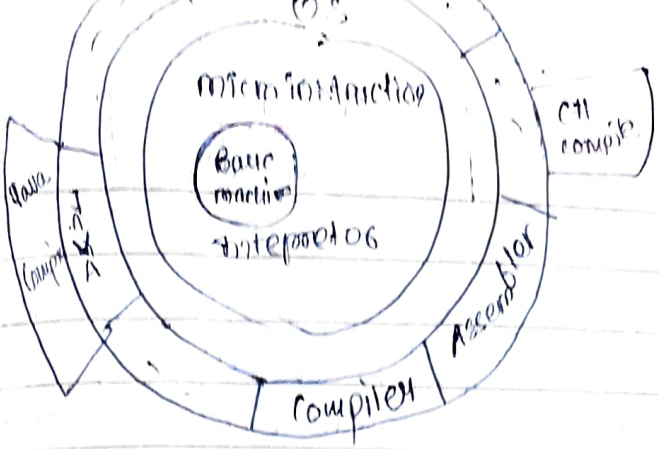


Layers → virtual computers provide interface

Implementation of programming lang

- compilers
- Pure Interpretation
- Hybrid Implementation system
- Preprocessors.

Compilation

- Lexical analysis
- Syntax analysis.
- Intermediate Code Generator
- Code optimization
- code generation.

Source program

Lexical analyzer ⇒ Gathers character into lexical unit
Identifiers, special words operators
Id1 Id2 etc

lexical unit

Symbol table

Syntax analyzer ⇒ construct parse tree. (not actual) represent syntactic str

Database of compiler process

generates assembly lang type

Parse tree

Intermediate Code generator → Optimization

Intermediate code

Code generator → to machine language. MOV ADD

machine lang.

Input data

Computers

Chapter-3 : Describing Syntax and Semantics

- Define - Syntax and semantics
- Context-free grammers (Backus-Naur form)
- Parse Trees, Ambiguity, operator precedence and Associativity, Extended Backus-Naur form
- Three formal method of describing semantics
  - operational, Axiomatic, Denotational

(1) Introduction:

Pbm: Diversity of the audience
- Initial Evaluators,
- Implementors,
- Production users.

Syntax: form of expression, statmts & prog units
Semantics: meaning.

(2) General problem of describing syntax:

Language → set of strings of chars from alpha

String of language → Sentences / Statements

lowest level syntactic units → lexemes

Language descript
- Syntax Description
- lexical specification

program → Strings of lexems.

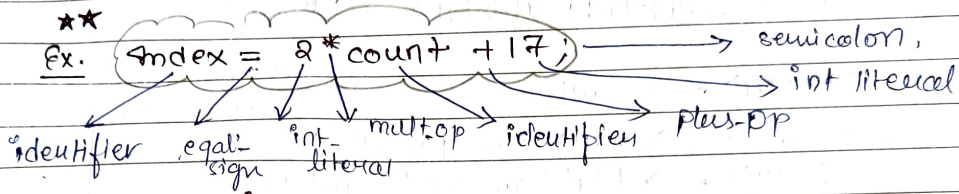
lexems — numeric literals, operators

gives lexems discription

lexems → into groups
- name of var
- methods
- classes.
→ Identifier.

token
- Identifier is token that have lexems or instance eg. sum, total
  (for var)
- Token for + (plus-op) has only one possible lexeme.

**
Ex. index = 2 * count + 17 ; → semicolon, → int literal



Black: lexems — eg. index, =, 2, *, count
Blue: tokens — eg: Identifier, equal-sign

(3) Language Recognizers

language → Recognization
         → Generation.

Eg. lang L uses Σ of char.

String of char from Σ → Recognition Device → accept / reject

↓ syntax analysis
(check for syntactic correct

(4) Language Generators

Device Generate sentences of lang → sent

- Recognizer → Trail-and-error mode
- Generators → compare with structure of the generator

(5) Formal methods of Describing Syntax

formal lang. generation mech → Grammer.
are used to describe the syntax.

↳ Backus-Naur form & Context-Free Grammars
Reused methods.

context-free
↓
Syntax

Eg.

total = subtotal1 + subtotal2

<assign> → stmt (single)
<stmt> → <var> = <expression>
<var> → total | subtotal1 | subtotal2
<expression> → <var> + <var>

Regular-grammer
↓
describe the form of tokens

Fundamentals :

- metalang - used to describe another lang.
- BNF - use abstraction.
- java assignment statement

• rule/production { <assign> → <var> = <expression>.
abstraction being defined. | defination of RHS

defination must be useful.

can be expand
abstraction : non-terminal (eg <var>, <expr>)
lexemes and tokens of the rules : terminal

BNF or Grammere — set of rules.

non-terminal → can have more possible syntactic way.

<if-stmt> → ... | ... | ...

Describing Lists

Vaulible-length list using (1,2,3--)

BNF require alternate method

int a, b, c;

* uses recursion

a rule is recursive if LHS appears on RHS

<ident_list> ⟶ identifier | identifier, <id...>

Grammer and Derivations

Grammer — generative device of defining lang

sentence generation — derivation

<program> ⟶ begin <stmt_list> end

<stmt_list> ⟶ <stmt> | <stmt> ; <stmt_list>

<stmt> ⟶ <var> = <expression>

<var> ⟶ A | B | C

<expression> ⟶ <var> + <var> | <var> - <var>
| <var>

<program> ⇒ begin <stmt_list> end

⇒ — — — —

— — — —

⇒ begin A = B + C ; B = C end

<program> → start symbol

⇒, ⟶ derives

generated sequence

Leftmost derivation ⟹ leftmost non-terminal will replaced

Rightmost , ⟶ Both Left and right also exist

derivation ⟶ until no non-term left

Example:

A = B * (A + C)

<assign> ⟶ <id> = <exp>

<id> ⟶ A | B | C

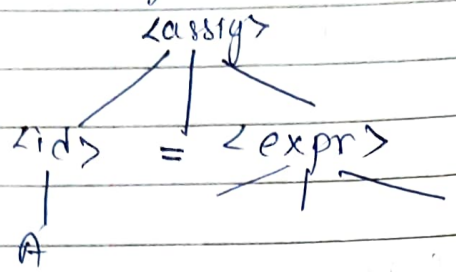
<exp> ⟶ <id> | 2 newline

<id> + <exp> | <id> * <exp> | (<exp>

Pause Tree :

Describe hierarchical stru of sentences of the

every internal nodes —— non-terminal
every leaf nodes —— terminal
every sub-tree —— one instance of an abstra



(6) Ambiguity :

Gramm generate sentential form for which two or more distinct pause tree exists.

<expr> ⟶ <expr>+<expr> | <expr> * <expr>

(7) Operator Precedence :

order of evaluation of operator

x + y * z

* More lower an operator in pause tree higher its precedence

Soln : add additional non-terminals and some new rules.

- <term>, <factor>

A = B + C * A

<assign> ⟶ <id> = <expr>
<id> ⟶ A | B | C

<expr> ⟶ <expr> + <term>
| <term>

<term> ⟶ <term> * <factor>
| <factor>

<factor> → (<expr>)
| <id>

Left most Derivation

Right most derivation

⑧ Associativity of operators

* two op with same precedence
* generally associativity of addition in compu is irrelevant

$(A+B)+C = A+(B+C)$

A=B+C+A

<assig> ⟶ <id> = <expr>

<id> ⟶ A | B | C left most

left recursive

<expr> ⟶ <expr> + <term>
| <term>

<term> ⟶ <term> * <factor>
| <factor>

<factor> ⟶ (<exp>)
| <id>

** exponentiation operator ⟶ right associative
— use right recursion

<factor> ⟶ <exp> ** <factor> | <exp>

<exp> ⟶ (<exp>) | id

⑨ Extended BNF :

Increase readability & writability

extension

- optional part of an RHS — [ ] subscript opt
- Indefinite repeti or leave it altogether { } ∅ upper limit
- Multiple-choice option (* | / | %)

→ replace by :

new line

| : not needed

| replace by "one of"

yacc

(10) Attribute Grammers :

(*) Extension of context-free grammers.

to describe type-compatibility

"int a = "value";

useful for static semantics

some characteristic is difficult to desc

- Attribute grammer - context free gram
  - attributes
  - Attribute comutation functions - how attribute value will comput
  - predicat function
    ↓ state static semantic rule.
- Attributes
  - Associated with grammer symbols
  - Similar to variables

Features :

(1) associated with each grammer symbol X is a set of attribute A(X)

E.val → F.val	E.val = F.val
E.val ↑ F.val	E.val ↓ F.val

A(X) → S(X) Synthesized Attributes ↑
→ I(X) Inherited Attributes ↓ ↔

(2) $X_0 \to X_1 \ldots X_n$, $S(X_0) = f(A(X_1), \ldots A(X_n))$

↓ Synthetic func

Same for $I_0(X_0)$

Chapter 4: Lexical and Syntax Analysis

Syntax analyzer — Application of grammers

Objective :

- Introduction to lexical Analysis
- General parsing problem
- Recursive-descent implementation technique → top-down parsers
- Bottom-up parsing → LR parsing Algorithm

Introduction :

3 diff approaches to implement programming lang
- compilation : C++, COBOL
- pure Interpretation : javascript.
- Hybrid : Java, Perl.

Most common syntax-description formalism
context free grammer / BNF

Syntax Analyzer Task : lexical and syntax analysis
- lexical → small-scale (Name, literals)
- syntax analysis → large scale (Expression etc)

Lexical analysis - pattern matches
- front end of syntax analyzer

collect chars into lexemes & assign to tokens

result = oldsum - value/100

Token	Lexemes
IDENT	result
ASSIGN_OP	=
IDENT	oldsum

State transition Diagram

Directed Graph
node → state name
arcs → Labbelled with input char. causes trans

State diagrams used for lexical analyzer are representations of a class of mathematical machines called finite automata.

used to recognize member of a class of lang. called regular language

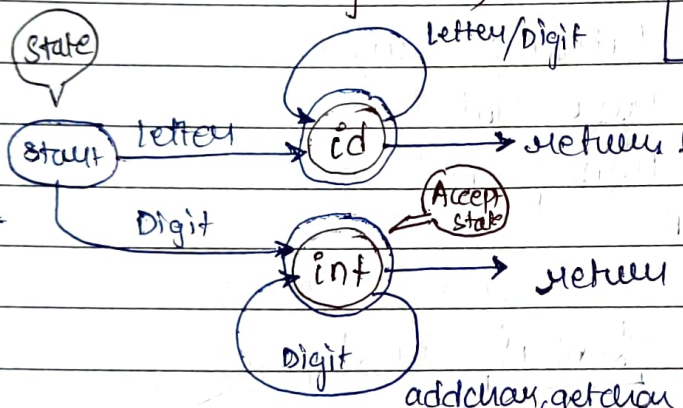
Tokens of a prog. lang.

lexical analyzer → finite automata

Example: int a;
float b;

Name	Type
a	int
b	float

Tokens	Lexem
keyword	int
identifier	a
semicolon	;



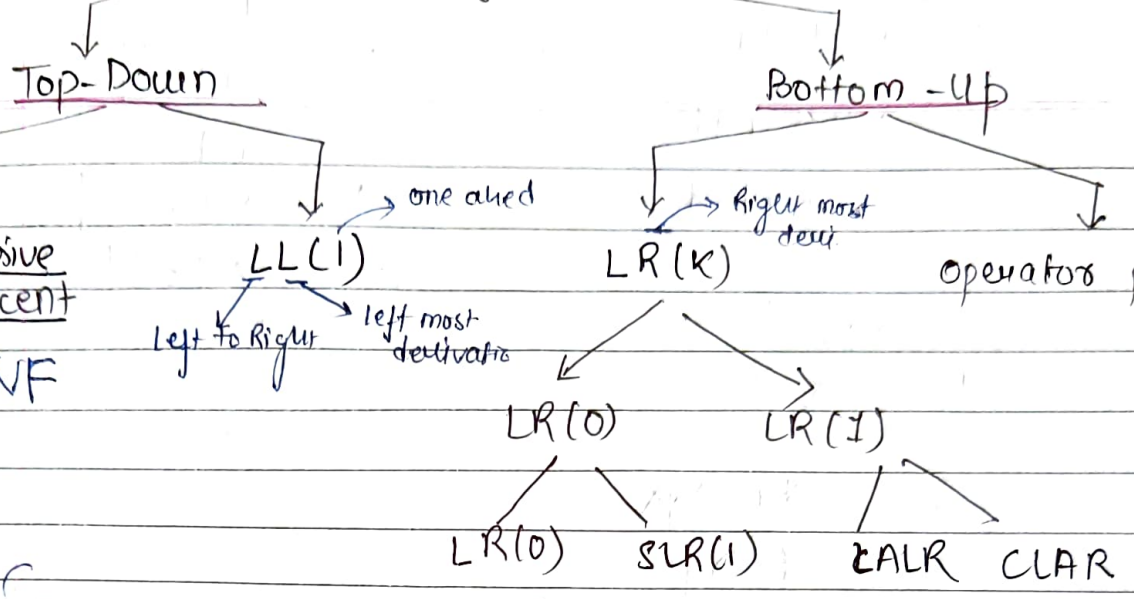
construct symbol table

var_name	Type
a	float
b	int.

Parsing: Top-down, Bottom-up.

Terminal symbol → (a, b, c - -) lowercase at start
var ← non-terminal symbol. → Uppercase start (A, B, C D-
Terminal or non-terminal → uppercase end (W, X, Y, Z)
Strings of terminals → lowercase at end of alph (w
Mixed strings → Lowercase Greek letters (α, β, γ, δ)

Parsing



first (terminal) = terminal
first (∈) = ∈

Left side

follow (start) = $ = first (A)

Right side

∈ → will never come

S → ABC

non terminal

① Recursive decent

EBNF, {} → 0 or more time

[] → once or not at all

<expr> → <term> { (+ | -) <term> }

<term> → <factor> { (* | /) <factor> }

<factor> → id | int_constant | {<expr>}

② LL Grammar class

A → A + B (Left recursion)

Direct left recus :

Rules A → Aα1 | --- | Aαm | β1 | β2 --- | βn (A → non terminal)

replace original A with

A → β1A' | β2A' - - - - | βnA'

A' → α1A' | α2A' | - - αmA' | ε

ε → Erasure rule

example

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow (E) \mid id$

$\Rightarrow$

$E \longrightarrow TE'$

$E' \longrightarrow +TE' \mid \varepsilon$

$T \longrightarrow FT'$

$T' \longrightarrow *FT' \mid \varepsilon$

$F \longrightarrow (E) \mid id$

here for E { $T \Rightarrow \beta$. , $+T \Rightarrow \alpha$. }

& for $T \longrightarrow *F \Rightarrow \alpha$

$F \Rightarrow \beta$.

Indirect left recursion

$A \longrightarrow Ba$

$B \longrightarrow Ab$

Problem for Top-bot algo.

$FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$

also, if $(\alpha \Rightarrow^* \in, \in$ is in $FIRST(\alpha))$

$*$ $\Rightarrow$ mean 0 or more derivation steps.

eg. $A \longrightarrow aB \mid bAb \mid Bb$

$B \longrightarrow cB \mid d$

$FIRST(A) = \{a\}\{b\}, \{c, d\}$.

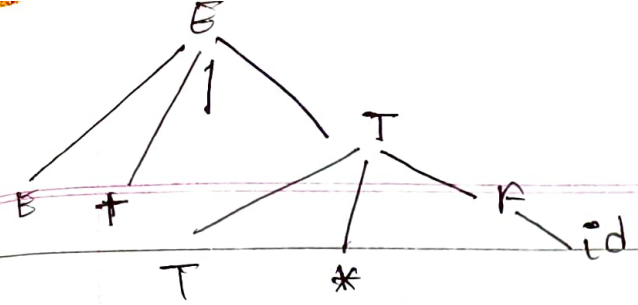
Bottom-Up Parsing

start expanding from right.

β is handel of right sentential form $\gamma = \alpha\beta w$ if only if $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow^*_{rm} \alpha\beta w$.

β is phrase if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$

β is simple phrase $S \Rightarrow^* \gamma \Rightarrow \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$



Phrase
- E+T*id
- T*id
- id

simple phrase - id

Shift reduce parsing : Parse

Grammer
$E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

String : id * (id+id)

Stack	i/p string	Action
default → $	id*(id+id)$	shift id
$id	*(id+id)$	E → id.
$E	*(id+id)$	shift *
$E*	(id+id)$	shift (
$E*(	id+id)$	shift id
$E*(id	+id)$	~~shift~~ reduce
$E*(E		
$E(E+id)		
$E*(E+E)		
$E*(E)		
$E*E		
$E		

action
- shift : otherwise x ∉ RHS (i/p to stack)
- reduce : if $x → x : RHS of grouer (Stack to input)
- accept
- error

PDA → pushdown automaton
every parser is ↑

LR parser

non recursive shift reduce bottom up parser
LR(K) → lookahead symbols
Left to Right → Right most derivation in reverse.

S_0	X_1	S_1

Action & Goto

S : State symbol
X → Grammer symbol.

$((S_0 X_1 S_1 X_2 S_2, --), a_i a_{i+1} \cdots a_n \$)$

Grammer: $S' \rightarrow S.$

$S \rightarrow AA$ — ① r_1

$A \rightarrow aA$ — ② r_2

$A \rightarrow b$ ③ r_3

paring the i/p string using tabel.

Steps	pausing stack	i/p	Action
1.	$\$0$	aabb$	swift a_3.
2.	$\$0a_3$.	abb$	swift a_3
	$\$0a_3a_3$.	bb$	Shift b_4
	$\$0a_3a_3b_4$	b$	reduce $(A \rightarrow b)$
	$\$0a_3a_3A$	b$	reduce r_2 $(A \rightarrow aA)$
	$\$0a_3A$.	b$	reduce r_2
	$\$0A$	b$	reduce r_1
	$\$0Ab_4$	$	reduce r_3

non terminal

States		Action (terminals)			Goto (variable)	
		a	b	$	A	S
① $0	0	S_3	S_4		2	1
1 $1	1			accept		
2 $2	2	S_3	S_4		5	
③ $3						
4 $4	3	S_3	S_4		6	
5 $5	4	r_3	r_3	r_3		
6 $6	5	r_1	r_1	r_1		
	6	r_2	r_2	r_2		

Regular Expression

↳ method use to represent language (L)
↳ regular language accept (FA)

like
$L = \{\varepsilon \text{ or } \lambda, a, aa, aaa \cdots\} = a^*$

Let 'R' → Regular expression over alphabet Σ if R.

- $R = \varepsilon$ $\quad L(R) = \{\varepsilon\}$
- $R = \phi$ $\quad L(R) = \{\ \}$
- $R = a$ $\quad L(R) = \{a\}$.

} prime

- $R_1 \cup R_2$ = Regular. (union) $\quad a \cup b = \{a, b\}$
- $R_1 . R_2$ = Regular (concatenable.)
- $(RE)^*$ = Regular (* → closure) $\quad a^* = \text{Reg}\{\varepsilon, a, aa, \cdots\}$
- $RE \Rightarrow (RE) \rightarrow$ regular

use recursively.

Regular Expression of finite lang

lang $\Sigma(a, b)$

no string : $\{\ \} \Rightarrow \phi$, length 0 : $\{\varepsilon\} \Rightarrow \varepsilon, \lambda$, length 1 $\{a, b\}$ $(a+b)$ ↑ union

length 2 : $\{aa, ab, ba, bb\}$
↳ $(aa + ab + ba + bb)$
↳ $(a+b)(a+b)$

length 3
↳ $(a+b)(a+b)(a+b)$

atmost 1
0, 1 $\rightarrow \{\varepsilon, a, b\} \Rightarrow (a+b+\varepsilon)$