Lecture 2. Context Free Grammar

September 4, 2018

Overview

1. Formal Language and Formal Grammar
2. CFG: Context Free Grammar
3. Strings and Grammar
   - Rewriting
   - Derivation
   - Parsing
4. Parse Tree
5. Ambiguity
6. Design a Grammar
7. Exercises

Specify Patterns in String

a program is a string that follows certain "rules"; the rules can be specified by:

- informal grammar – English description
- formal grammar
  1. what are the atoms? (terminals)
  2. how to compose them to form a sentence?
  3. others, e.g., regular expressions

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

Grammar

1. Rules to define a string in the language; Every programming language has a grammar describing the valid syntax.
2. Bases for parsing – use the given grammar to validate the grammatical correctness of a given program as a sequence of strings.

Formal Grammars Define Formal Languages

- Regular grammar for regular languages: Typical pattern-based searching
- Context-free grammar from context-free languages: Typical programs
- Context-sensitive grammar for context-sensitive languages: Typical counting patterns, scoping in programs
- Unrestricted grammar for recursively enumerable languages

Chomsky Hierarchy. COM S 331 Theory of Computing.

Formal Grammars Define Formal Languages

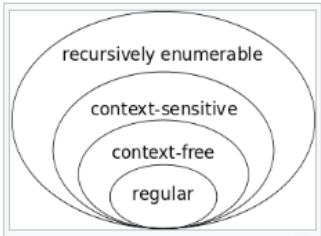
Chomsky Hierarchy

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)*
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow \mathrm{a}$ and $A \rightarrow \mathrm{a}B$

* Meaning of symbols:

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



Set inclusions described by the Chomsky hierarchy

Language: Set of strings

Context Free Grammar (CFG)

1. formal definitions
2. notations
3. examples of CFG
4. derivation (left-most and right-most), parsing

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)

CFG Formal Definition:

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: $X \longrightarrow \omega$, such that $X \in V$ and $\omega \in (\Sigma \cup V)^+$

Context Free Grammar (CFG)

Real number

CFG Examples

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

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .\}$
- $V = \{\texttt{real-number}, \texttt{part}, \texttt{digit}, \texttt{sign}\}$
- $S = \texttt{real-number}$
- P (Backus-Naur form/BNF)

$$
\begin{aligned}
\texttt{real-number} &\longrightarrow \texttt{sign part . part} \\
&\quad \mid \texttt{sign part} \\
\texttt{sign} &\longrightarrow + \mid - \\
\texttt{part} &\longrightarrow \texttt{digit} \mid \texttt{digit part} \mid \varepsilon \\
\texttt{digit} &\longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

+10 $\quad x \mapsto \omega$

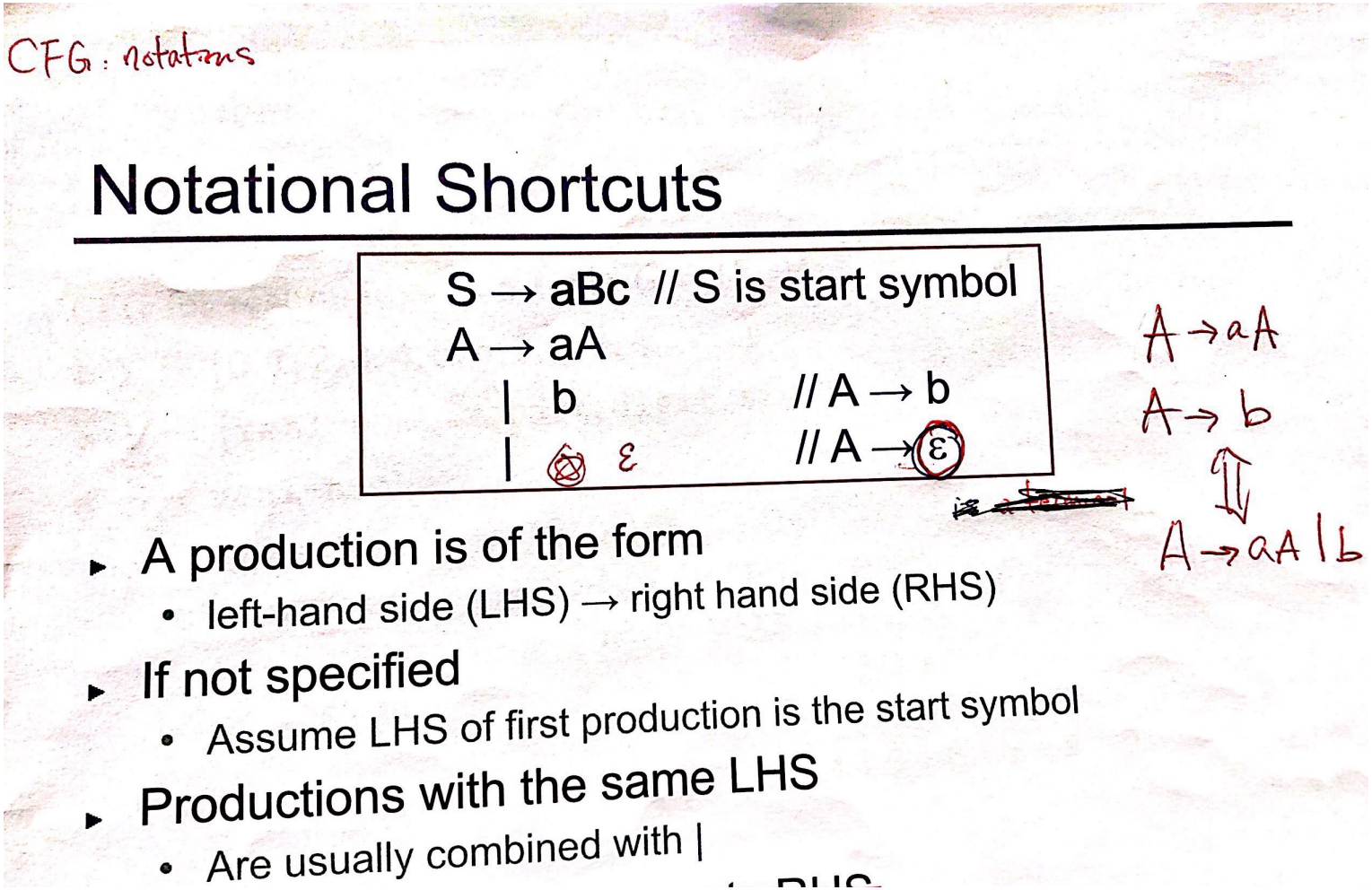
10 $\quad (\Sigma \cup U)^+$

Context Free Grammar (CFG)

Backus-Naur Form

- Context-free grammar production rules are also called Backus-Naur Form or BNF
  - Designed by John Backus and Peter Naur
    - Chair and Secretary of the Algol committee in the early 1960s. Used this notation to describe Algol in 1962
- A production A → B c D is written in BNF as <A> ::= <B> c <D>
  - Non-terminals written with angle brackets and uses ::= instead of →
  - Often see hybrids that use ::= instead of → but drop the angle brackets on non-terminals

Context Free Grammar (CFG)



Context Free Grammar (CFG)

Example Derivations

Generating Strings

- Generating strings from grammar by rewriting
- Example grammar G

  $S \rightarrow 0S \mid 1S \mid \varepsilon$
- Generate string 011 from G as follows:

$S \Rightarrow 0S$ // using $S \rightarrow 0S$

$\Rightarrow 01S$ // using $S \rightarrow 1S$

$\Rightarrow 011S$ // using $S \rightarrow 1S$

$\Rightarrow 011$ // using $S \rightarrow \varepsilon$

In-Class Exercises: Strings and Derivations

$S \rightarrow (S)|\epsilon$

- $S \implies ()$
- $S \implies (S) \implies (())$
- $S \implies SS \implies ()()$

Context Free Grammar (CFG)

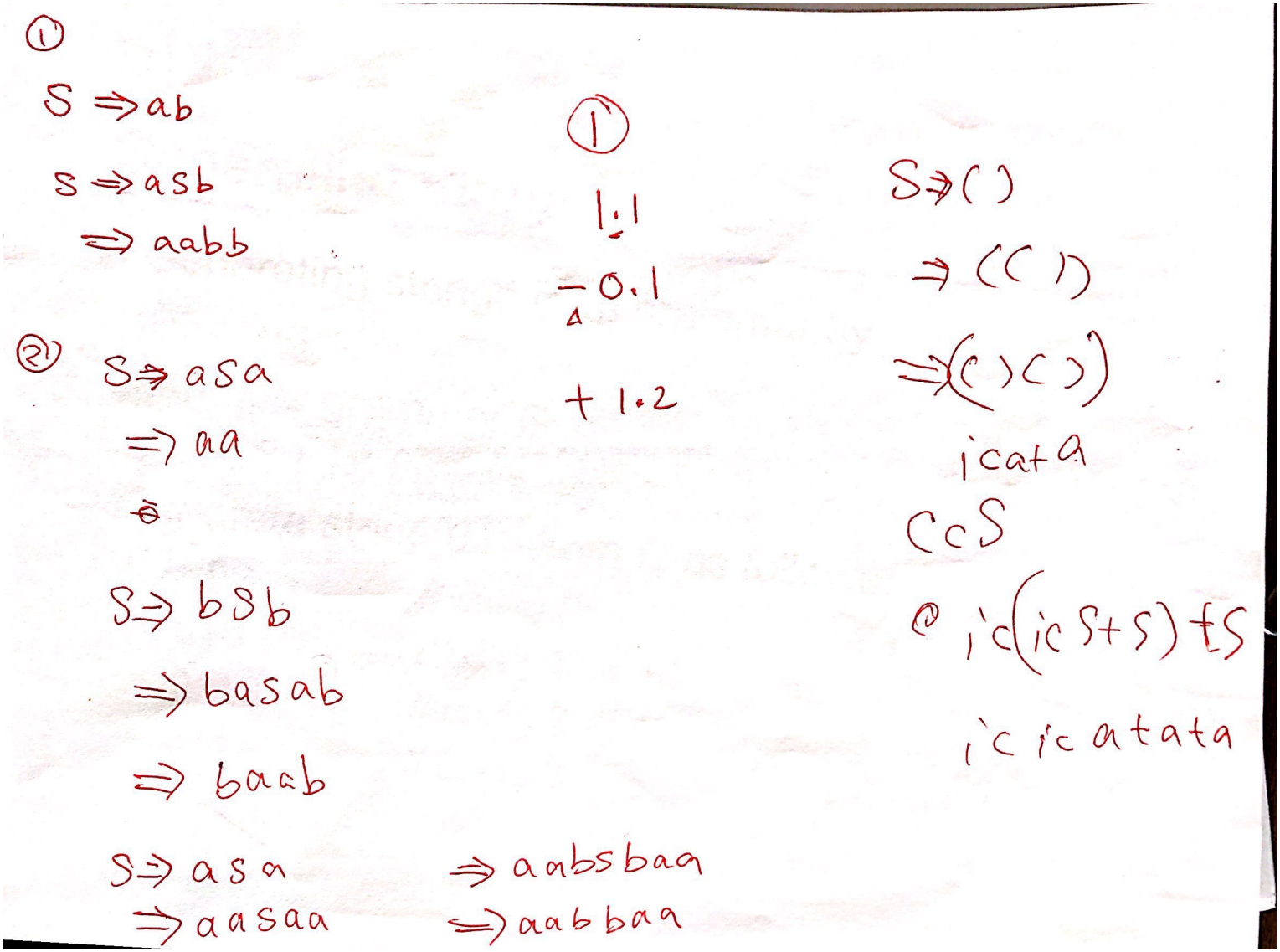
CFG Derivation Examples

Find the pattern of strings generated by the following grammars.

1. $S \longrightarrow aSb \mid ab$
2. $S \longrightarrow aSa \mid bSb \mid \epsilon$
3. $S \longrightarrow SS \mid (S) \mid ()$
4. $S \longrightarrow i\ c\ S\ t\ S \mid a$
5. $S \longrightarrow \texttt{part} \mid S\ +\ S \mid S\ -\ S$

Context Free Grammar (CFG)

Different Derivations



Accepting Strings (Informally)

- Checking if $s \in L(G)$ is called acceptance
  - Algorithm: Find a rewriting starting from G's start symbol that yields s
  - A rewriting is some sequence of productions (rewrites) applied starting at the start symbol
    - $011 \in L(G)$ according to the previous rewriting
- Terminology
  - Such a sequence of rewrites is a derivation or parse
  - Discovering the derivation is called parsing

Derivations

- Notation
  - $\Rightarrow$ 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
  - $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 010$
  - $S \Rightarrow^+ 010$
  - $010 \Rightarrow^* 010$

Language Generated by Grammar

- L(G) the language defined by G is

$$L(G) = \{ s \in \Sigma^* \mid S \Rightarrow^+ s \}$$

  - S is the start symbol of the grammar
  - Σ is the alphabet for that grammar

- In other words
  - All strings over Σ that can be derived from the start symbol via one or more productions

Strings and Grammar

A program P is a string s over term (keywords, symbols, numbers, operators, etc). P is said to be syntactically correct if and only if s can be generated/derived from the grammar G for the language in which the program is written (i.e., $s \in L(G)$).
The derivation of s from G is a sequence of application of production rules of the grammar.

Derivations

Leftmost vs Rightmost Derivation

Derivation of $1 + 2 + 3$

Leftmost	**Rightmost**
$S \Rightarrow S + S$	$S \Rightarrow S + S$
$\Rightarrow$ part $+ S$	$\Rightarrow S +$ part
$\Rightarrow$ digit $+ S$	$\Rightarrow S +$ digit
$\Rightarrow 1 + S$	$\Rightarrow S + 3$
$\Rightarrow 1 + S + S$	$\Rightarrow S + S + 3$
$\Rightarrow 1 +$ part $+ S$	$\Rightarrow S +$ part $+ 3$
$\Rightarrow 1 +$ digit $+ S$	$\Rightarrow S +$ digit $+ 3$
$\Rightarrow 1 + 2 + S$	$\Rightarrow S + 2 + 3$
$\Rightarrow 1 + 2 +$ part	$\Rightarrow$ part $+ 2 + 3$
$\Rightarrow 1 + 2 +$ digit	$\Rightarrow$ digit $+ 2 + 3$
$\Rightarrow 1 + 2 + 3$	$\Rightarrow 1 + 2 + 3$

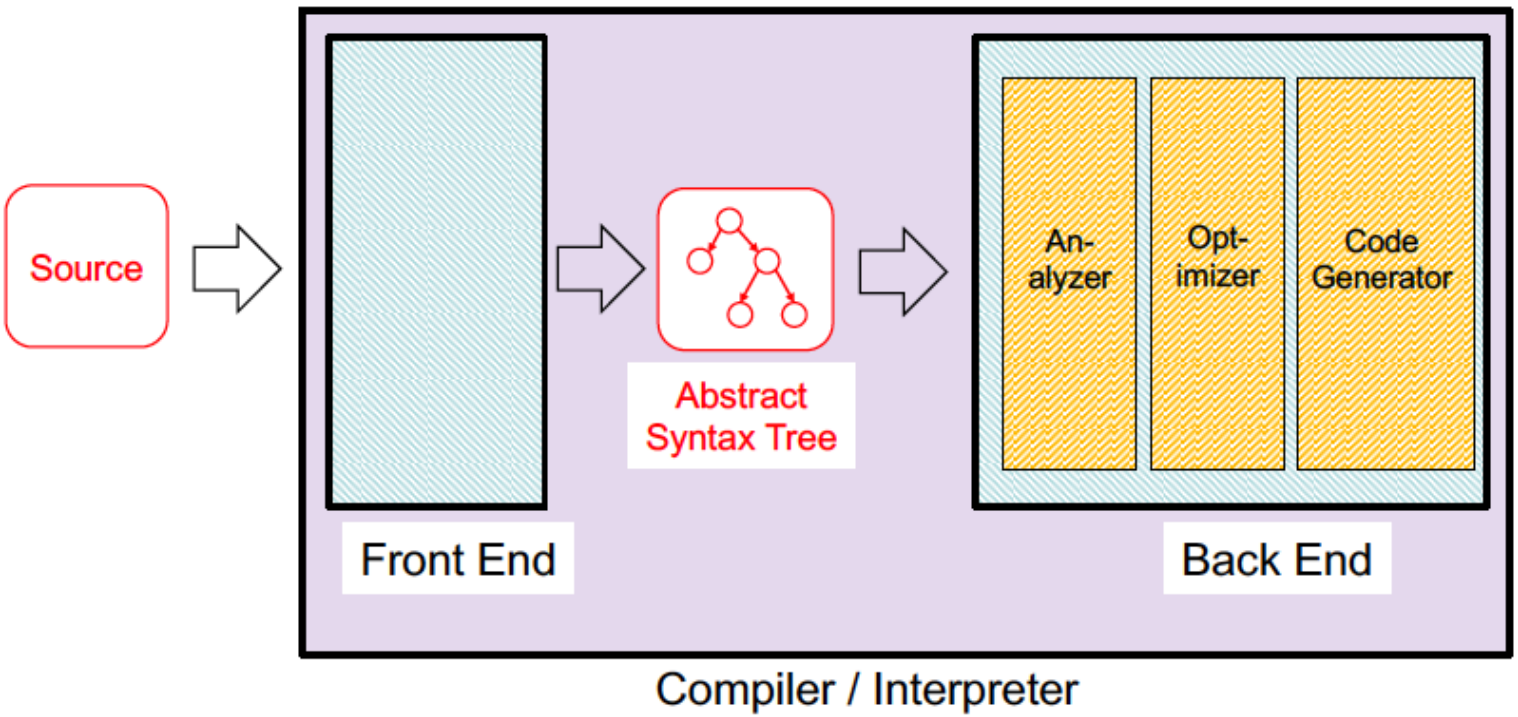
Derivations

Leftmost vs Rightmost Derivation

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

Parsing

Architecture of Compilers, Interpreters



Parsing

1. Parsing
2. **Parse tree**

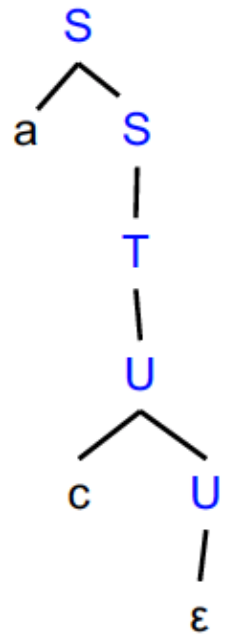
Parse Tree Example

$S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac$

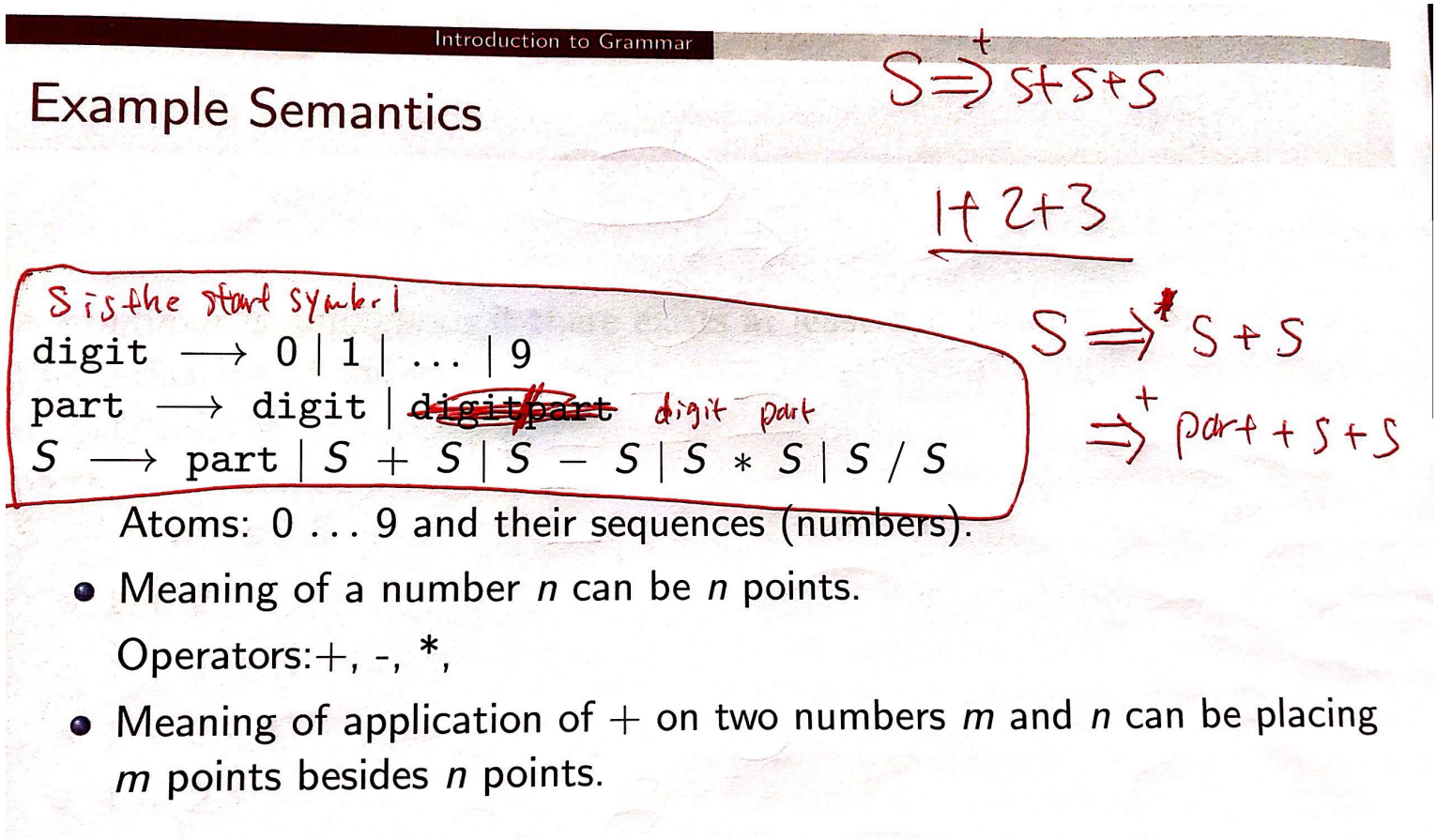
$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$

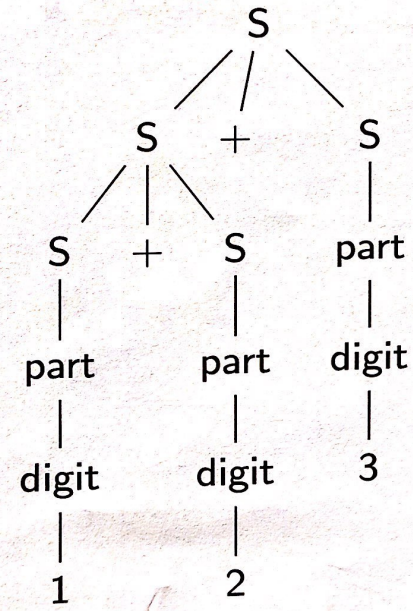


Semantics


Introduction to Grammar
Example Semantics
S ⇒+ S+S+S
1+2+3
S is the start symbol
digit ⟶ 0 | 1 | ... | 9
part ⟶ digit | digitpart
digit part
S ⟶ part | S + S | S − S | S ∗ S | S / S
S ⇒* S+S
⇒+ part+S+S
Atoms: 0 . . . 9 and their sequences (numbers).
Meaning of a number n can be n points.
Operators:+, -, *,
Meaning of application of + on two numbers m and n can be placing m points besides n points.

Parse Tree – Semantics

Evaluating Sematics



((1+2)+3) =6

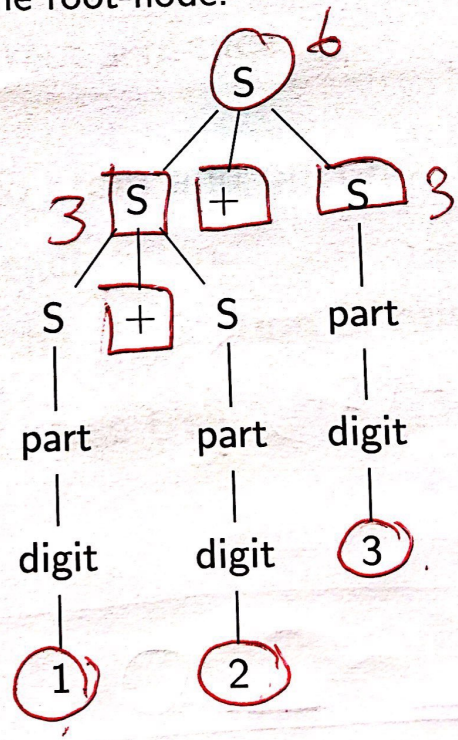
sem(+, sem(+, sem(1), sem(2)), sem(3))

Parse Tree – Semantics

Parse Tree and Semantics

Evaluate the semantics using parse tree.

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



Parse Tree – Semantics

Semantics

Meaning of a syntactically correct sentence.

1. Classify the terminals into atom and operator classes.
2. Associate meaning with each atom.
3. Associate meaning with the application of operator(s) on atom(s).

Evaluate the semantics using parse tree.

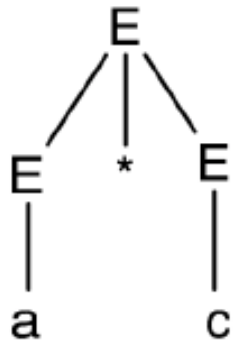
Parse Trees for Expressions

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

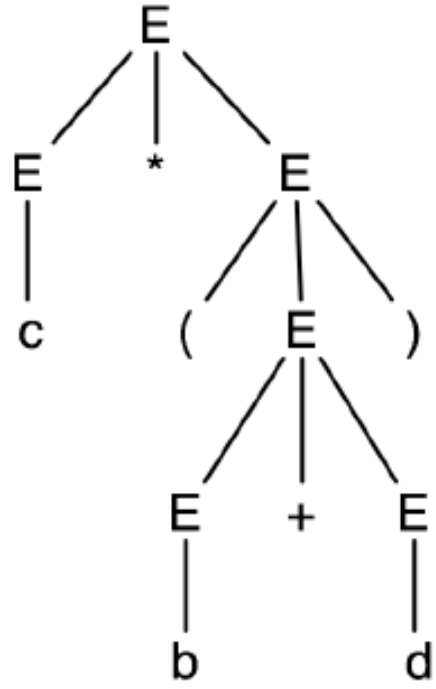
$$E \rightarrow a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E)$$

a

a*c



c*(b+d)



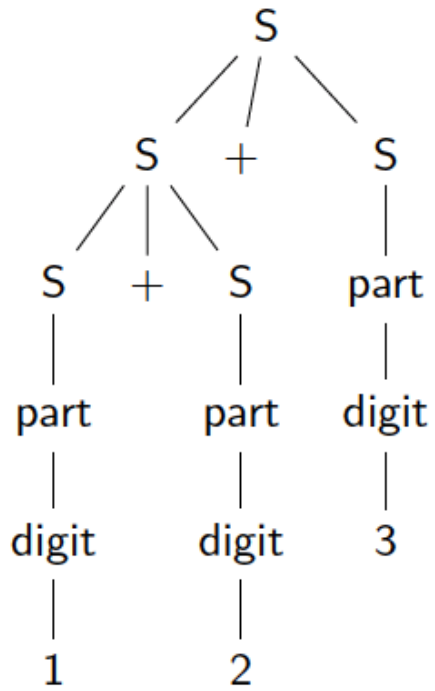
Parse Tree

A parse tree results from the derivation sequence.

- Each node in the tree is a terminal or non-terminal in the production rule.
- Each edge in the tree from a non-terminal results from the application of production rule on the non-terminal.
- Application of production rule always result in new nodes in the tree.
- A terminal is a leaf node

Ambiguity

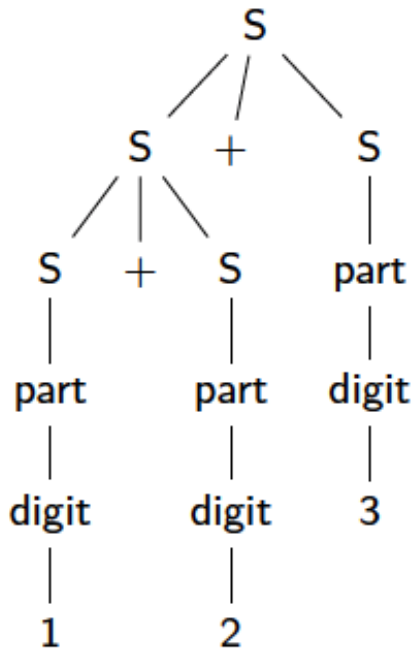
Parse Tree for 1 + 2 + 3

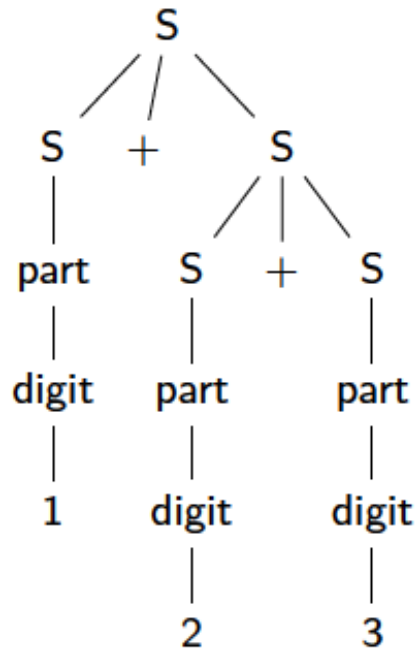


The same parse tree can be generated by left-most and right-most derivation.

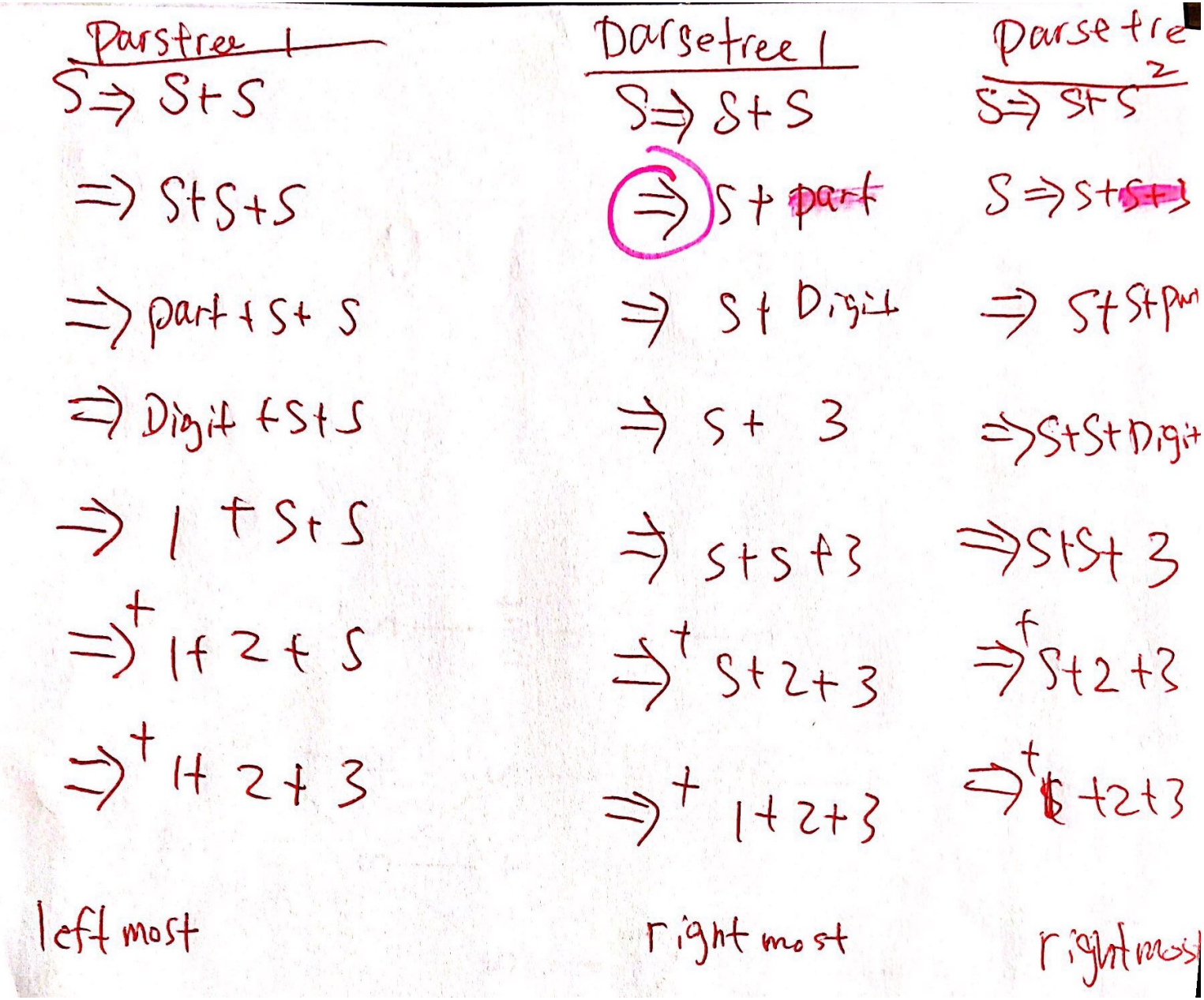
Ambiguity

Parse Tree for 1 + 2 + 3





Ambiguity


Parstree 1
S ⇒ S+S
⇒ S+S+S
⇒ part+S+S
⇒ Digit+S+S
⇒ 1+S+S
⇒+ 1+2+S
⇒+ 1+2+3
leftmost
Darsetree 1
S ⇒ S+S
⇒ S+part
⇒ S+Digit
⇒ S+3
⇒ S+S+3
⇒+ S+2+3
⇒+ 1+2+3
rightmost
Parsetre 2
S ⇒ S+S
S ⇒ S+S+S
⇒ S+S+part
⇒ S+S+Digit
⇒ S+S+3
⇒+ S+2+3
⇒+ 1+2+3
rightmost

Ambiguity

A grammar is ambiguous if there exists at least two distinct parse trees for the derivation of the same string.

$\texttt{digit} \longrightarrow 0 \mid 1 \mid \ldots \mid 9$

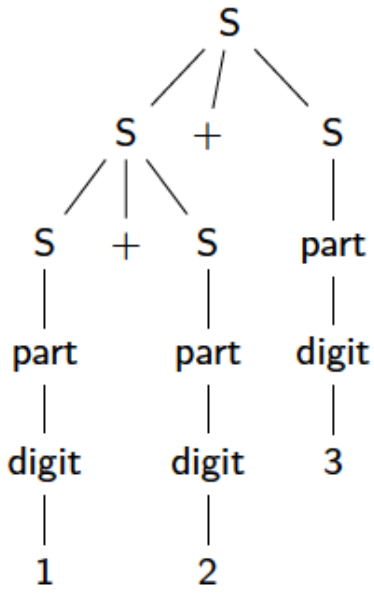
$\texttt{part} \longrightarrow \texttt{digit} \mid \texttt{digitpart}$

$S \longrightarrow \texttt{part} \mid S + S \mid S - S \mid S * S \mid S / S$

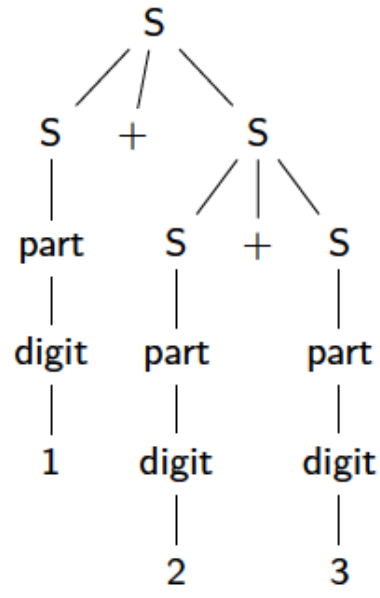
- $1 + 2$
- $1 + 2 + 3$
- $1 + 2$ - 3
- $1 + 2 * 3$

Ambiguity

Parse Tree for 1 + 2 + 3



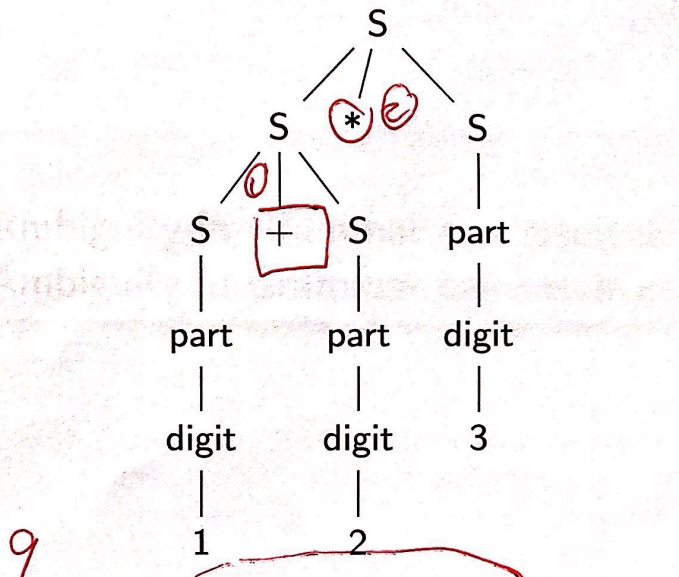
sem(+, sem(+, sem(1), sem(2)), sem(3))



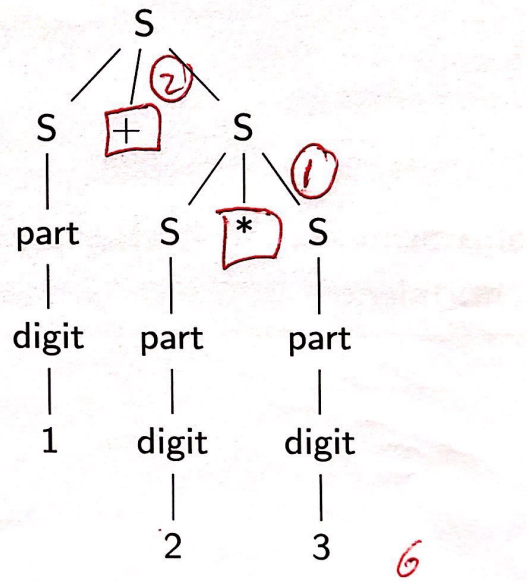
sem(+, sem(1), sem(+, sem(2), sem(3)))

Ambiguity

More Critical for 1 + 2 * 3



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



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

Ambiguity

- Ambiguity in Grammar can result in incorrect application of semantic rules
- Ambiguity in Grammar can result in incorrect syntax-directed translation

Ambiguity

Another example

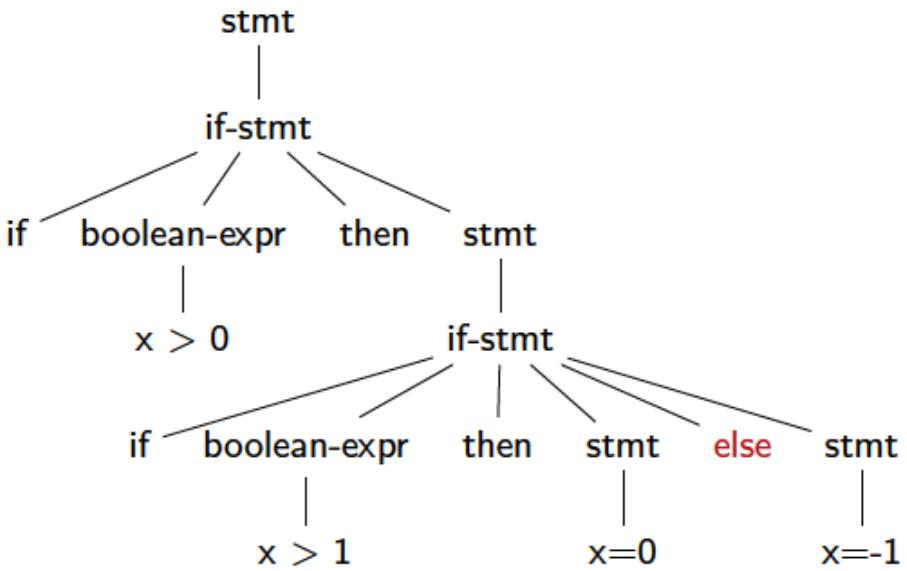
$$
\begin{array}{rcl}
\texttt{stmt} & \longrightarrow & \ldots \mid \ldots \mid \texttt{if-stmt} \\
\texttt{if-stmt} & \longrightarrow & \textbf{if}\ \texttt{boolean-expr}\ \textbf{then}\ \texttt{stmt} \\
 & & \mid \textbf{if}\ \texttt{boolean-expr}\ \textbf{then}\ \texttt{stmt}\ \textbf{else}\ \texttt{stmt}
\end{array}
$$

if $(x > 0)$ **then**
if $(x > 1)$ **then** $x = 0$
else $x = -1$

Ambiguity

Parse Trees

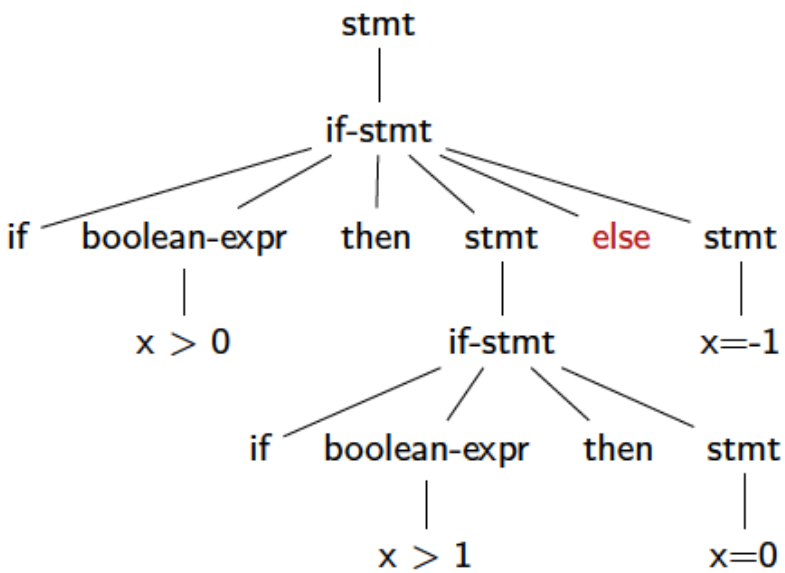
if $(x > 0)$ **then**
if $(x > 1)$ **then** $x = 0$
else $x = -1$



Ambiguity

Parse Trees

if $(x > 0)$ **then**
if $(x > 1)$ **then** $x = 0$
else $x = -1$



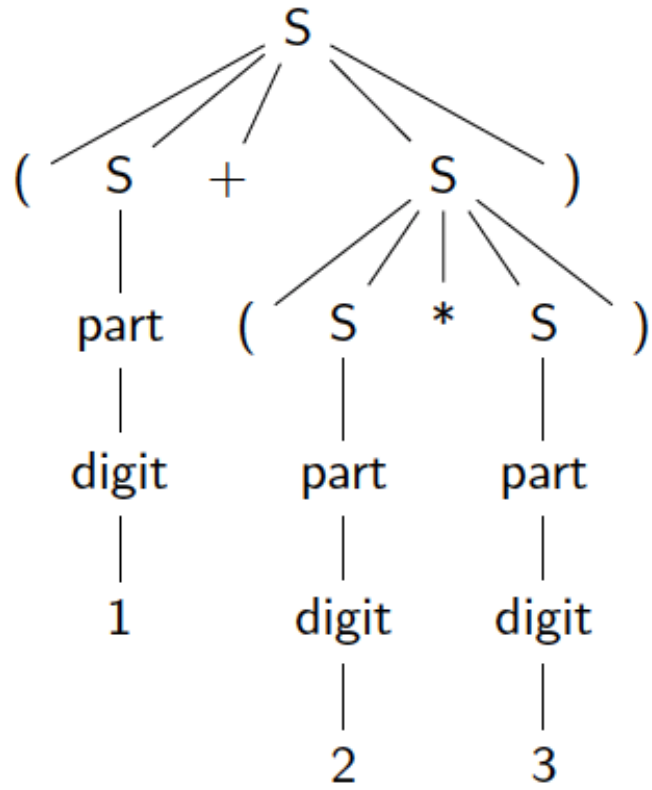
Removing Ambiguity

- Add delimiters (e.g., parenthesis; begin and end in if statements)
- Add operator precedence and associativity

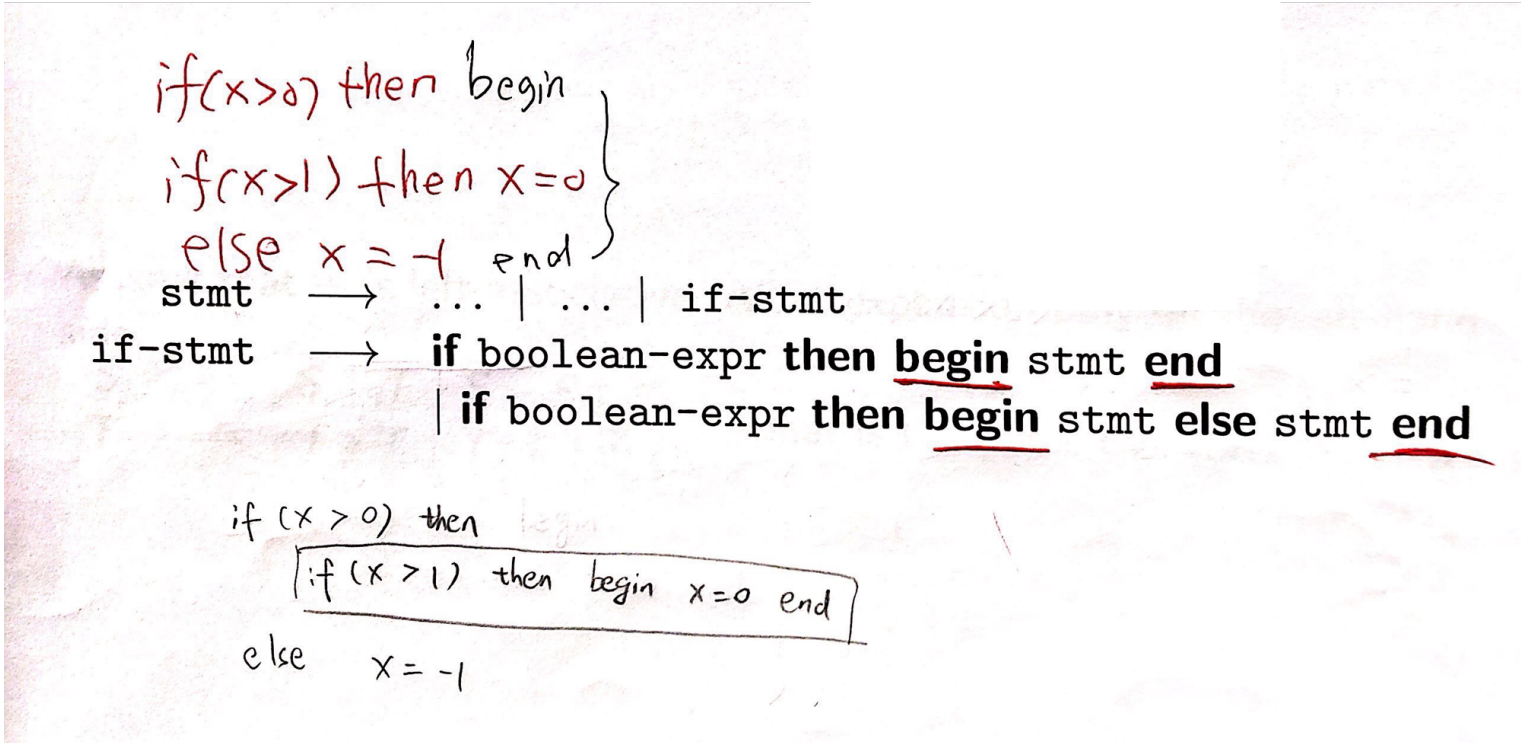
Delimiters

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

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



Delimiters


if(x>0) then begin
if(x>1) then x=0
else x=-1 end
stmt ⟶ ... | ... | if-stmt
if-stmt ⟶ if boolean-expr then begin stmt end
| if boolean-expr then begin stmt else stmt end
if (x > 0) then
if (x > 1) then begin x=0 end
else x = -1

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.

Add non-terminals for each precedence level. Push the higher levels towards the bottom of the parse-tree (stratification of tree)

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

old

$S \rightarrow S+S \mid S-S \mid S*S \mid S/S$

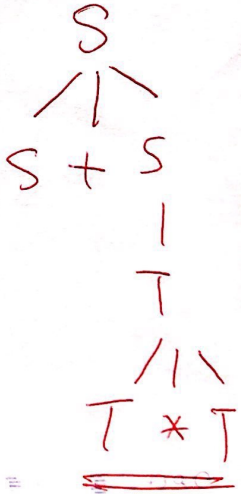
$1+2*3$

$S \Rightarrow S+S$

$\Rightarrow S+T$

$\Rightarrow S+T*T$

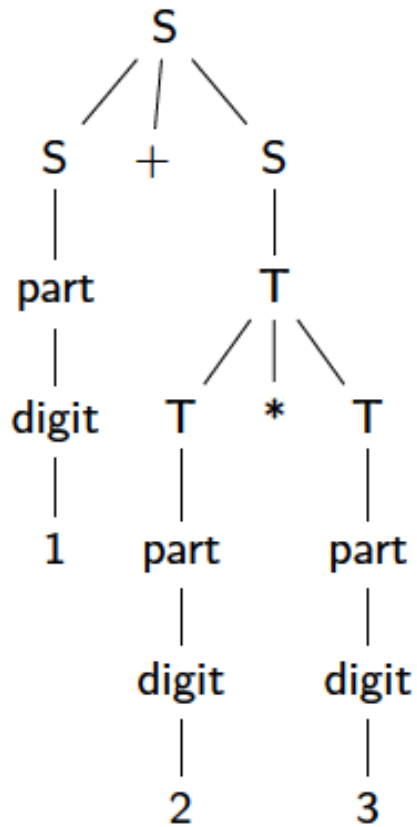
$\Rightarrow 1+2*3$



Operator Precedence

Parse Tree for 1 + 2 * 3

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



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 $(x/y)/z$ is $/$ is left associative.

Associativity

Assume that $+$ is left-associative. Allow expansion only on the left-hand side.

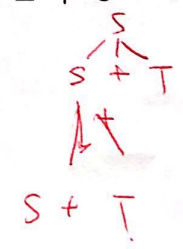
$$\begin{array}{rcl} S & \longrightarrow & S + T \mid S - S \mid T \\ T & \longrightarrow & T * T \mid T / T \mid \texttt{part} \end{array}$$

What is the parse tree for $1 + 2 + 3 * 4$

$S \Rightarrow S + T \Rightarrow S + T + T$

$\Rightarrow T + T \Rightarrow 1 + 2$

$\Rightarrow$ part



1 + 2 + 3

Dealing With Ambiguous Grammars

- Ambiguity is bad
  - Syntax is correct
  - But semantics differ depending on choice
    - Different associativity (a-b)-c vs. a-(b-c)
    - Different precedence (a-b)*c vs. a-(b*c)
    - Different control flow if (if else) vs. if (if) else
- Two approaches
  - Rewrite grammar
    - **Grammars are not unique** – can have multiple grammars for the same language. But result in different parses.
  - Use special parsing rules
    - Depending on parsing tool

Elimination of Ambiguity

- In general, there is no technique that will always convert an ambiguous grammar to a unambiguous one using additional rules.
- Some grammars are left ambiguous for better representation of concepts
- A language is ambiguous if any grammar that generates the strings in the language is ambiguous.

More on all these in COM S 331 and COM S 440.

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

Designing Grammars

4. For a language that is the union of other languages, use separate nonterminals for each part of the union and then combine

$\{ a^n(b^m|c^m) \mid m > n \geq 0\}$

Can be rewritten as

$\{ a^nb^m \mid m > n \geq 0\} \cup \{ a^nc^m \mid m > n \geq 0\}$

$S \rightarrow T \mid V$

$T \rightarrow aTb \mid U$

$U \rightarrow Ub \mid b$

$V \rightarrow aVc \mid W$

$W \rightarrow Wc \mid c$

Review

1. CFG
   - Formal language and formal grammar
   - Definitions
2. String and Grammar
   - Derivations, left-most derivation and right-most derivation
3. Parse Tree
4. Ambiguity
5. Eliminate Ambiguity
6. Design Grammars