# Ch3. Syntax and Semantics

**Syntax** – specifies how programs in the language are built up;

**Semantics** – specifies what the program means;

Same syntax can have different semantics in different world (language).

ex) syntax: DD/DD/DDDD

string (program): 01/02/2017

possible semantics: Jan. 02, 2017 (USA), Feb. 01, 2017 (India?)

Lexical syntax – spelling of words;

notation: regular expression, ex) a*, ab+, (a|b)+, . . .

sample words: a, aa, aaa, ab, abb, abab,. . .

grammar (CFG)

notation: BNF (Backus-Naur Form)

EBNF (extended Backus-Naur Form)

syntax chart (graphical)

ex) Lexical syntax

b * b – 4 * a * c → Lexical analyzer → token stream (terminals)

id op id op num op id op id

b * b – 4 * a * c

lexeme

## Expression notation

Expressions such as a+b*c have been in use for centuries and were a starting point for the design of programming languages.

- **prefix** notation – easy to decode

  *evaluate right-most op first*

  \+ 3 * 5  7 = 38

  * +  20  30  60 = 3000

  ex) read(x);  max(x, y);  min(x, y, z);  //operation first, then data

- **postfix** notation – evaluated by stack

  *evaluate left-most op first*

  3  5  7  *  +  = 38

  20  30  +  60  *  = 3000

| |
|---|
| + |
| * |
| 7 |
| 5 |
| 3 |

| |
|---|
| * |
| 60 |
| + |
| 30 |
| 20 |

- **infix** notation – user friendly (easy to read)

  decoding (evaluation) needs rules for precedence & associativity;

  ( ) breaks the rules;

  *only infix can use ( )*

  *higher prec*

  ^

  *, /

  +, -

  left-assoc:

  +, -, *, /

  ex) 8/4/2 = 1

  right-assoc:

  ^

  ex) 2^3^4 = 2^81

- **Abstract syntax tree (AST)** – syntactic structure of an expression

  independent from the three notations (pre/post/infix);

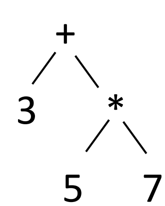
  identifies the meaningful components of each construct in the language;

  ex) infix:  3 + 5 * 7  (=38)

  prefix:  + 3 * 5 7

  postfix:  3 5 7 * +

  AST:

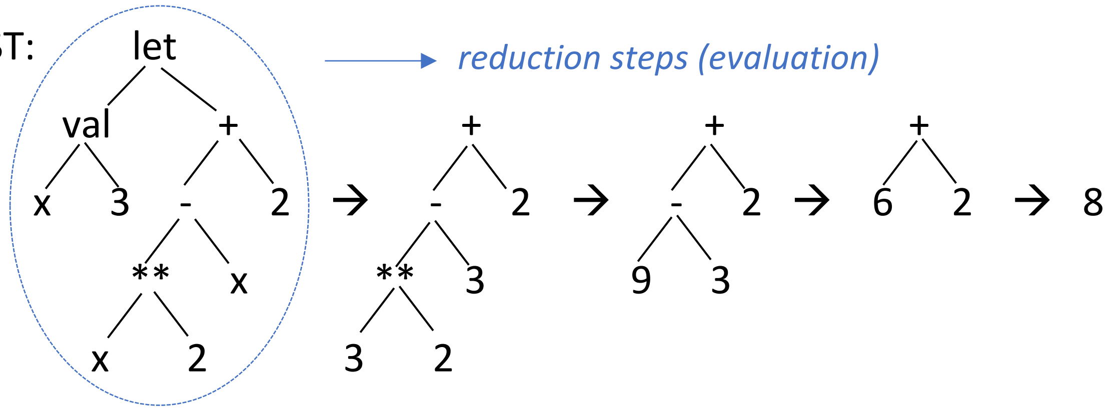


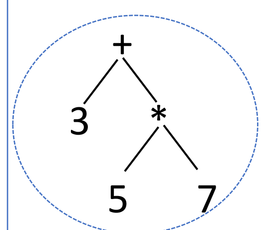
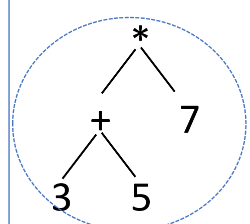
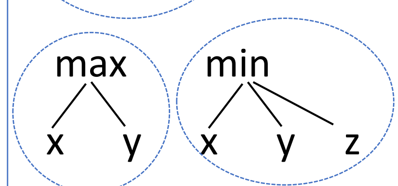

lower precedence up; higher precedence down;

ex) ML (a functional language) let construct

let val x=3 in
  x**2 – x + 2

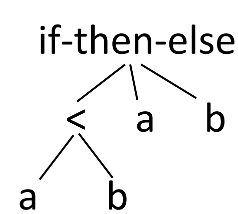
→ AST:




| ex) infix | prefix | postfix | AST |
| --- | --- | --- | --- |
| 3 + 5 * 7 =38 | + 3 * 5  7 =38 | 3  5  7 * + =38 |  |
| (3 + 5) * 7 =56<br>*only infix can use ( )* | * + 3  5  7 =56 | 3  5 + 7 * =56 |  |
| | read(x);<br>max(x, y);<br>min(x, y, z); | |  |

ex) mix-fix expression:

if  a<b  then  a  else  b



<u>Practice with:  $b^2$- 4ac</u>

## Context-Free Grammar (CFG)

is used to specify syntax of a language; infinite set of strings;

notations: BNF, EBNF, syntax chart

Formal definition:

G = (V, T, P, S), where

- V: finite set of non-terminals
- T: finite set of terminals
- P: finite set of productions; l.h.s. (V) → r.h.s. (V + T)
- S: start symbol (from V)

derivation symbol: ::=
simplified notation: →

ex)
<sentence> → <noun-phrase> <verb-phrase>
<noun-phrase> → <adjective> <noun-phrase>
<noun-phrase> → <noun>
<noun> → boy | girl
<adjective> → little
<verb-phrase> → . . . .

terminals

Q: Is “little boy” a part of a legal string in this language? → Yes

Is “boy little” a part of a legal string in this language? → No

Is “little little boy” a part of a legal string in this language? → Yes

- Original motivation of CFG: description of natural language;

ex) syntax of integer numbers (1, 123, 23, . . .)

G = (V, T, P, S), where

V = { I, D }

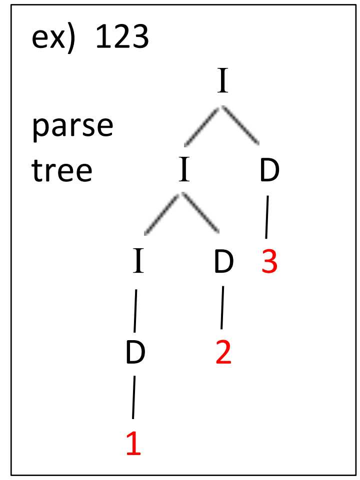
T = {0, 1, 2, 3, . . ., 9}

S = { I }

P = I → D
I → I D
D → 0
D → 1
. . . .
D → 9

$\equiv$ I → D | I D
D → 0 | 1 | 2 |. . . | 9

//total 10 productions




ex) syntax of real numbers (123.12, 123, .123, . . . )

G = (V, T, P, S), where

V = { R, I, D }

T = {0, 1, 2, 3, . . ., 9, . }

decimal point

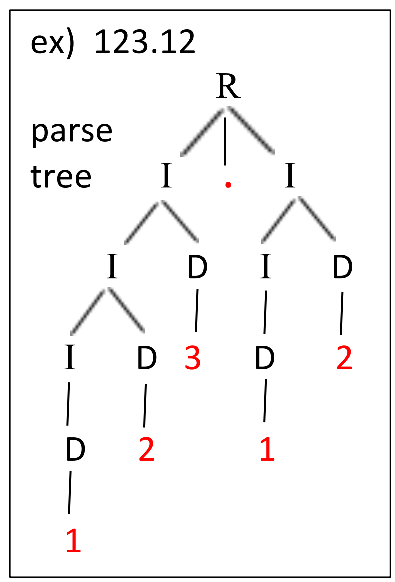
S = { R }

P = R → I . I
R → I
R → . I
I → D
I → I D
D → 0
D → 1
. . . .
D → 9

same as integer

$\equiv$ R → I . I | I | . I
I → D | I D
D → 0 | 1 | 2 |. . . | 9

//total 15 productions



ex) Language: simplified infix arithmetic expression with
no ( ), no power operation, single-digit int num.

G = (V, T, P, S), where

V = { E, T, F, Num }

T = { +, –, *, /, 0, 1, 2, 3, . . ., 9 }

S = { E }

P = E → E + T | E – T | T

T → T * F | T / F | F

F → Num

Num → 0 | 1 | 2 |. . . | 9

E – for expression
T – for term
F – for factor

**Parse tree**

depicts concrete syntax;

shows how a string is built from a given syntax (grammar);

A string (e.g., C++ program) is in a language (e.g., C++) iff it is generated by a parse tree, i.e., ∃ derivation.

The construction of a parse tree is called parsing.

Parse tree structure:

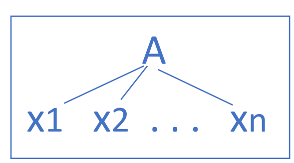
root – start symbol from grammar;

each leaf – labeled by a terminal symbol;

internal nodes – non-terminals;

if a non-terminal A has children labeled x1, x2, . . . , xn,

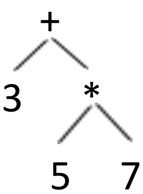
then A → x1 x2 . . . xn is a production.



- A grammar for a language is designed to reflect AST;

  the productions are chosen, s.t. parse trees are as close as possible to AST.

  ex) string: 3 + 5 * 7

  AST: 

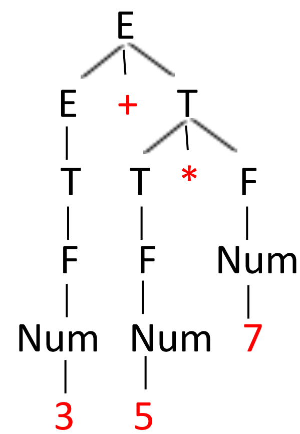
  CFG: E → E + T | E – T | T

  T → T * F | T / F | F

  F → Num

  Num → 0|1|2|. . .|9

  parse tree:

  AST – built from string only;

  parse tree – built from G and string;

  Lower subtree is evaluated first;

**<u>Derivation</u>** – snap shots of leaf nodes during building a parse tree

- right-most derivation: derive the right-most non-T first (in r.h.s. of a prod.);
- left-most derivation: derive the left-most non-T first (in r.h.s. of a prod.);

ex) G: above example, string: 3+5*7

| left-most: | right-most: |
| --- | --- |
| E → E + T | E → E + T |
| → T + T | → E + T * F |
| → F + T | → E + T * Num |
| → Num + T | → E + T * 7 |
| → 3 + T | → E + F * 7 |
| . . . . | . . . . |
| → 3 + 5 * 7 | → 3 + 5 * 7 |

## Syntactic ambiguity

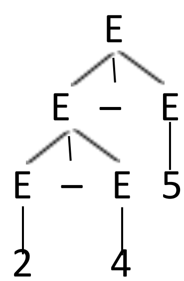
A grammar for a language is ambiguous if some string in the language has more than 1 parse trees.

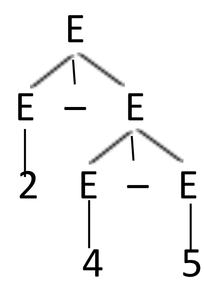
ex) G: E → E – E

| 0 | 1 | 2 |. . .| 9

string: 2 – 4 – 5

parse tree:




(2 – 4) – 5 = -7 //correct     2 – (4 – 5) = 1 //incorrect

∃ two parse trees for a given string → <u>grammar is ambiguous</u>;

*incorrect grammar*

- for writing unambiguous grammar,

  1. keep precedence, i.e., lower precedence up;
  2. keep associativity, i.e.,

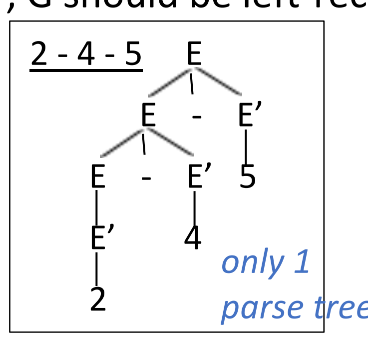
     left-assoc. op – left-recursive grammar,

     right-assoc. op – right recursive grammar;

E → E + T | T ---- +
T → T * F | F ---- *
F → Num ^ F ---- ^
. . . .

ex) since ( – ) is left-associative operator, G should be left-recursive.

E → E – E'
| E'
E' → 0 | 1 | 2 |. . . |9



<u>Associativity</u> – reflect associativity in the grammar (for unambiguous G);

left-associative op (+, -, *, /)

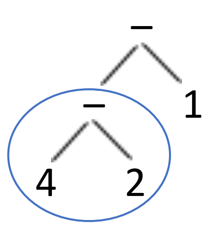
→ G should be left-recursive;

ex) +, –

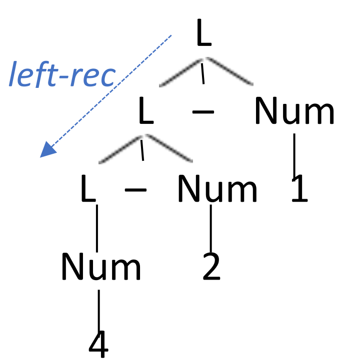
L → L + Num
    | L – Num
    | Num
Num → 1|2|. . .|9

ex. string: <u>4 – 2 – 1</u>

AST:




parse tree:




right-associative op (^, :=) assignment

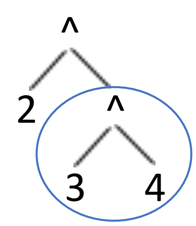
→ G should be right-recursive;

ex) ^ (power)

R → Num ^ R
    | Num
Num → 1|2|. . .|9

ex. string: <u>2 ^ 3 ^ 4</u>

AST:




2^(3^4) = $2^{81}$

parse tree: