BITS Pilani
Dubai Campus

Principles of Programming Languages

CS F301

innovate achieve lead

BITS Pilani
Dubai Campus

Language description: Syntactic structure

Objective

- Introduction
- Expression Notation
- Abstract Syntax Tree
- Lexical Syntax
- BNF and context-free grammars
- Derivation and Parse trees
  - Bottom Up
  - Top Down
- Ambiguity in grammars
- Grammar for Expressions

Introduction

- Syntax
  - The form or structure of the expressions, statements, and program units.
  - Includes two layers
    - ❖Lexical layer
    - ❖Grammar layer

- Semantics
  - The meaning of the expressions, statements, and program units

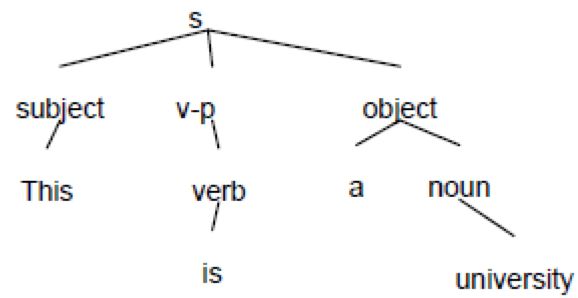
Comparison with Eng Lang.

- Grammar is way to
  - Describe a language : all possible legally correct programs
  - Analyze a sentence : check if a program is valid
  - Derive a sentence: program

sentence	–>	**<subject> <verb-phrase> <object>**
subject	–>	This | Computers | I
verb-phrase	–>	**<adverb> <verb>** | **<verb>**
adverb	–>	never
verb	–>	is | run | am | tell
object	–>	the **<noun>** | a **<noun>** | **<noun>**
noun	–>	university | world | cheese | lies

This is a university.
Computers run the world.
I am the cheese.
I never tell lies.

sentence	–>	**<subject> <verb-phrase> <object>**
	–>	This **<verb-phrase> <object>**
	–>	This **<verb> <object>**
	–>	This is **<object>**
	–>	This is a **<noun>**
	–>	This is a university



Expression Notation

Three ways

- **Prefix :** Binary operator written before its operands Eg. +ab
  - ✓ Easy to decode during left to right scan of an expression
- **Postfix :** Binary operator written after its operands Eg. ab+
  - ✓ Can be evaluated with the help of a stack
- **Infix :** Binary operator written between its operands Eg. a+b

Note:

✓ Prefix and postfix notation are called parenthesis free as the operands of each operator can be found unambiguously.

Prefix Notation: Evaluated L to R

Example 1:

*** + 20 30 60**

Step 1: * (20+30) 60

Step 2: *50 60

Step 3: 3000

Practice Question

^ / 20 + 60 – 80 40 3

Ans: 1/125

Example 2:

*** 20 + 30 60**

Step 1: * 20 (30 + 60)

Step 2: * 20 90

Step 3: 1800

Practice Question

sqrt + + * 30 50 500 * 5 100

Ans: 50

Postfix Notation : Evaluated R to L

Example 1:

20 30 + 60 *

Step 1: (20+30) 60 *

Step 2: 50 60 *

Step 3: 3000

Practice Question

50 40 + 20 30 - *

Ans: -900

Example 2:

20 30 60 + *

Step 1: 20 (30 + 60) *

Step 2: 20 90 *

Step 3: 1800

Practice Question

40 70 - 700 10 / + sqrt

Ans: sqrt (40)

Postfix Evaluation Example

Postfix Expression : 2536+**5/2-		
Token	**Action**	**Stack**
2	Push **2** to stack	[2]
5	Push **5** to stack	[2, 5]
3	Push **3** to stack	[2, 5, 3]
6	Push **6** to stack	[2, 5, 3, 6]
+	Pop **6** from stack	[2, 5, 3]
	Pop **3** from stack	[2, 5]
	Push **3+6 =9** to stack	[2, 5, 9]
*	Pop **9** from stack	[2, 5]
	Pop **5** from stack	[2]
	Push **5*9=45** to stack	[2, 45]
*	Pop **45** from stack	[2]
	Pop **2** from stack	[]
	Push **2*45=90** to stack	[90]
5	Push **5** to stack	[90, 5]
/	Pop **5** from stack	[90]
	Pop **90** from stack	[]
	Push **90/5=18** to stack	[18]
2	Push **2** to stack	[18, 2]
-	Pop **2** from stack	[18]
	Pop **18** from stack	[]
	Push **18-2=16** to stack	[16]
Result : 16		

Prefix Evaluation Example

Prefix Expression : -/*2*5+3652

Reversed Prefix Expression: 2563+5*2*/-

Token	Action	Stack
2	Push **2** to stack	[2]
5	Push **5** to stack	[2, 5]
6	Push **6** to stack	[2, 5, 6]
3	Push **3** to stack	[2, 5, 6, 3]
+	Pop **3** from stack	[2, 5, 6]
	Pop **6** from stack	[2, 5]
	Push **3+6 =9** to stack	[2, 5, 9]
5	Push **5** to stack	[2, 5, 9, 5]
*	Pop **5** from stack	[2, 5, 9]
	Pop **9** from stack	[2, 5]
	Push **5*9=45** to stack	[2, 5, 45]
2	Push **2** to stack	[2, 5, 45, 2]
*	Pop **2** from stack	[2, 5, 45]
	Pop **45** from stack	[2, 5]
	Push **2*45=90** to stack	[2, 5, 90]
/	Pop **5** from stack	[2, 5]
	Pop **90** from stack	[2]
	Push **90/5=18** to stack	[2, 18]
-	Pop **18** from stack	[2]
	Pop **2** from stack	[]
	Push **18-2=16** to stack	[16]

Result : 16

S

Infix Notation

The infix notation works on

1. Associativity
2. Precedence = BODMAS

1. Associativity

Left associative and right-associative operators: An operator is left-associative if sub-expressions containing multiple occurrences of the operator are grouped from left to right and vice-versa for right associative operators. Example, +, -, /, * are left associative operators whereas exponentiation and assignment are right associative.

Eg 1 : b* b – 4 * a *c

(LEFT ASSOCIATIVE)

(b * b) – ((4 * a) * c)

Eg. 2 : $2^{3\ \wedge 2}$

(RIGHT ASSOCIATIVE)

$2^{(3 \wedge 2)} = 2^9$

512

Eg. 3 : x= (2+3) * 4 – 7

(RIGHT ASSOCIATIVE)

X= (5 * 4) -7

X= 20 -7

X =13

Mixfix Notation

- Symbols and keywords occur with components of expression.
    - Eg. **If** a > b **then** a **else** b

    If, then, else = keywords

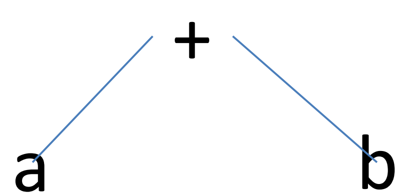
    A, b > = components of expression

Abstract Syntax tree

- Abstract syntax of a language identifies the meaningful components of each construct in the language.

- Tree showing the operator / operand structure of an expression is called an abstract syntax tree because they show the syntactic structure of an expression independent of the notation in which the expression was originally written.

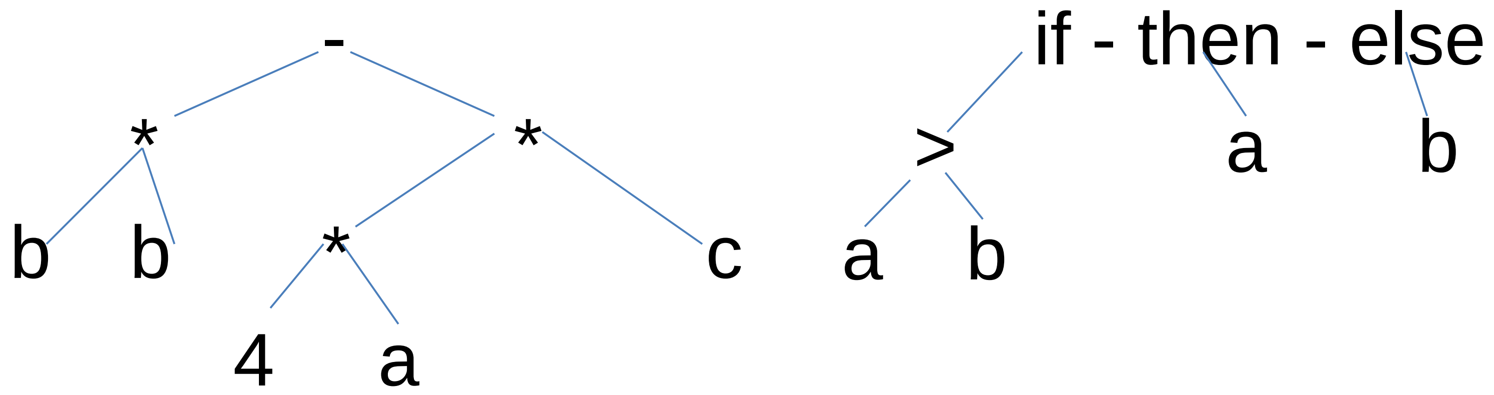
- Example, +ab, a+b and ab+ have the same abstract syntax tree



Abstract Syntax tree

Eg: b * b – 4 * a * c

Eg: if a > b then a else b



Lexical Syntax

innovate achieve lead

- ✓ It helps to group characters of the source program into meaningful sequence (omitting blank spaces and comments) and generate tokens (also called terminals).
- ✓ Syntax of token

  <token name, attribute value>

- ✓ Token classes:

1. Keyword eg if, else etc.
2. Operator eg. + , - etc.
3. Variable / Identifier eg. Interest, a, total
4. Constant eg. Numeric constant or string constant
5. Punctuation mark eg. , ; { }

Eg. position= initial + rate * 60 Generated tokens:

<id, position> <op,=> <id, initial> <op,+> <id, rate> <op,*> <number, 60>

Eg. b * b – 4 * a * c

Generated tokens:

<id, b> <op,*> <id, b> <op,-> <number, 4> <op,*> <id, a> <op,*> <id, c>

Context Free Grammar (CFG)

- Used to specify syntax of a programming language.

- CFG has 4 parts
  - A **set of tokens or terminals**; atomic symbols of the language.
  - A set of **non-terminals**.
  - A **set of rules** (called productions)
    - Each production: has a nonterminal on its left hand-side, the symbol **::=** or **->** and string of terminals/non-terminals on its right-hand side.
  - A non-terminal is the chosen as the starting non-terminal. Unique start terminal called **starting symbol**

Bacus Naur Form (BNF)

- BNF notations is used to specify the grammar
- 4 parts
  - **Terminal** (tokens) appear as keyword, operator, identifiers, constant, punctuation mark
  - **Non-terminals** are enclosed between <>: eg. < fraction>
  - **Productions.**

    Read ::= as ‘can be’ , | as or

    – <fraction> ::= <digit>| <digit><fraction>

    – Fraction can be a digit or fraction can be a digit followed by fraction

CFG for real number BNF

<real-number> ::= <integer_part> . <fraction>

<integer_part> ::= <digit> | <integer_part> <digit>

<fraction> ::= <digit>| <digit> <fraction>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9

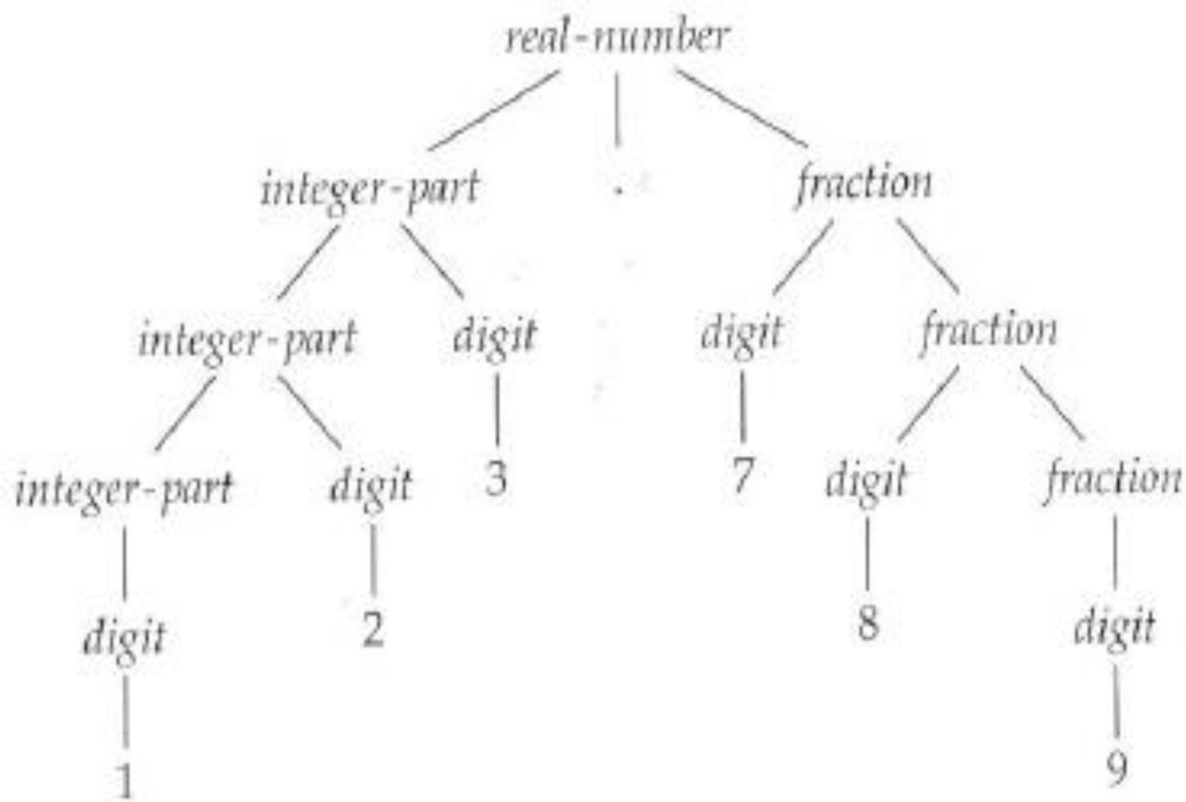
Parse Tree

- Production rules are rules for building strings of tokens
- Begin with the starting nonterminal, & use the rules to build a tree
- The parse tree – (concrete syntax tree)
  - ✓ Each leaf is labeled with a terminal.
  - ✓ Each non leaf is labeled with a non terminal.
  - ✓ Root is labeled with the starting non terminal.
  - ✓ Generates the string formed by reading terminals at its leaves from left to right
  - ✓ A string is only in a language if it is generated by some parse tree
- Construction of a parse tree is called parsing.
- A single production generates a parse tree of the form
  - Eg. <real-no> ::- <integer-part> . <fractional-part>

Parse Tree Example

<real-number> ::= <integer_part> . <fraction>
<integer_part> ::= <digit> | <integer_part> <digit>
<fraction> ::= <digit>| <digit> <fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9

- Represent
  123.789



Syntactic ambiguity of a CFG

- Consider the grammar

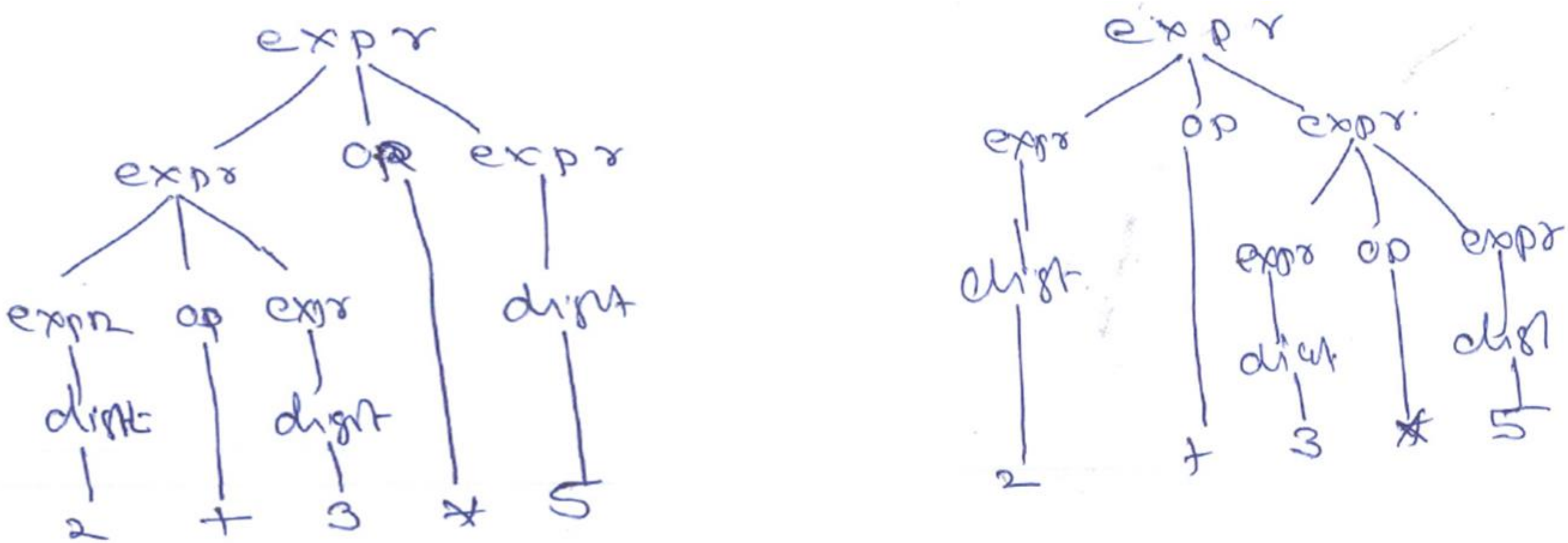
<expr> ::= <expr> <op> <expr> | < digit>

<op> :: + | - | * | /

<digit> ::= 0 | 1 | 2 | …. | 9

- Generate parse tree of the sentence 2 + 3 * 5

Syntactic ambiguity of a CFG Eg1.



<expr> ::= <expr> <op> <expr> | < digit>

<op> :: + | - | * | /

<digit> ::= 0 | 1 | 2 | …. | 9

NOTE: 2 Parse trees for the same sentence. Hence Grammar is ambiguous. So, A grammar for a language is syntactically ambiguous or ambiguous if some String in its language has more than one parse tree.

Syntactic ambiguity of a CFG Eg 2

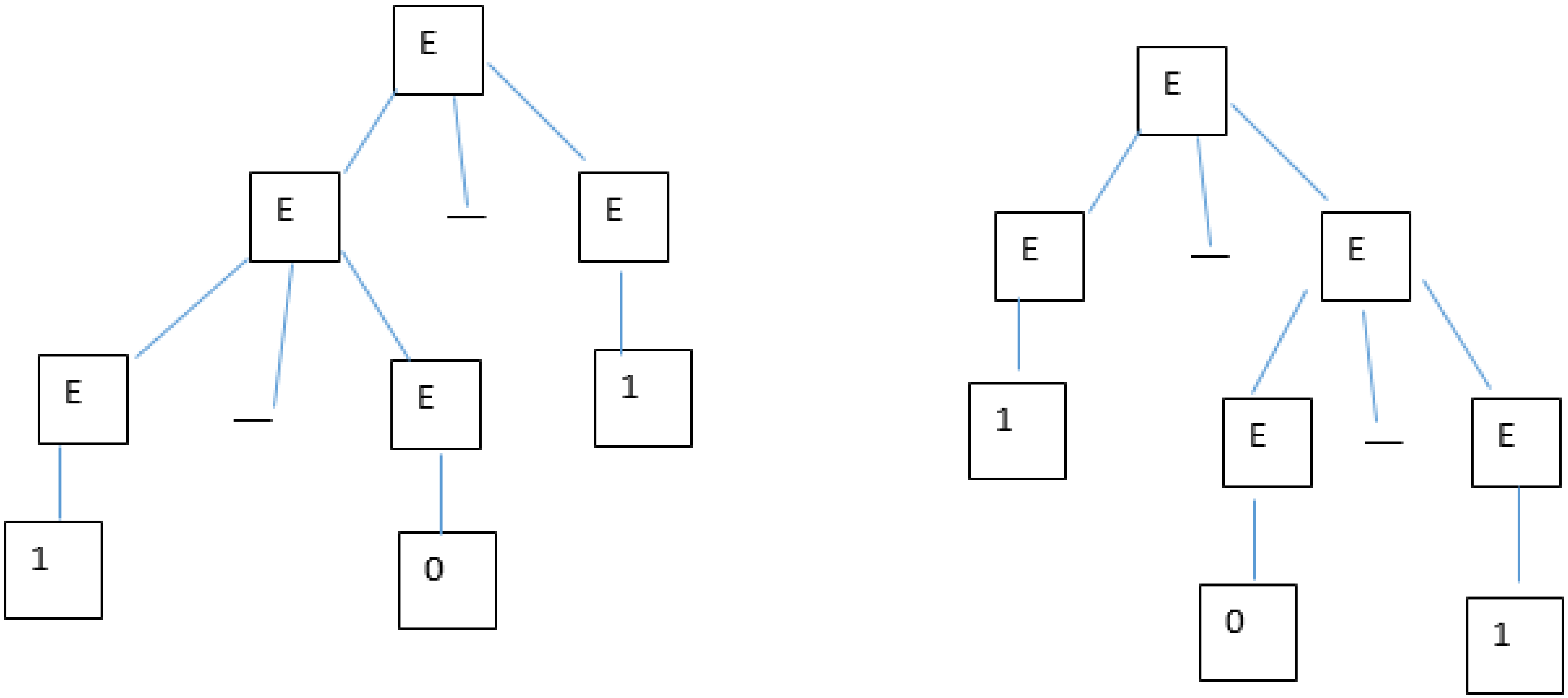
Task:

Generate parse tree for the string 1 – 0 – 1 using the following grammar and hence analyze if the grammar is ambiguous or not?

Grammar:

E ::= E – E | 0 | 1

Syntactic ambiguity of a CFG Eg 2



NOTE: 2 Parse trees for the same string. Hence Grammar is ambiguous.

Syntactic ambiguity of a CFG Eg 3

Dangling else ambiguity

Grammar:

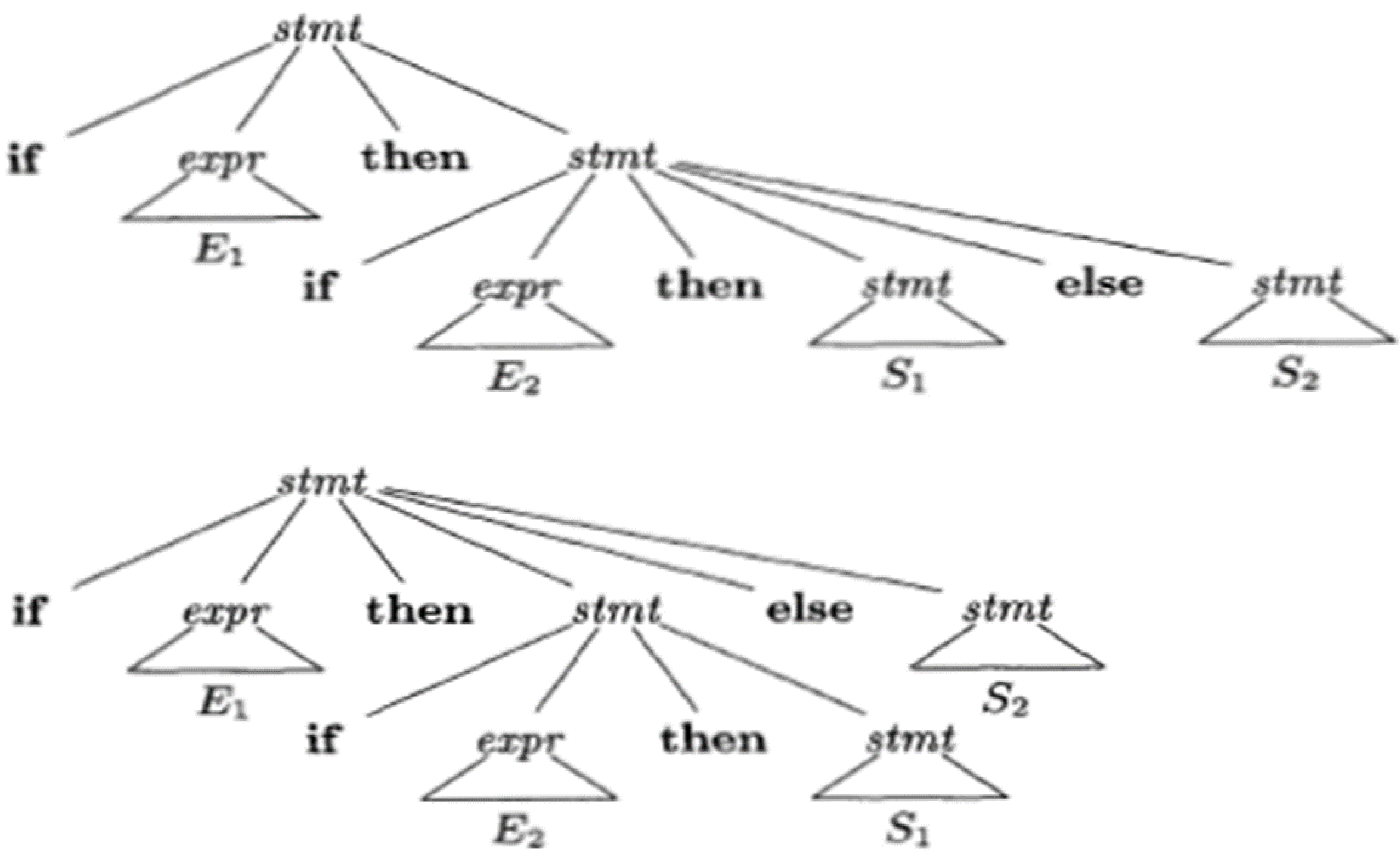
S ::= if E then S

S ::= if E then S else S

String to be generated:

```
if  E1 then if E2 then S1 else S2
```

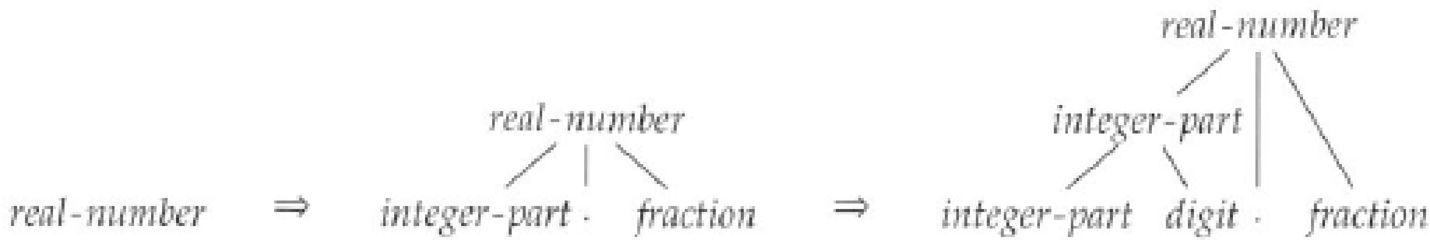
Syntactic ambiguity of a CFG Eg 3



Since we have two parse tress for the given String, the grammar is ambiguous. Moreover we cannot uniquely associate the else with an if, hence its dangling else.

Derivations

- Text version of Parse Tree,
- Eg.



$$real\text{-}number \Rightarrow integer\text{-}part\ .\ fraction \Rightarrow integer\text{-}part\ digit\ .\ fraction \Rightarrow$$

- 2 possibilities
  - **Top down :**
    - Start from Starting symbol and derive the sentence.
    - Replace the LHS of a production by RHS
  - **Bottom up:**
    - Start from the sentence and reach the start symbol.
    - Replace the RHS of a production by the LHS

Top Down Derivation Eg.

- Example of top down derivation:
- Derive the string “21.89” using the grammar for real numbers

real-number ⇒ *integer-part* . *fraction*

⇒ *integer-part digit* . *fraction*

⇒ *digit digit* . *fraction*

⇒ 2 *digit* . *fraction*

⇒ 2 1 . *fraction*

⇒ 2 1 . *digit fraction*

⇒ 2 1 . 8 *fraction*

⇒ 2 1 . 8 *digit*

⇒ 2 1 . 8 9

<real-number> ::= <integer_part> . <fraction>
<integer_part> ::= <digit> | <integer_part> <digit>
<fraction> ::= <digit>| <digit> <fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9

Bottom up Derivation Eg.

Reduces a sentence/string to start symbol by replacing the LHS/body of production with the RHS

Consider the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow int \mid int * T \mid (E)$$

Check if the string

int*int +int

belongs to the grammar using bottom up derivation

Bottom up Derivation Eg.

Derivation	Rule used
int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

Step 1

Derivation and Parse Tree

int * int + int

$E \rightarrow T + E \mid T$

$T \rightarrow int \mid int * T \mid (E)$

int * int + int

Step by step construction of a parse tree by bottom up derivation

Step 2

Derivation and Parse Tree

int * int + int

int * T + int

$E \rightarrow T + E \mid T$

$T \rightarrow int \mid int * T \mid (E)$

T

int * int + int

Step by step construction of a parse tree by bottom up derivation

Step 3

Derivation and Parse Tree

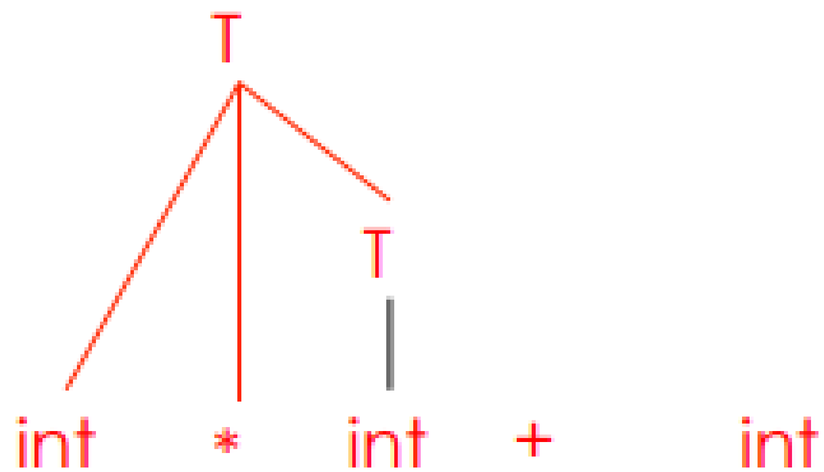
int * int + int

int * T + int

T + int

$$E \rightarrow T + E \mid T$$

$$T \rightarrow int \mid int * T \mid (E)$$



Step by step construction of a parse tree by bottom up derivation

Derivation and Parse Tree

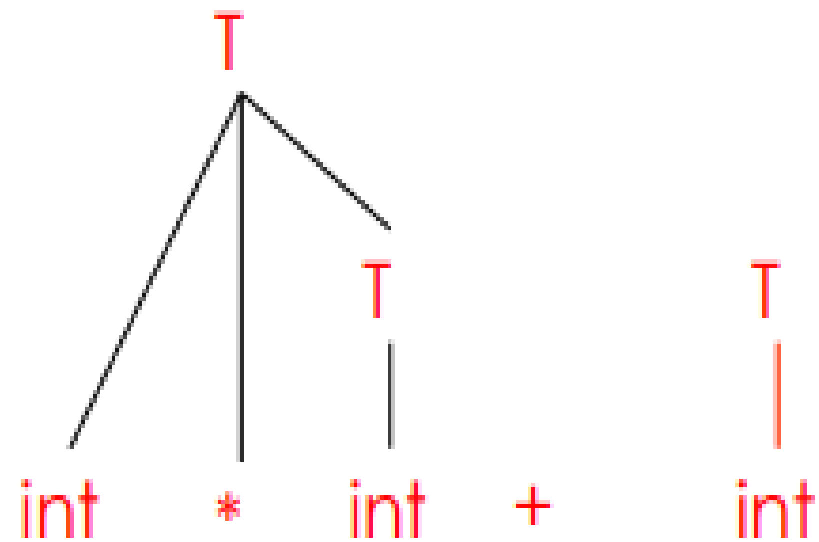
Step 4

int * int + int

int * T + int

T + int

T + T



$E \rightarrow T + E \mid T$

$T \rightarrow int \mid int * T \mid (E)$

Step 5

Derivation and Parse Tree

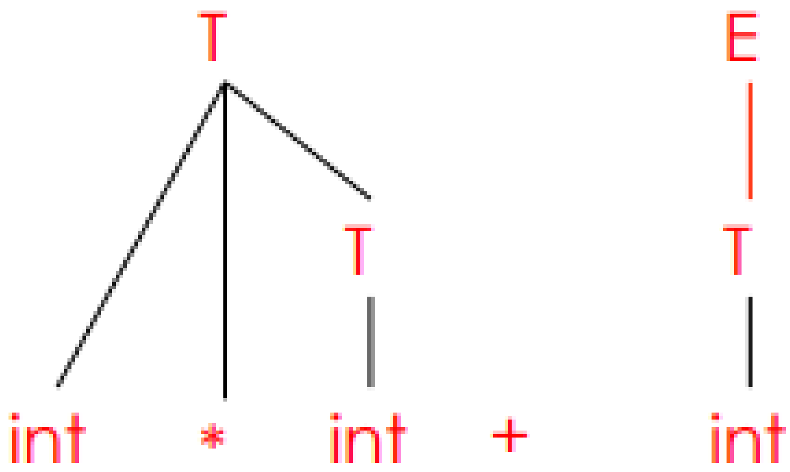
int * int + int

int * T + int

T + int

T + T

T + E



$E \rightarrow T + E \mid T$

$T \rightarrow int \mid int * T \mid (E)$

Step 6

Derivation and Parse Tree

int * int + int

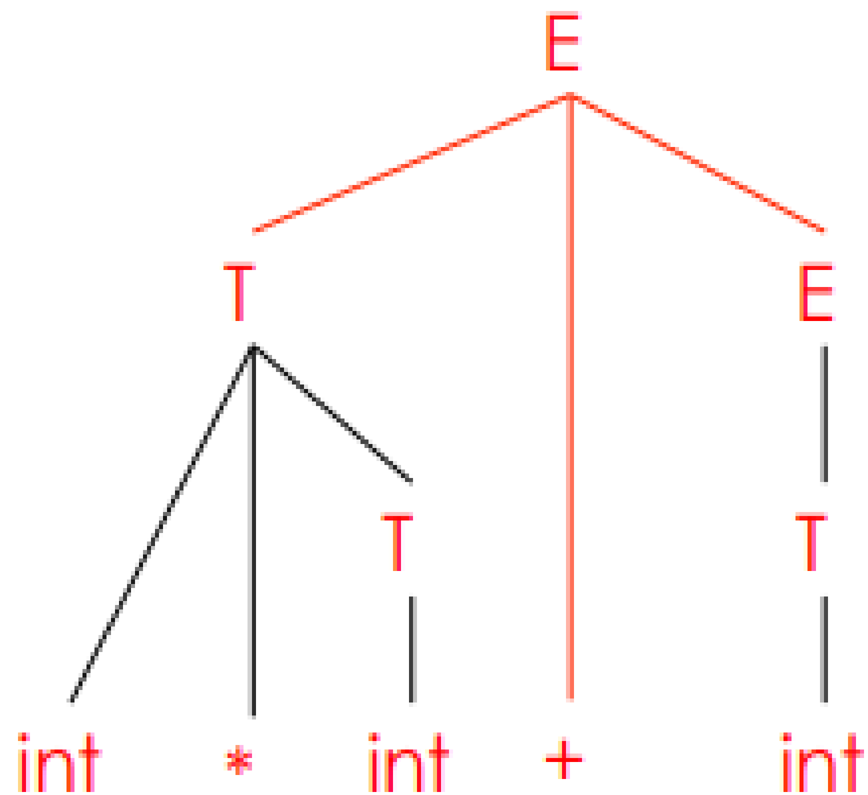
int * T + int

T + int

T + T

T + E

E



$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

Derivation eg

$S \rightarrow TW$

$T \rightarrow U\mathbf{c}$

$U \rightarrow \mathbf{a}U\mathbf{cc} \mid V$

$V \rightarrow \mathbf{b}V \mid \epsilon$

$W \rightarrow \mathbf{d}W \mid \epsilon$

Given the string **abbcccd**

Show the **leftmost derivation**. (In leftmost derivation, the leftmost nonterminal is replaced repeatedly)

Show the **rightmost derivation** (in rightmost derivation, the rightmost nonterminal is replaced repeatedly.

Leftmost Derivation eg

$S \rightarrow TW$

$T \rightarrow U\mathbf{c}$

$U \rightarrow \mathbf{a}U\mathbf{cc} \mid V$

$V \rightarrow \mathbf{b}V \mid \epsilon$

$W \rightarrow \mathbf{d}W \mid \epsilon$

Given the string **abbcccd**

Show the leftmost derivation. (In leftmost derivation, the leftmost nonterminal is replaced repeatedly)

$S \rightarrow TW$
$\rightarrow UcW$
$\rightarrow aUcccW$
$\rightarrow aVcccW$
$\rightarrow abVcccW$
$\rightarrow abbVcccW$
$\rightarrow abbcccW$
$\rightarrow abbcccdW$
$\rightarrow abbcccd$

Leftmost Derivation eg

$S \rightarrow TW$

$\rightarrow$ UcW

$\rightarrow$ aUcccW

$\rightarrow$ aVcccW

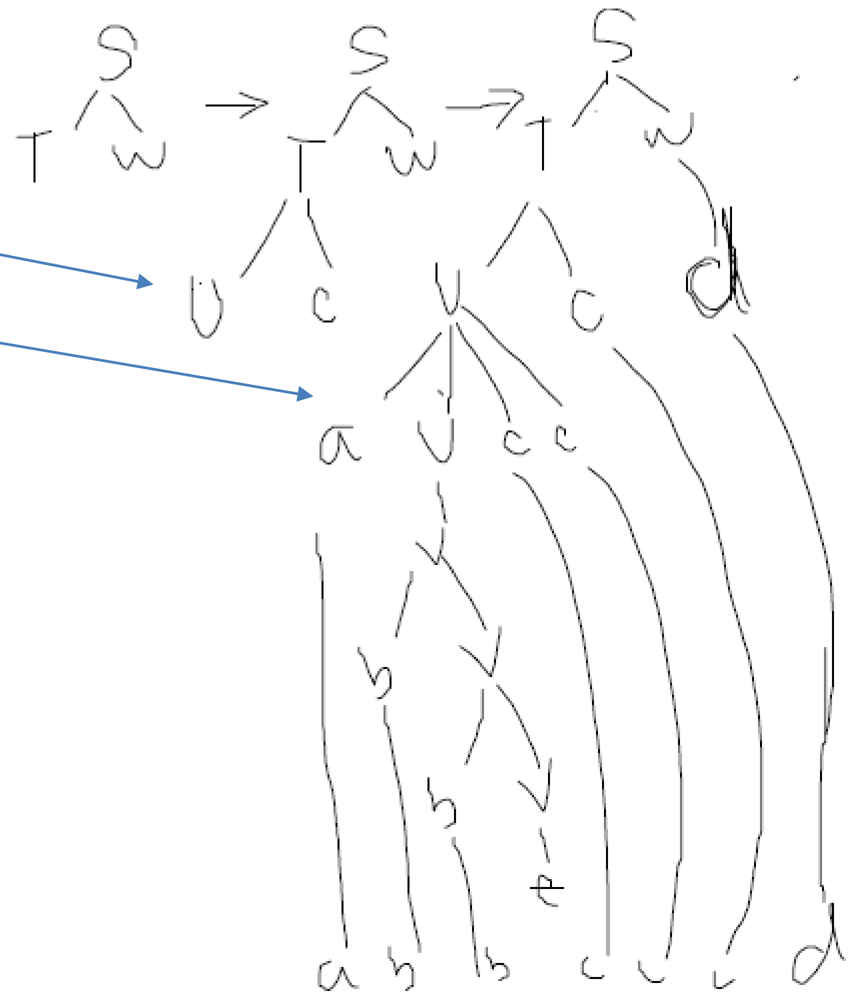
$\rightarrow$ abVcccW

$\rightarrow$ abbVcccW

$\rightarrow$ abbcccW

$\rightarrow$ abbcccdW

$\rightarrow$ abbcccd



Rightmost Derivation eg

$S \rightarrow TW$

$T \rightarrow U\mathbf{c}$

$U \rightarrow \mathbf{a}U\mathbf{cc} \mid V$

$V \rightarrow \mathbf{b}V \mid \epsilon$

$W \rightarrow \mathbf{d}W \mid \epsilon$

Given the string **abbcccd**

Show the rightmost derivation. (In rightmost derivation, the rightmost nonterminal is replaced repeatedly)

$S \rightarrow TW$

$\rightarrow$ TdW

$\rightarrow$ Td

$\rightarrow$ Ucd

$\rightarrow$ aUcccd

$\rightarrow$ aVcccd

$\rightarrow$ abVcccd

$\rightarrow$ abbVcccd

$\rightarrow$ abbcccd

Rightmost Derivation eg

S → *TW*

→ TdW

→ Td

→ Ucd

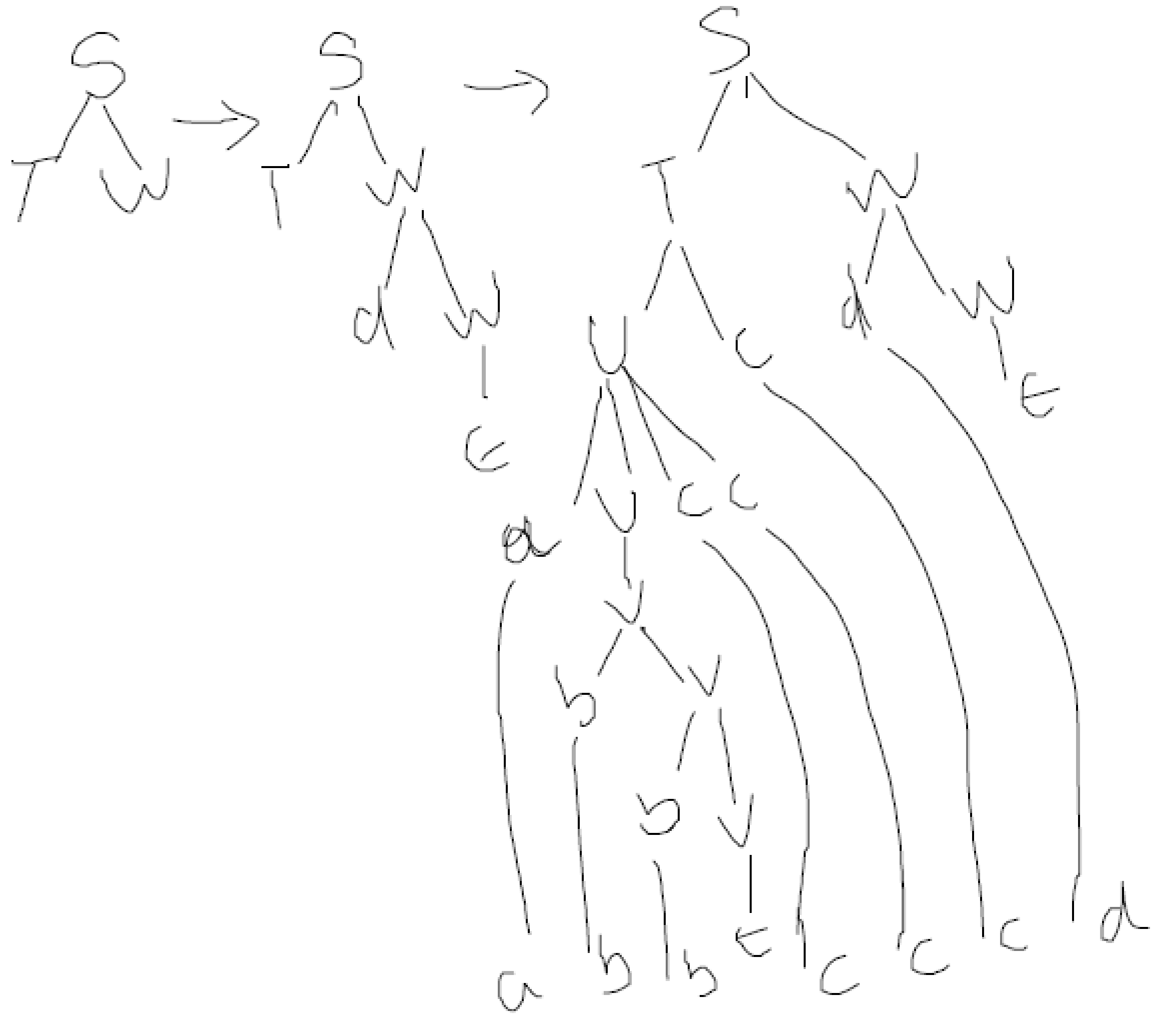
→ aUcccd

→ aVcccd

→ abVcccd

→ abbVcccd

→ abbcccd



Grammar for Expressions

- The CFG which we saw earlier : was ambiguous.

  <expr> ::= <expr> <op> <expr> | < digit>

  <op> :: + | - | * | /

  <digit> ::= 0 | 1 | 2 | …. | 9

- Revised grammar.

$$
\begin{array}{lll}
E & ::= & E + T \mid E - T \mid T \\
T & ::= & T * F \mid T \mid F \mid F \\
F & ::= & \textbf{number} \mid \textbf{name} \mid (E)
\end{array}
$$

Figure 2.6 A grammar for arithmetic expressions.

Grammar for Expressions

- Revised grammar
- Parse Tree for sentences
- 2 + 3 * 5,

$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T \,/\, F \mid F \\
F &::= \textbf{number} \mid \textbf{name} \mid (E)
\end{aligned}
$$

?

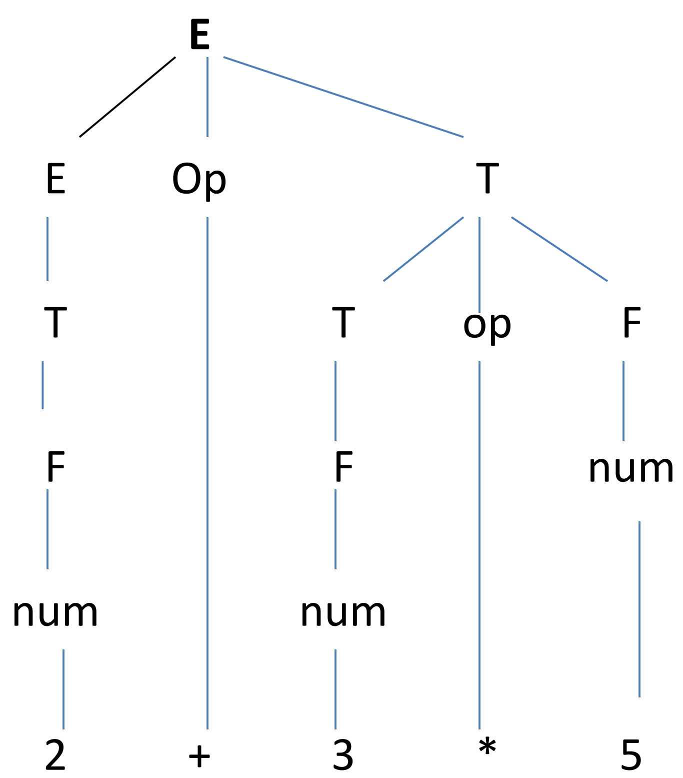
- 3* 5 + 2

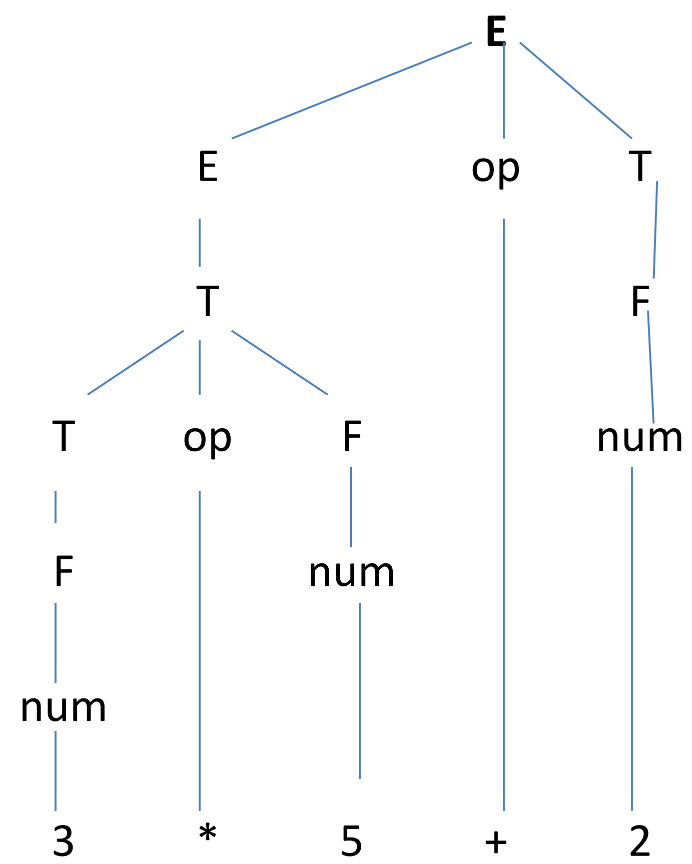
?

Grammar for Expressions

- Revised grammar
- Parse Tree for sentences
- 2 + 3 * 5, 3* 5 + 2

$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T \,/\, F \mid F \\
F &::= \textbf{number} \mid \textbf{name} \mid (\, E \,)
\end{aligned}
$$





Grammar for Expressions

- **Left recursive grammars** can handle left associativity

(Left recursive grammar is one where the non terminal on left hand side of a production appears as the first non terminal on the right hand side of the production.)

Eg. L ::= L + num | L – num | num

- **Right recursive grammars** can handle right associativity

(Right recursive grammar is one where the non terminal on left hand side of a production appears as the right most non terminal on the right hand side of the production.)

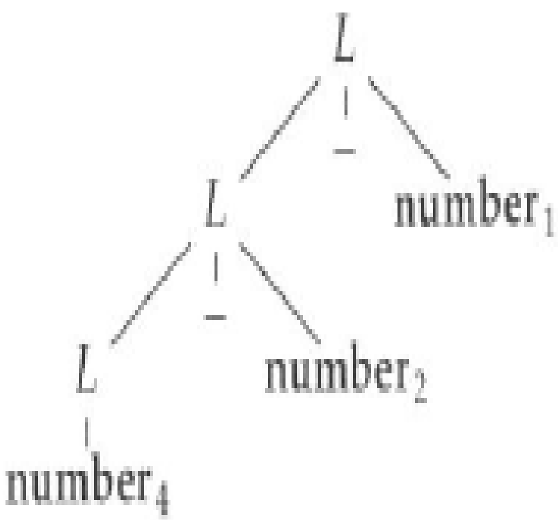
Eg. R ::= num + R | num – R | num

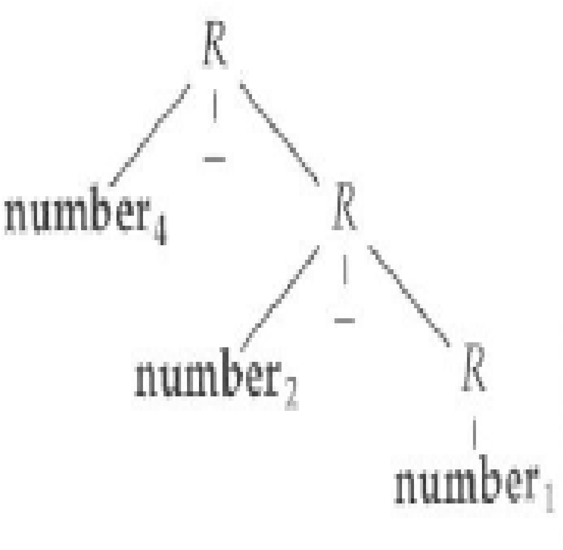
Grammar for Expressions

- Construct the parse tree : 4 – 2 – 1

L ::= L + num | L – num | num

R ::= num + R | num – R | num





Grammar for Expressions

Lowest Precedence

assignment	=
logical or	||
logical and	&&
inclusive or	|
exclusive or	^
and	&
equality	== !=
relational	< <= >= >
shift	<< >>
additive	+ -
multiplicative	* / %

Highest Precedence

$:=$	right associative
$+\ -$	left associative
$*\ /$	left associative

$$
\begin{array}{lll}
A & ::= & E := A \mid E \\
E & ::= & E + T \mid E - T \mid T \\
T & ::= & T * F \mid T / F \mid F \\
F & ::= & (E) \mid \textbf{name} \mid \textbf{number}
\end{array}
$$

Variants of Grammars

- Extended BNF
  - Empty sequence. Eg. C statements

$$\langle statement\text{-}list \rangle \ ::= \ \{ \ \langle statement \rangle \ ; \ \}$$

$$\begin{aligned} \langle statement\text{-}list \rangle \ &::= \ \langle empty \rangle \\ &\mid \ \langle statement \rangle \ ; \ \langle statement\text{-}list \rangle \end{aligned}$$

$$\langle real\text{-}number \rangle \ ::= \ [\langle integer\text{-}part \rangle] \ . \ \langle fraction \rangle$$

$$\begin{aligned} \langle real\text{-}number \rangle \ &::= \ \langle integer\text{-}part \rangle \ . \ \langle fraction \rangle \\ &\mid \ . \ \langle fraction \rangle \end{aligned}$$

Variants of Grammars

- Extended BNF
  - Braces, { and }, represent zero or more repetitions.
  - Brackets, [and], represent an optional construct.
  - A vertical bar, | represents a choice.
  - Parentheses, (and), are used for grouping.

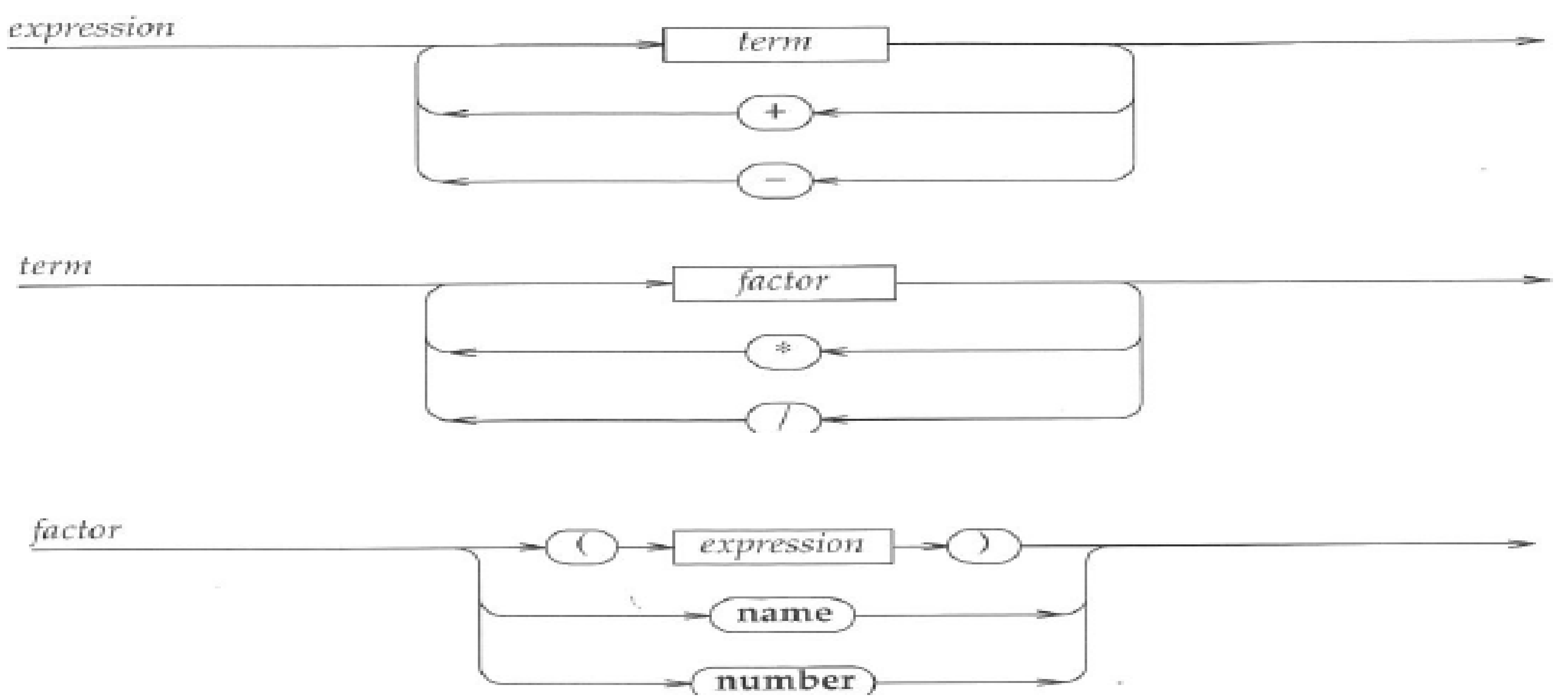
$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T \,/\, F \mid F \\
F &::= \textbf{number} \mid \textbf{name} \mid (E)
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathit{expression} \rangle &::= \langle \mathit{term} \rangle \; \{ \; (+|-) \; \langle \mathit{term} \rangle \; \} \\
\langle \mathit{term} \rangle &::= \langle \mathit{factor} \rangle \; \{ \; (*|/) \; \langle \mathit{factor} \rangle \; \} \\
\langle \mathit{factor} \rangle &::= \text{'('} \; \langle \mathit{expression} \rangle \; \text{')'} \mid \textbf{name} \mid \textbf{number}
\end{aligned}
$$

Variants of Grammars

- Syntax Chart

$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T / F \mid F \\
F &::= \textbf{number} \mid \textbf{name} \mid (E)
\end{aligned}
$$



Reference

- Chapter 2, Ravi Sethi, "Programming Languages: Concepts and Constructs" 2nd Edition by Addison Wesley, 2006.

innovate achieve lead

BITS Pilani
Dubai Campus

Thank You!