
nline
Together we can

Chapter (3)-part1-

Syntax: the structure of the expressions, statements, and program units

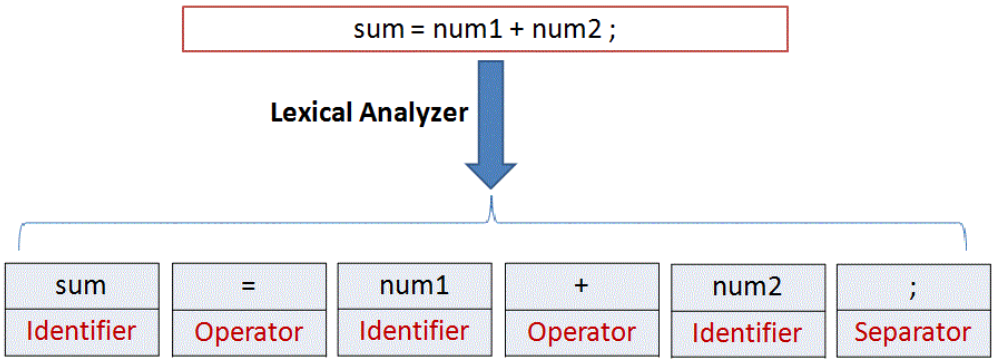
Semantics: the meaning of the expressions, statements, and program units

A sentence is a string of characters

A language is a set of sentences

A lexeme is the lowest level unit of program

A token is a category of lexemes



Token	Lexeme
identifier	sum
operator	=
identifier	num1
operator	+
identifier	num2
separator	;

Formal Definition of Languages:

Recognizers

decides whether the input strings belong to the language or not

Example: syntax analysis (in Chapter 4)

Generators

generates sentences of a language & determine if the syntax of a input sentence is correct or not by comparing it to the structure of the generator

Ex:

input : if then else

Reconizer:

"if" is word from my language ? yes ,so it's correct

"Then" is word from my language ? No ,so it's incorrect

And so on..

Generator:

"if " should be in front of "else" ? yes,it correct

BNF and Context-Free Grammars:

Context-Free Grammars (Developed by Noam Chomsky in 1950)

Backus-Naur Form (Invented by John Backus in 1959)

They describe the syntax of natural languages

BNF Fundamentals:

abstractions are used to represent classes of syntactic structures

(called nonterminal symbols, or terminals)

لو عندنا مثال زى ال If condition هتبقى

<Assign> -> if (<expr>)
<expr> -> number > number | number < number

فهنا بقول ان فى حاجه هتتحط بين القوسين و الحاجه ديه هعرفهالك لسه يعنى لسه هنفكها على عكس كلمه If و number اللى حطيناها على طول و مش فكيناها لانها هى تعتبر منتهيه فملهاش حاجه ممكن تساويها على عكس ال Expr اللى مش منتهى و ليه حاجات ممكن تساويه فبكل بساطه اى حاجه مش هنفكها تانى بنسميها terminal

و اى حاجه ممكن نفكها تانى بنسميها Nonterminal يبقى اعتبارا من كده فاكيد لازم دايما الحاجه اللى على شمال علامه الـ -> (اللى معناها يساوى) لازم تكون Nonterminal بس اللى على يمين العلامه ديه ممكن يكون Terminal or nonterminal زى مثلا لما اجى اعمل كده

z = a + 5
a = 6+7

يبقى الـ a , z هنا الاتنين Nonterminal بس الـ 5 , 7 Terminal

Terminals: lexemes or tokens

A rule:

- left-hand side (LHS) is a nonterminal
- right-hand side (RHS) is terminals and nonterminals

- Nonterminal: are often enclosed in angle brackets(<>)(Ex: <sum>)
- can have more than one RHS(Ex: <stmt> → id - id | id + id)

Grammar(a finite set of rules):

A start symbol : a special nonterminal symbol that appears in the initial rule

Example:

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<integer> ::= ['-'] <digit> {<digit>}

the symbols (-,0,1,2,3,4,5,6,7,8,9) are terminal symbols

<digit> and <integer> are nonterminal symbols.

<digit> is start symbol.

Examples of BNF rules:

1) The Grammar of (x,y,z) is:

   <ident_list> → identifier | identifier, <ident_list>

2) The Grammar of (if statement)

   <if_stmt> → if <logic_expr> then <stmt>

   <stmt> → <single_stmt> | begin <stmt_list> end

A derivation:

is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

يعنى هنمشى مع ال Rules اللى عندنا لحد ما نوصل للـ Input

Grammar:

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

Input: a=b+5

So, Derivations steps are :

<program> => <stmts> => <stmt>

=> <var> = <expr>

=> a = <expr>

=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

=> a = b + const

=> a = b + 5

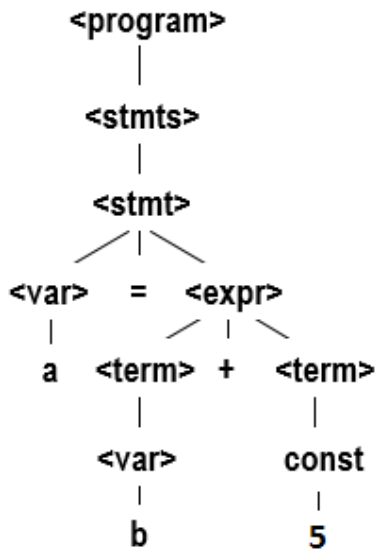
Derivation types:

1) leftmost

2) rightmost

Parse Tree:

A hierarchical representation of a derivation



Ambiguity Grammar:

generates two or more distinct parse trees

Example:

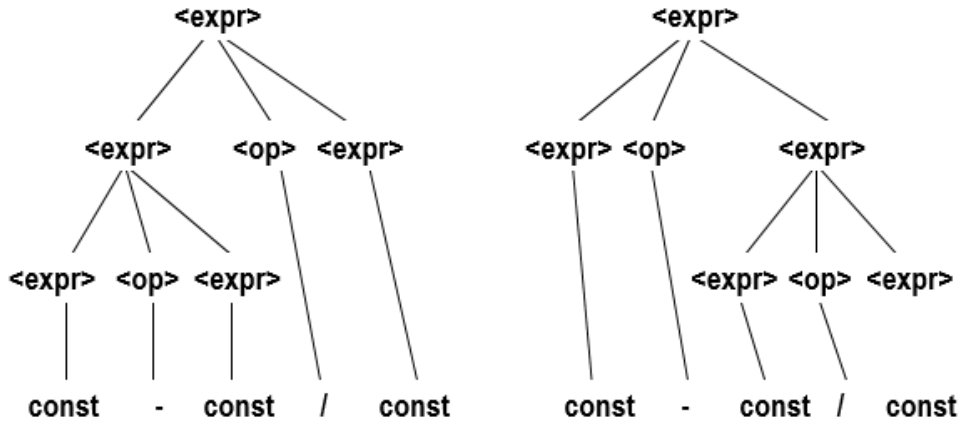
Grammar:

<expr> → <expr> <op> <expr> | const

<op> → / | -

Parse Tree:

it generates 2 parse trees so it's ambiguous grammar



But The Ambiguity can be solved by change the associativity of grammar

To make it unambiguous grammar.

An Unambiguous Grammar:

we just have one parse tree for it

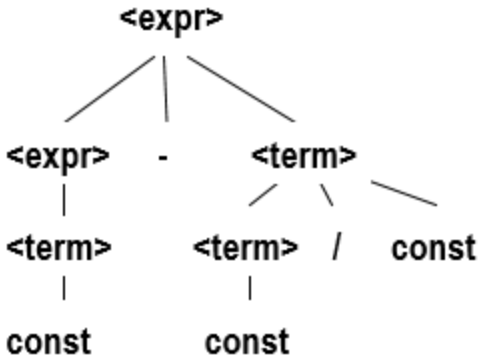
Example:

Grammar:

<expr> → <expr> - <term> | <term>

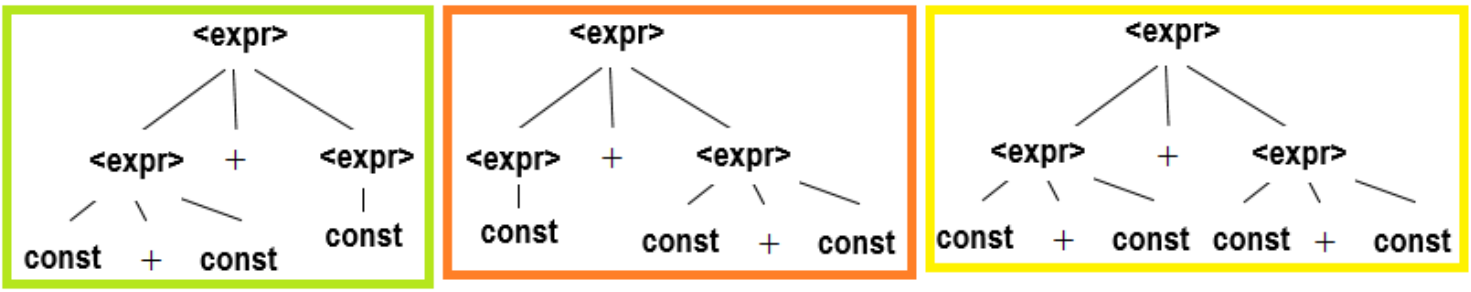
<term> → <term> / const| const

Parse Tree:

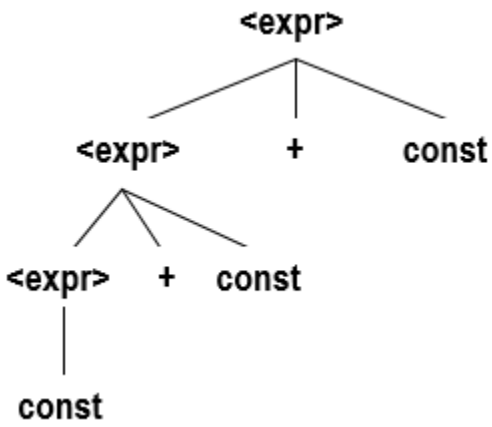


Associativity of Operators:

<expr> -> <expr> + <expr> | const (ambiguous)



<expr> -> <expr> + const | const (unambiguous)



Extended BNF (EBNF):

1) Optional parts [ ] : <proc_call> -> ident [(<expr_list>)]

1) Choose part (|) : <term> → <term> (+|-) const

3) Repetitions (0 or more) { } : <ident> → letter {letter|digit}

<ident> ->letter{digit} this will generate [Q11111134365 or Q]

BNF vs. EBNF:

BNF: <expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>

EBNF: <expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}

Recent Variations in EBNF:

1)Alternative RHSs can put on separate lines

Ex:

<expr>-> digit + digit | digit - digit

Is the same as :

<expr>-> digit + digit

<expr>-> digit - digit

2)Use of a colon(:) instead of =>

3)Use of $_{opt}$ for optional parts

4)Use of oneof for choices