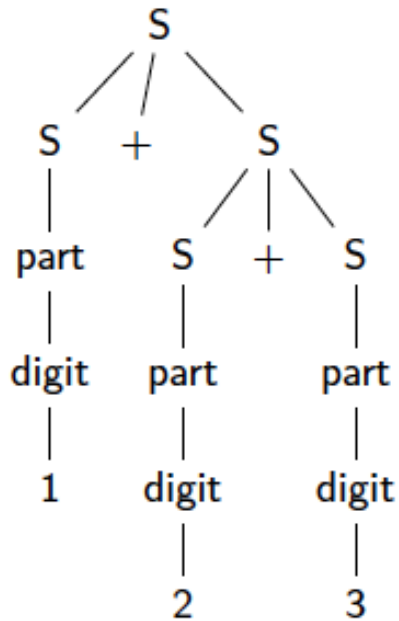


sem(+, sem(1), sem(*, sem(2), sem(3)))

Ambiguity

Parse Tree for 1 + 2 + 3





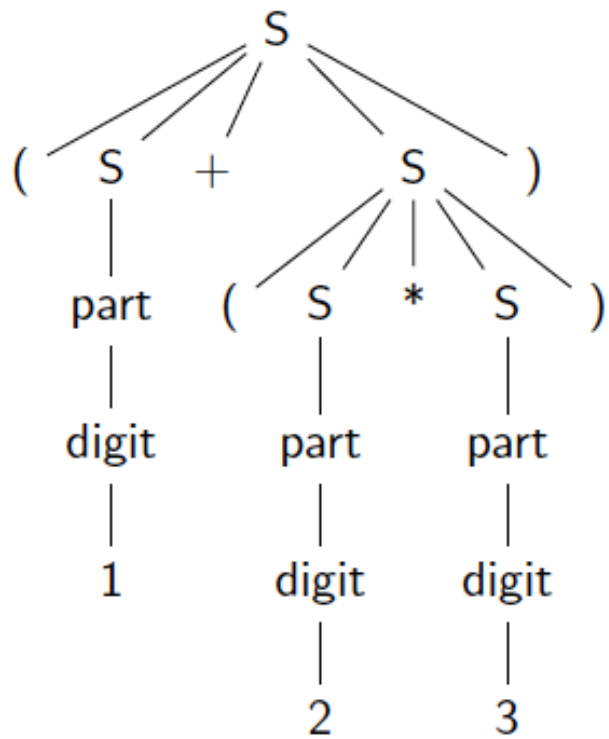
Remove Ambiguity

- Add delimiters (e.g., parenthesis) – modify grammars, add terminals of deliminator
- Add operator precedence and associativity – modify grammars, giving additional rules beyond grammar

Delimiters

$$\texttt{digit} \longrightarrow 0 \mid 1 \mid \ldots \mid 9$$
$$\texttt{part} \longrightarrow \texttt{digit} \mid \texttt{digitpart}$$
$$S \longrightarrow \texttt{part} \mid (S + S) \mid (S - S) \mid (S * S) \mid (S / S)$$

The parse tree for $(1 + (2 * 3))$



Give Operator Precedence

- If more than one operator is present in the expression, the precedence order decides the order in which the operators should be applied.
- Modify grammar: add non-terminals for each precedence level, push the higher levels towards the bottom of the parse trees

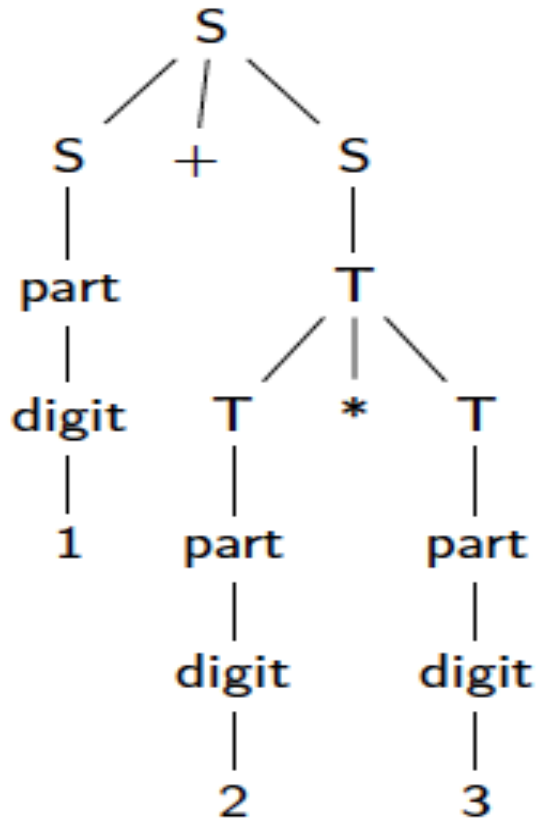
S→ part|S+S |S-S|S*S|S/S

S→ S+S |S-S|T
T→ T*T|T/T|part

Operator Precedence

Parse Tree for 1 + 2 * 3

$$
\begin{array}{lcl}
S & \longrightarrow & S + S \mid S - S \mid T \\
T & \longrightarrow & T * T \mid T / T \mid \texttt{part}
\end{array}
$$



Define Associativity

- If the same operator appears more than once in the same expression, then associativity rule decides the order in which the operators should be applied.
- There are two types of associativity: left and right:
  - Left associativity: operators on the left are applied before the operators on the right.
  - Right associativity: operators on the right are applied before the operators on the left.

Examples: x/y/z becomes (x/y)/z if / is left associative.

Define Associativity: example

Given: S→ S+S |S-S|T
T→ T*T|T/T|part

- Assume that + is left-associative: What is the parse tree for 1 + 2 + 3 * 4?
- Allow expansion only on the left-hand side of the parse tree, how to modify the grammar?

S→ S+T |S-S|T
T→ T*T|T/T|part

Remove Ambiguity: revisit

- There are no techniques that can always convert an ambiguous grammar to an unambiguous grammar
- A language is ambiguous if its grammar is ambiguous
- Some grammars are left ambiguous for better representation of concepts
- Observe how grammar rules can impact the shape of the parse trees

Questions:

- Does the left-derivation or right-derivation strategies impact the ambiguity of the grammar?
- If the grammar is unambiguous, do the two derivation strategies result in the same sequence of production rules?

Design Grammars

How to specify the string patterns:

- regular expressions $a^* b^+$: ab, aab, ... (* the letter occurs 0 or more time, + the letter occurs 1 or more times)
- English description – palindrome on the alphabet of a b c: aba, aa, acca ...
- set $\{a^n(b^m|c^m)|m > n > 0\}$
- grammars

Designing Grammars

1. Use recursive productions to generate an arbitrary number of symbols

   $A \rightarrow xA \mid \varepsilon$ // Zero or more x's

   $A \rightarrow yA \mid y$ // One or more y's

2. Use separate non-terminals to generate disjoint parts of a language, and then combine in a production

   a*b* // a's followed by b's

   $S \rightarrow AB$

   $A \rightarrow aA \mid \varepsilon$ // Zero or more a's

   $B \rightarrow bB \mid \varepsilon$ // Zero or more b's

Designing Grammars

3. To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle

$\{a^n b^n \mid n \geq 0\}$ // N a's followed by N b's

$S \rightarrow aSb \mid \varepsilon$

Example derivation: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

$\{a^n b^{2n} \mid n \geq 0\}$ // N a's followed by 2N b's

$S \rightarrow aSbb \mid \varepsilon$

Example derivation: $S \Rightarrow aSbb \Rightarrow aaSbbbb \Rightarrow aabbbb$

Review: homework and exam points

- grammar
- CFG
- strings, language and grammar
- derivation: left-most and right-most derivations
- parsing
- parse tree
- evaluating an expression using parse tree
- ambiguity
- removing ambiguity
- design grammar