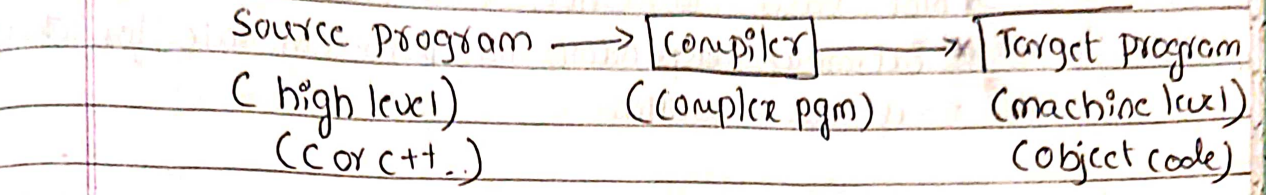
13/01/2018

Chapter-01 - Compilers

What are compilers?

Compilers are computer programs that translate one language to another. (high level to machine level)

Example: gcc, Turbo c compiler ...



Basic organization and operation of compiler should be known.

Prerequisites: Automata theory, Data structures, Discret Maths, machine architecture, assembly language.

Why compilers? History[John von neumann]

Computers perform desired computations, which require writing of sequence of codes or programs.

Initially programs were written in machine language-

* The lowest-level programming language understood only by computers.
* Machine language consists of entire numbers.

Example: C7 06 0000 0002 - // Represent instruction to move number 2 to location 0000 on Intel 8x86 processor

Writing such programs is extremely difficult and time consuming, so it is replaced by Assembly language in which instructions and memory locations are given symbolic forms.

Example: mov x, 2

An assembler translates the symbolic codes and memory locations into corresponding numeric codes of m/c language.

Ass language Improves speed and accuracy.

Disadvantages -

* Not easy to write, read and understand.
* Extremely dependent on particular m/c for which it was written.

To overcome above there was a need to write programs in such way that (1) it is independent of any particular machine and capable of itself being translated by a program into executable code (2) easy to write, read and understand.

Example $x = 2$

* FORTRAN
* Compiler — Noam Chomsky (natural language)
  * classification of languages according to complexity of grammars
  * Algorithms

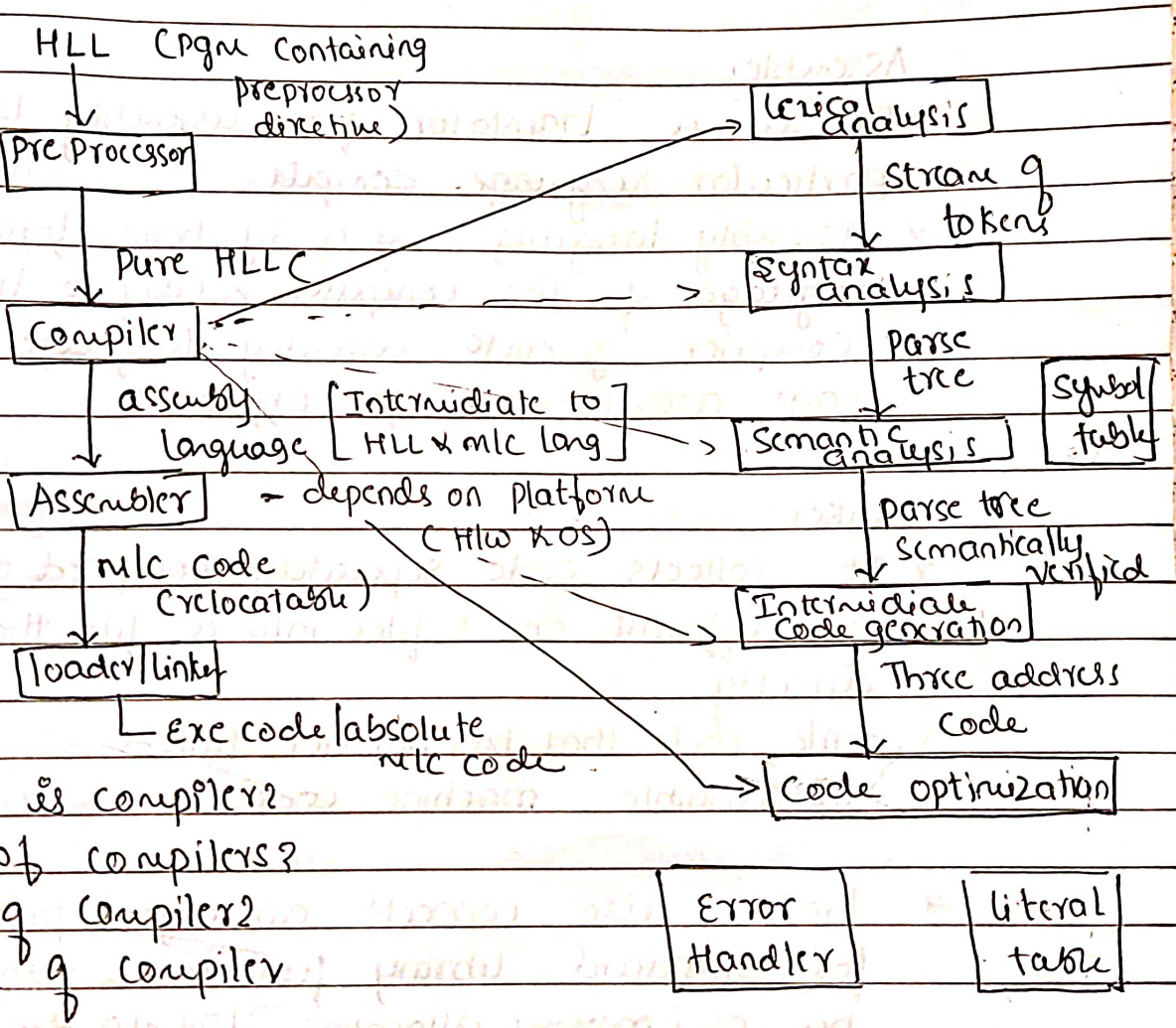
Chomsky hierarchy (structure of Natural language) made easier - compiler, partial automation.

- classification acc. complexity of grammar & power of algorithm to recognize

Type 0	unrestricted	Recursively Enumerable	Turing machine
Type 1	context sensitive	context sensitive	Linear Bounded
Type 2	Context Free	CFL	Pushdown
Type 3	Regular	Regular language	NFA, DFA, ε-NFA

Type 2 - Context Free Grammar useful for programming language (standard way for representing structure of programming language)

* Parsing (Recognition of Context Free Language) 1960-70
* Finite automata and Regular Expressions (Type 3)
* *Symbolic methods for expressing the structure of words or tokens of a programming language.
* Code optimization (code improvement) techniques.



What is compiler?
List of compilers?
Phases of compiler?
History of compiler

phases of compiler CProg

Programs related to/used together with compiler

Interpreters:

* It is a language translator, which executes the source program immediately rather than generating object code.
* Any language can be either interpreted or compiled.
* choice depends on language & circumstances under which translation occurs. Ex: BASIC, LISP
* Compilers are preferred when speed is required.
* Hybrid translators exist [blw interpreter & compiler]

Assembler:

* It is a translator for assembly language of a particular language computer.
* Assembly language is a symbolic form of machine language of the computer & easy to translate.
* Compiler generate assembly language and then rely on assembler to get objectcode.

Linker:

* It collects code seperately compiled or assembled in different object files into a file that is executable directly
* m/c code that has not yet linked
  executable machine code.

* Linker also connects an object program to the code for standard library functions & resources supplied by OS [memory allocators, i/p & o/p devices]

Loader:

* compiler, assembler or linker will produce data that is not yet ready to execute & fixed.
* Principal memory refrences were made relative to

undetermined starting location - anywhere in memory
- relocatable
* Loader resolve all relocatable addresses relative to given start address
* Use of loader makes executable code more flexible (occurs behind scene)

Preprocessors
- seperate program, called by compiler before actual translation begins.
- Preprocessor can delete comments, include other files and perform macro substitutions

Editors

Debugger

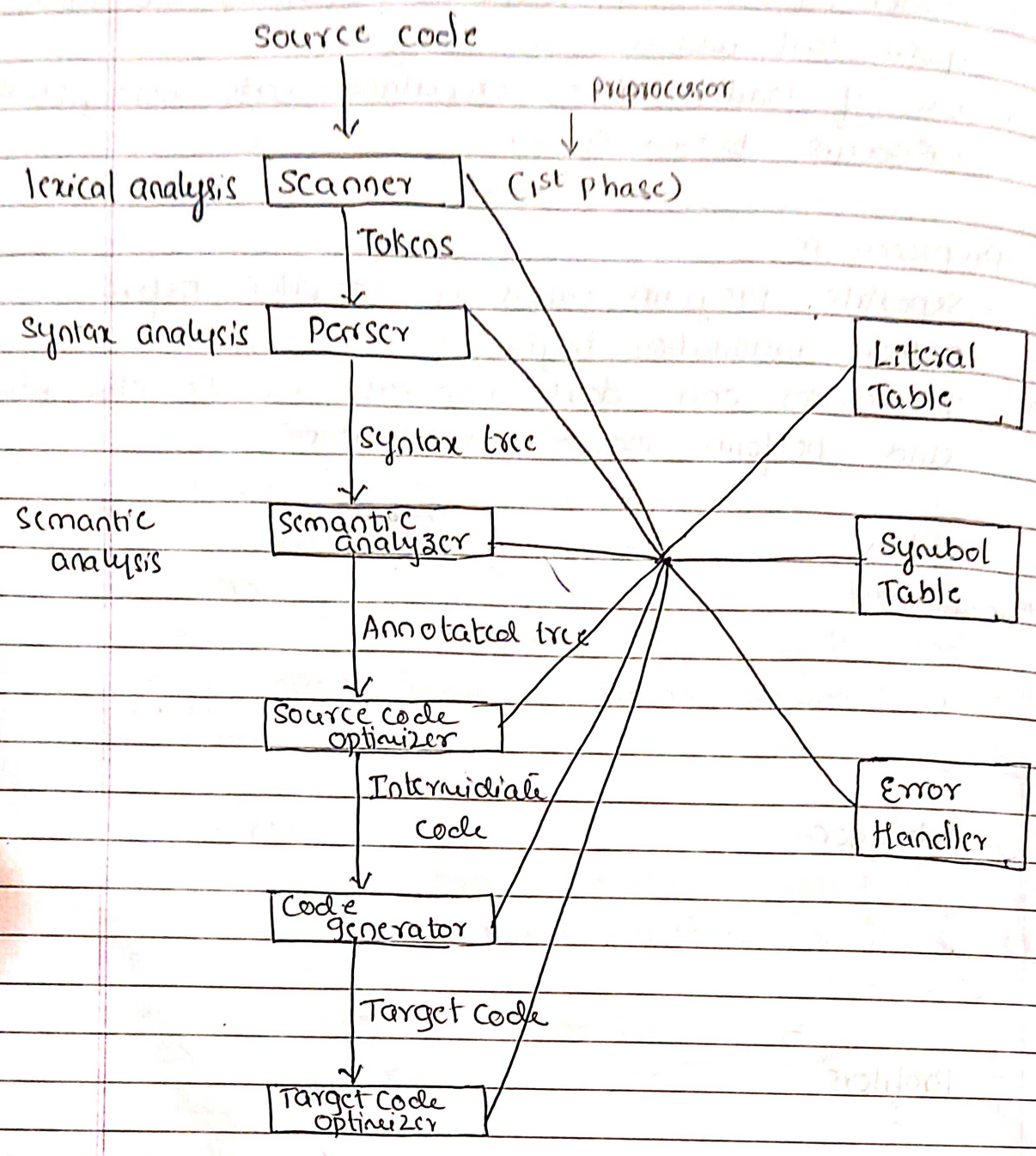
Profilers

Project Managers

Difference b/w interpreter & compiler

The Translation Process [Phases of compiler]

- Compiler consists of number of Phases - performing distinct logical operations.



The scanner: Reads the source program (stream of characters) (lexical analysis) convert it into token (meaningful unit) removes whitespaces or comments

Ex: a[index] = 4+2

a - identifier, [- left bracket, index - identifier,] - right bracket
= - assignment, 4 - number, + - Plus sign, 2 - number

id[id] = id + id
num + num

135
1350

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id.$

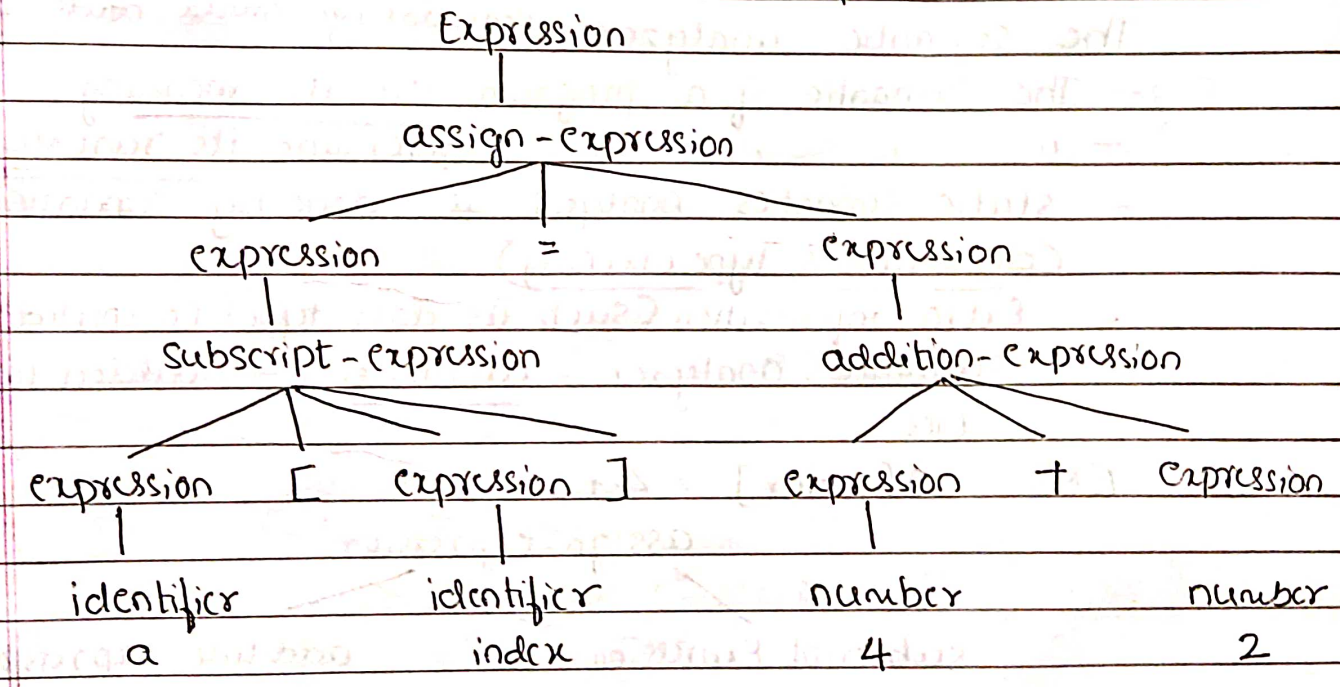
x = a+b*c

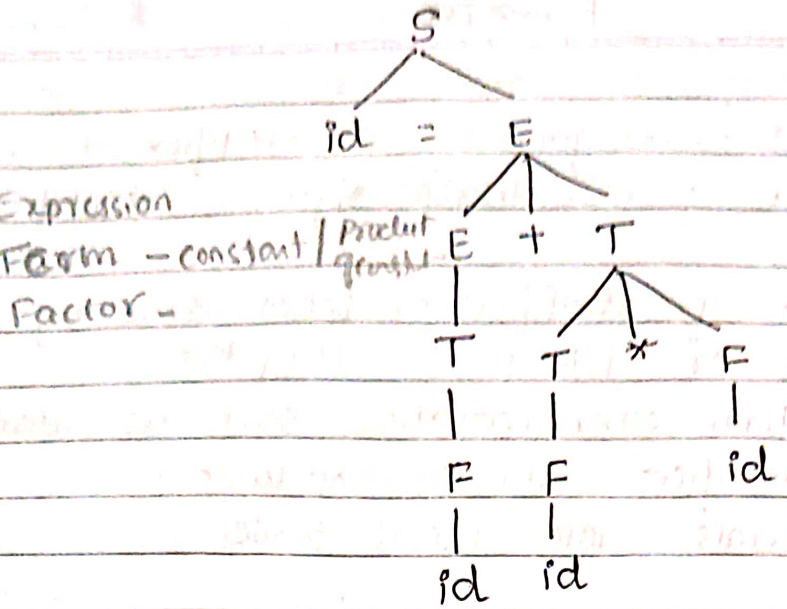
x-identifier, =- assignment, a-identifier, + - Plus sign
b-identifier, * - multiplication sign, c-identifier

* Identification of identifiers or tokens done by Regular Expressions or tokens. Ex $l(l+d)*$
* Scanner perform other operations such as identification of
  - entering identifiers into symbol table
  - " literals into literal table

The parser (syntax analysis)
- Receives source code in the form of tokens and perform syntax analysis - which determine structure of program
- Performing gramatical analysis as in natural language
- Determine structural elements of program and their relationship.
- The Parser produces parse tree or syntax tree.

Ex: a[index] = 4+2 - Expression





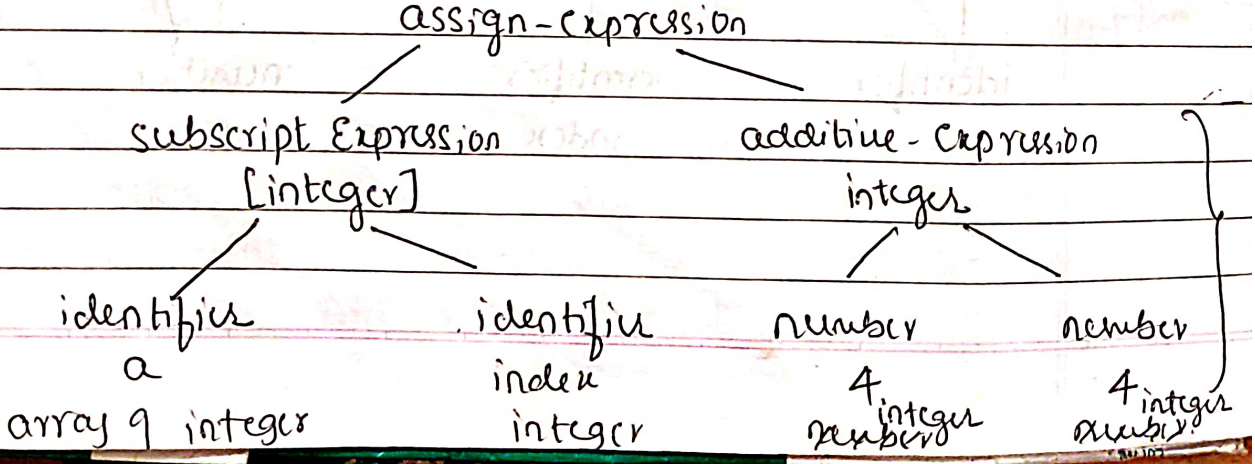
Expression
Term - constant / Product
Factor -

- Internal node of parse tree are labeled by names of Structures. they represent
- Leaves of parse tree represent sequence of tokens from input.

- Parse tree is inefficient in its representation
- So Parser generate syntax tree [condensation of information contained in Parser tree]. - abstract syntax tree

The semantic analyzer: (Redundant info. removed, delete nodes of (x))
- The semantic of a program are its meaning
- " " " " " determine its runtime behaviour.
- static semantics Analysis is done by semantic analyzer (declarations, type checking)
- Extra information (such as data type) computed by semantic analyzer - attributes. - added to parse tree

Ex a[index] = 4 + 2



prior information gathered before analysis
a - array of integer values with subscript from subrange of integers
index - integer value.

semantic analyzer
- Annotate syntax tree with types of all sub-expressions & checks that assignment makes
- declare a type mismatch error
- Result of annoted tree shown in Figure
- static semantics, dynamic semantics [salary 10% to 20%]

int a,c;
float b;
c = a+b;

The source code optimizer:
- Compiler includes code improvement or optimization steps after semantic analysis.
- Code improvement based on source code
- Code optimization depend on compilers.
- Example the expression 4+2 can be precomputed by compiler to result 6. [constant folding]
- Can be performed directly on (annoted) syntax tree by collapsing right-hand subtree of root.
- Three-address code (standard), P-code (Pascal compilers)

RHS - number 6 integer (collapsed to)

Ex t = 4+2
a[index] = t

- Intermidiate code [code representation intermidiate b/w source and object code]

The code generator
- Receives intermidiate code [IR] and generates code for target machine.
- Properties of target machine become important
- Representation of data & instruction play major role. [how many bytes or words variable of int & float data type occupy in memory

prior information gathered before analysis
a - array of integer values with subscript from subrange of integers
index - integer value.

semantic analyzer
- Annotate syntax tree with types of all sub-expressions & checks that assignment statements declare a type mismatch error
- Result of annoted tree shown in Figure
- Static semantics, dynamic semantics [salary 10% to 20%]

int a,c;
float b;
c = a+b;

The Source code optimizer:
- Compiler includes code improvement or optimization steps after Semantic analysis.
- Code improvement based on source code
- Code optimization depend on Compilers.
- Example the expression 4+2 can be precomputed by compiler to result 6. [constant folding]
- Can be performed directly on (annotated) syntax tree by collapsing right-hand subtree of root.
- Three-address code (standard), P-code (Pascal compilers)

RHS - number 6 integer (collapsed to)

Ex t = 4+2
a[index] = t

- Intermidiate code [code representation intermidiate blw source and object code]

The code generator
- Receives intermidiate code [IR] and generates code for target machine.
- Properties of target machine become important
- Representation of data & instruction play major role. [how many bytes or words variable of int & float data type occupy in memory]

Target code generator.

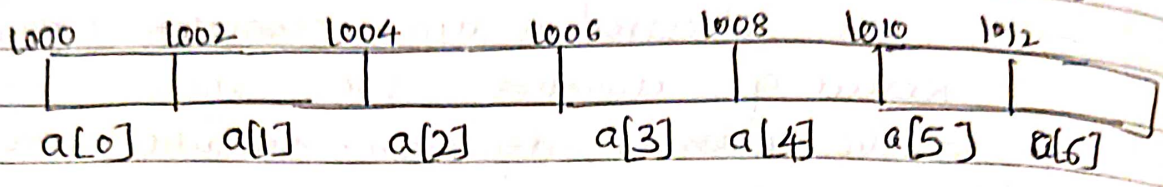
Input : IR (Three code address)

Process: generate code → assembly language.

Target code

```
MOV  R0, index
MUL  R0, 2
MOV  R1, #a
ADD  R1, R0
MOV  *R1, 6
```

Suppose a[5] the
R0 = 5 , a = 1000
R0 = 10 [integer 2 byte instr]
R1 = 1000
→ R1 ← 1010 ← 6



Target code Optimizer

* improve target code
  - choose addressing modes
  - replace slow instruction by faster one
  - Eliminating redundant or unnecessary operations.

Improvement for the above code

* use Shift instruction instead of multiplication [second line] [Expensive in terms of execution time]
* use powerful addressing mode as indexing addressing.

Multiplication addressing.

```
MOV  R0, index
SHL  R0
MOV  #a[R0], 6
```

R0 = 5
0101 = 5
↓ —— SHL
01010 = 10

Major Datastructures in a compiler

* The interaction blw the algorithms used by compiler and data structures that support these phases is strong.
* Ideally a compiler should compile a program in time proportional to size of program i.e O(n), n- pgm size
* Data structures needed by the phases as part of their operation and that serve to communicate information among the phases.

Tokens [lexeme - is an actual character sequence (lexeme)]

* Token is a sequence of characters that can be treated as a unit in the grammar of programming languages.
* In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations are considered as tokens. (In C six type of tokens.)
  Ex: int value = 100; [keyword, constant, identifier, string, special symbol, operator]
  Contains tokens int (keyword), value (identifier), = (operator), 100 (constant), ; (symbol)
* The scanner represent tokens of source language as a value of an enumerated data type
* Example: 1) sum = 3+2;
  Tokens: sum, =, 3, +, 2, ;

  2) if (x1 * x2 < 1.0) {
  Tokens: if, (, x1, *, x2 < 1.0) {

+ve int "+"? [0-9]+
-ve int "-" [0-9]*

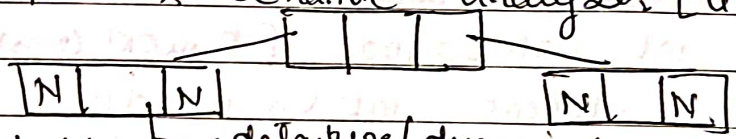
Table 1.1

Token	Lexemes	Pattern (Regular expression - Notation used to describe all tokens)
Const if	if	if
relation	<, <=, =, <>, >, =>	<|<=, =, <>, >, =>
id	Pi, count, D2	(letter (letter + digit)*)
num	3.1426, 0.6	numeric constant [0-9]*
literal	"Core dumped"	Char b/w " "

Example : Const Pi = 3.1416
Pi is a lexeme for the token "identifier"

Parser

Syntax Tree [Tree representation of the abstract syntactic structure of source code)
* Pointer based structure, dynamically allocated
* Syntax tree represented using single variable pointing to root node
* Each node represent information [record] collected both by parser & semantic analyzer [linked list / array]



Symbol table
* This data structure keeps information associated with identifiers : functions, variables, constants and data types.
* Symbol table interacts with every phase of compiler.
  - Scanner, parser, or semantic analyzer enters identifier
  → Semantic analyzer will add data type and other information.
  - optimization and code generation phase will use the information provided by symbol table
  - Symbol table is accessed frequently - access operations need to be efficient (Constant - time operations) For this standard data structure is hashtable O(n)

Literal table
- Stores constants and strings used in program.

- insertion & lookup are frequent.
- Literal table need not allow deletions [bez of global data]
- Constant or string appear only once in table.
- Literal table reduces the size of a program in memory by allowing reuse of constants & strings.
- Required by code generator.

Intermidiate code
- Three-address code & P-code
- Different optimizations performed
- Depending on above parameters code may be kept as an array of text strings, a temporary text file or as linked list of structures.

Temporary files
- As computers did not posses enough memory for an entire program to be kept in memory during compilation
- The above problem solved using temporary Files, to hold results of intermidiatory steps.
- Keep only enough information from earlier parts of source program to enable translation to proceed

GATE Question on lexical Analysis

```
Printf("i = %d, &i = %x", i, &i);
```

```
Printf
(
"i = %d, &i = %x"
,
i
,
&
i
)
;
```

8.2. In a compiler, keywords of a language are recognized during lexical analysis of the program.

Example:

x = a+b*c

↓

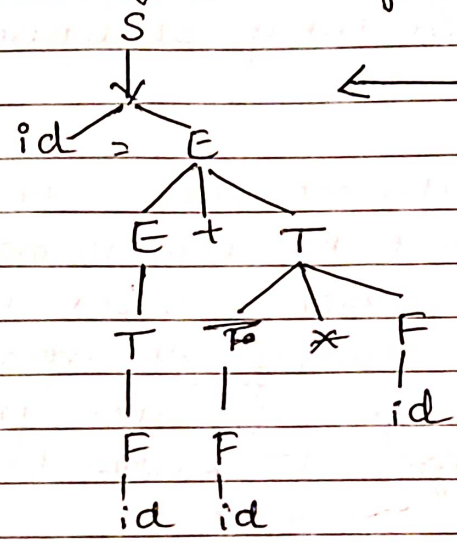
lexical Analysis

↓

id = id+id*id (Token)

↓

Syntax Analysis (syntax tree)



S → id = E

E → E+T | T

T → T*F | F

F → id

↓

Semantic analyzer [symantically Verified] (type checking, co-ercions (data type))

↓

Annoted Tree

↓

← Source code optimizer

ICG (Three-address code)

t1 = b*c

t2 = a+t1

x = t2

↓ CO

Target code generator optimized (shorter code)

t1 = b*c

x = a+t1

mlc independent ↓ mlc dependent

Writing code for assembler — depends on platform

Target code (Assembly level) Generation.

need not be save

Move → MUL R1, R2 — a = R0
ADD R2, R0 — b = R1
MOV a, R2 — c = R2

② Position = initial + rate * 60

$ gcc -Wall filename.c -o filename

-Wall - enable all compiler's warning messages - generate better code

-o - specify output name (if not specified use a.out)

- Pre-processing → Removal of comments, macro exp. eg included files, conditional compilation
- compilation
- Assembly
- Linking

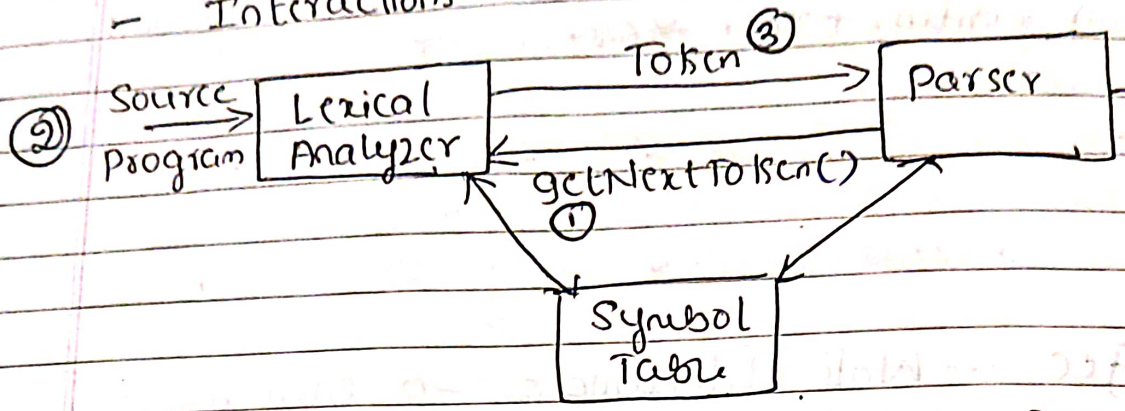
$ gcc -Wall -save-temps filename.c -o filename

filename.i
filename.s - Assembly level instruction
filename.o - mle level instruction

$ size filename.o $ size filename

Role of Lexical Analyzer

- Lexical analyzer when identifies a lexeme as identifier, it needs to enter that lexeme into symbol table.
- Interactions



* Other Functionalities of Lexer are
  - stripping out comments, white spaces (blank, newline tab or other)
  - Correlating error messages generated by compiler with source program.
    1. — assosiating line number with error message as lexer keep track of new line character
    — source pgm copied to insert error message
  - cascading of two processes
    1) scanning - 2) lexical analysis (complex)
    simple - do not require tokenization of i/p.

Advantages of Seperating lexer & parser

* Simplicity: Atleast one of the phase will be simplified
  Ex: handling of whitespace & comments
  Efficiency: using buffering techniques for reading i/p can speed up Compiler
  portability: I/p device specific properties can be restricted to only lexer

Tokens, Patterns & Lexemes:

Token: <token name, attribute value>
(optional)

main() - not a keyword wrt compiler
- same as other functions { main() sort() }

Token name is abstract symbol representing lexical unit. [keyword, identifier]
Token names are input to parser

*Pattern: Description of the form that the lexemes of a token may take
Ex: For a keyword - the pattern is just sequence of characters that form keyword
For identifiers - complex pattern - l(l+d)*

*Lexeme - sequence of characters in source pgm that matches the pattern for a token.

Table 1.1 [Previous pages] shows example for above

Example: Printf("Total = %d \n", score)
printf & score are lexemes matching pattern for token id, & "Total> %d \n" is a lexeme matching literal..

Classes of token
* one token for each keyword
* Token for the operators either individually or in classes such as token comparision
* one Token representing all identifiers
* one or more tokens representing constants [numbers and literal strings.
* Token for each Punctuation symbol, left and right paranthesis, comma and semicolon.

Keywords	Special symbol
identifiers (main	operators
Constants	
Strings	

Attributes for Tokens

* The lexical analyzer must provide the subsequent compiler phases an additional information about Particular lexeme matched.

* So lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by token
[token name influences Parser decision
attribute value influences translation of tokens after the Parse]

* Tokens have atmost one assosiated attribute. [attribute - may be a structure consisting several pieces of information]
* identifier — information may be — its lexeme, its type and location at which it first found (to handle error message) - kept in symbol table
* Thus appropriate attribute value for an identifier is a pointer to symbol-table entry for that identifier.

Example

E = M * C **2

Symbol table

id	Value
id	Pointer to symbol table →
assig_op	-
id	* to ST
mult_op	-
id	* to ST
exp_op	-
number	2

— In practice a typical compiler instead store a character string representing constant and use an attribute value for number, a pointer to that string

used by other phases.

* Symbol table created by lexer and parser.
* Structure of symbol table is designed by compiler designer.
* Info - Name, type, location, scope, other attribute
* operations associated with symbol table
  Lookup, Insert, Modify, delete

Issues in Symbol table

1) Format - what data structure need to be used
2) Access method - " search method " " " "
3) Location of Storage - RAM or secondary memory
4) Scope

Lexical errors

- An input that can be rejected by lexer.

Types

- A character sequence that can not be scanned into any valid token is a lexical error.
- Misspelling of identifiers, keyword or operators are

Ex: if the string fi is encountered for 1st time in C program

fi (a == f(x)) . .

* Lexer can not tell whether fi is a misspelling of if keyword or an undeclared fun identifier.
* Since fi is valid lexeme for token id, the Lexer must return token id, to parser & let other phases of Compiler - Probably Parser in this case.

* When none of the patterns for tokens matches any prefix of remaining input - then lexical analyzer is unable to proceed.

— Use a simple method "panic mode" recovery

Classification of compile time errors
* lexical * Syntactic * Semantic

Unmatched patterns are deleted successively from remaining input, until lexical analyzer can find well-formed token at begining of what input is left

other error-recovery techniques are
- Delete one character from remaining input.
- Insert a missing character into remaining input
- Replace a character by another character
- Transpose two adjcent characters.

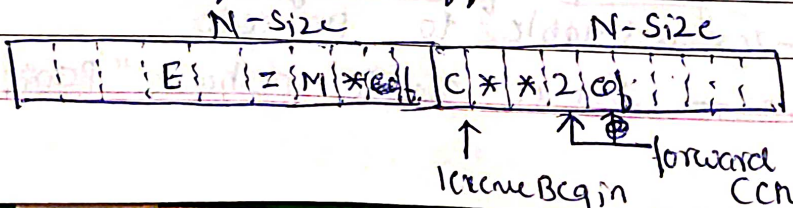
(Data structure) → Input Buffering: [Scanning Process]
* How to recognize lexeme of the input.
* We need to see one or more characters beyond next lexeme before we can be sure we have right lexeme.
* Example: we are not sure we have seen the end of an identifier until we see a character that is not a letter or digit & is not token id. In C single character operators like -, = or < could also be begining of two-character operator like ->, == or <=.
* So we have Two-buffer Scheme to handle lookaheads.

what is the necessity to look next lexeme ?

Buffer pairs:
* Special buffering techniques are required, since large number of characters must be processed during compilation.
* Techniques developed to reduce overhead required to process a single input character.

Tokenizer

→ Usage of Two buffers. - reloaded alternatively.

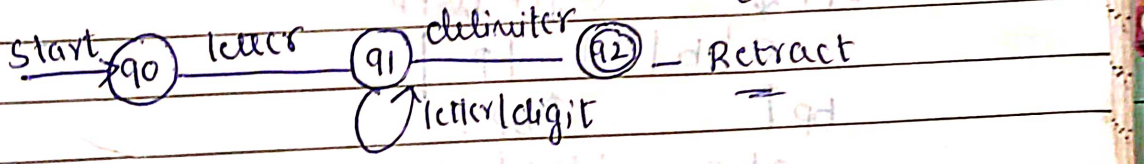


Single statement

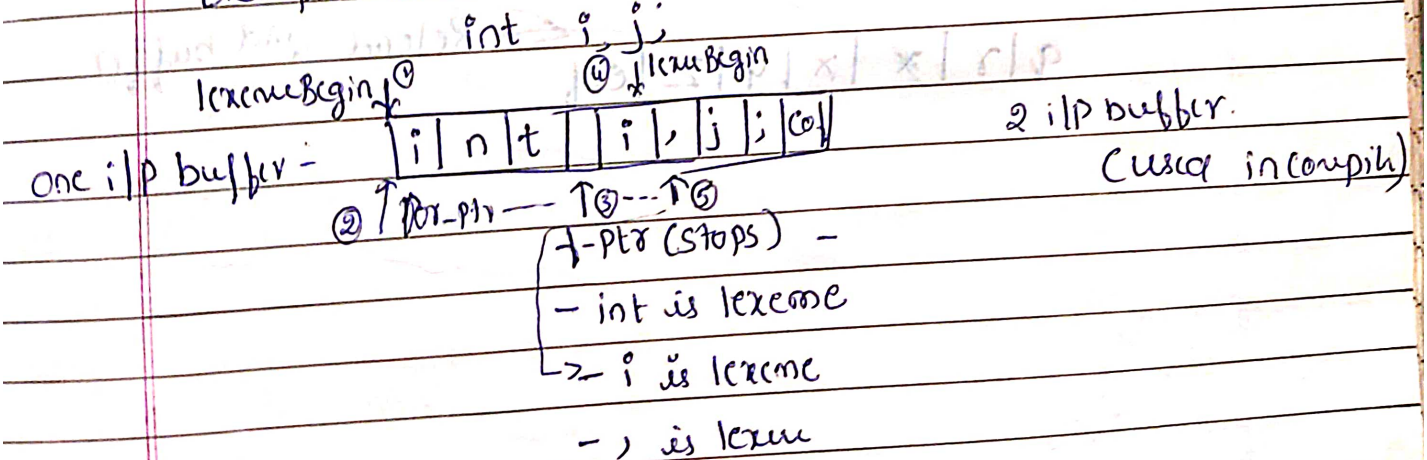
current_lexeme: set of characters blw two pointers.

* Using system read command, read N characters into buffer, rather using one system call per character.
* If fewer than N characters remain in input file, then a special character eof is read to be inserted. It is different from any other characters.
* Two pointers to the input are
  - lexemeBegin - marks begining of current_lexeme
  - forward - scans ahead until a pattern match is found.

* Once lexeme is determined, forward is set to character at its right end.
* lexeme is recorded and returned to parser.
* lexemeBegin is set to character immediately after lexeme found.
* Once we reach end of one of buffers, we must reload other buffer from inpt and move forward to begining of newly loaded buffer.



Example 2:



Sentinels

If scheme of Input Buffering is used then we must ensure

→ each time we advance forward (we have not moved off one of the buffer. if, so then must reload other buffer

i.e For each character read we make 2 test

→ one for end of buffer

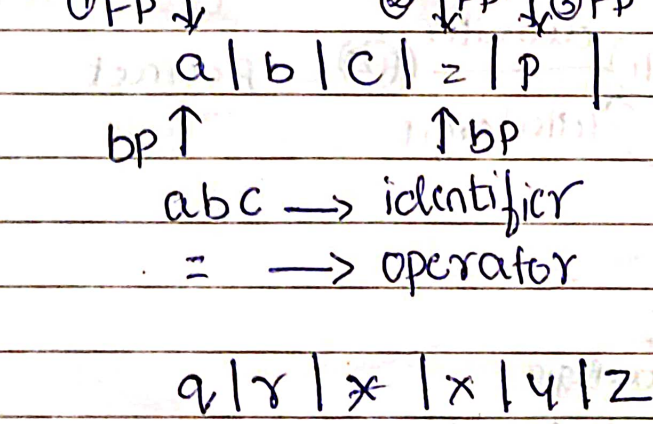
→ what character read

We can combine buffer-end test for current character if we can store sentinel character at end of buffer

Sentinel: Is a special character that can not be part of the source program and choice is the character 'eof'.

- eof is marker for the end of entire input.
- Any eof that appear other than at end of a buffer means that the i/p is at an end of

Example: abc = pqr * xyz;



Class Exercise

① int max(x, y)
int x, y; — 25 tokens
{ /* Find max of x & y */
return (x>y ? x:y);
}

Is it possible to write entire 'c' program in a line? (Caller line)

② Printf("%d Hai", xx); — 8 - token
(string literal, one operator)

③ xx — one token (longest match)

④ Entries of a symbol table xx

④ x a=10
b=20
* Printf("%d", a++++++b); — longest match, correct one
a++ | + | ++b
No lexical error
a | ++ | ++ | + | b — Syntax error. (Can not generate parse tree)
(no operand before +)
* int i = 2000; — Token for entire number = 5 tokens

* a(1) ++(2) *(3) ++(4) b(5) — No lexical error, No syntax error (No parsing)

* a ++ + ++ b — No lexical error, syntax error.
(one op, other operand)
No operator.

a++ ++b

```
int main()
{
  int a=10, b=20;
  pf("sum is %d", a+b);
  return 0;
}
```

```
Defproc  myproc (int A, float B) {
  int  D, E;
  D = 0;
  E = A | round(B)
  if( E>5) {
    Print D
  }
}
```

Symbol	Token	Dtype
myproc	id	Proc name
A	id	int
B	id	float
D	id	int
E	id	int

```
int main( ) { a=10 int, a, b, c; b=10, c=10;
a = b + c*100
```

Symbol	Token	Dtype
main	Id	fun name
a	Id	int
b	Id	int
c	Id	int

Specification of Tokens - Regular Expression (RE)

RE are important notation for specifying lexeme Pattern.

→ Formal Notation
→ How to use RE in lexical-analyzer.
→ Conversion of RE to automata to recognize tokens specified.

Terminologies

Alphabet: Finite set of symbols. Ex: letters, digits and punctuation.

Ex: {0,1} — Binary alphabet
ASCII - Alphabet used in slw systems

String: Finite sequence of symbols drawn from alphabet (sentence or word)

|s| - length of string - no of characters in a string.

Empty string - ε

|compiler| - 8

Language: Finite set of strings.

Concatenation: If x and y are strings then xy denotes concatenation.

$s\varepsilon = \varepsilon s \longrightarrow s$ [Empty string is identity under concatenation]

$\therefore s^0 = \varepsilon$
$s^1 = s$
$s^2 = ss$
$s^3 = sss$

} Exponentiation of strings.

Operations on languages

Union - $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ in } M\}$

concatenation - $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ in } M\}$

Closure - Kleene closure of L $L^* = \bigcup_{i=0}^{\infty} L^i$ $(U\Sigma)$

Positive closure of L $L^+ = \bigcup_{i=1}^{\infty} L^i$

(Σ*) Kleene closure denoted by L^* of a language L - set of strings by concatenating L zero or more times
L^0 - concatenation of L - ε

(Σ+) positive closure - denoted by L^+ - same as Kleene closure but without term L^0.

Ex L = {A..Z, a..z} D = {0, 1..9}

1) LUD = set of letters and digits - 62 strings of length one.
2) LD = set of strings of length two - 520 strings.
3) L^* - set of all strings of letters, including ε
4) L^4 - set of all 4-letter strings.
5) L(LUD)* - set of all strings of letters and digits begining with a letter.
6) D^+ - set of all strings of one or more digits.

Regular Expressions

RE for valid identifier - l(l+d)* (Note: '_' can be included in letter), where l - letter and i.e d - digit.
or letter (letter | digit)*

RE are built recursively out of smaller regular expressions using below rules.

* Each regular expression r denotes a language L(r).

Rules: To define RE over some alphabet Σ.

Basis
1) ε is a RE and L(ε) is {ε}
2) If a is a symbol of alphabet Σ then a is RE and L(a) = {a} - language with one string of length one.

Basis Induction | Priority

If r and s are regular Expressions denoting L(r) and L(s) then

1. (r)|(s) is a regular Expression denoting Language L(r) ∪ L(s)
2. (r)(s) is a RE denoting language L(r)L(s).
3. (r)* is a RE denoting (L(r))*.
4. r is a RE denoting L(r). - we can use paranthesis around RE without changing the language meaning.

Priority - Precedence of operator. (* > · > |)

1) The unary operator * has highest precedence and is left assosiative
2) Concatenation has second highest precedence & is left assosiative.
3) | has lowest precedence and is left assosiative

Under above assumption
(a)|((b)*(c)) can be replaced by a|b*c

Let Σ = {a,b} What are the languages denoted by following RES.

1) a|b 2) (a|b)(a|b) 3) a* 4) (a|b)* 5) a|a*b

1) {a, b}
2) {aa, ab, ba, bb}
3) {ε, a, aa, aaa ...}
4) {ε, a, b, aa, bb, ab, ba ...} or (a*b*)*
5) {a, b, ab, aab, aaab .....}

A language defined by a regular Expression is called regular set. If two regular Expression r and s denote the same regular set, then they are equivalent i.e $r = s$

Ex $(a|b) = (b|a)$

Algebraic laws for regular Expression for which RE are equivalent

$r\|s = s\|r$	| is Commutative
$r\|(s\|t) = (r\|s)\|t$	| is assosiative
$r(st) = (rs)t$	Concatenation is assosiative
$r(s\|t) = (rs)\|rt$	distributive over |
$(s\|t)r = sr\|tr$	distributive over |
$\varepsilon r = r\varepsilon = r$	ε is identity for Concatenation
$r^* = (r\|\varepsilon)^*$	ε is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular Expression Exercises.

1) Write regular Expression for valid C identifiers.

letter → A|B|...Z|a|b...|Z|_

digit → 0|1|...9

id → letter(letter|digit)*

2) RE For unsigned numbers such as 5280, 0.01234 6.336E4 or 1.89E-4

digit → 0|1|...9 or

digits → digit.digit*

optionalFraction → .digits|ε (. has to be followed by digit atleast one)

optionalExponent → (E(+|-|ε)digits)|ε

number → digits optionalFraction optional Exponent

Note: \ " . ^ $ [ ] * + ? { } | / - special Meaning
these symbols need to be turned off, if they need be represented in character string - use " " or use \ (back slash)

Extensions of Regular Expression.

1) one or more instances - unary, postfix operator (+).
Positive closure. of a RE and its language
i.e if r is a RE then (r)+ denotes (L(r))+.
+ has same precedence & assosiativity as operator *.
$r^* = r+|\varepsilon$
$r+ = rr^* = r^*r$
} Relate b/tcen closure

digit (digit)* — digit+

2) Zero or one instance - The unary, postfix operator
? - has same precedence and assosiativity as *, +.
i.e $r? = r|\varepsilon$
$L(r?) = L(r) \cup \{\varepsilon\}$ — Exponent.
(E [+|-]? digit+)?

3) character class - [ ]
A Regular Expression $a_1a_2 \ldots a_n$ where a_i's are Symbol of the alphabet, can be replaced by shorthand
[$a_1a_2 \ldots a_n$]
i.e a|b|...|z - [a-z]
a|b|c - a|b|c

Ex: digit → [0-9]
digits → digit+ 1.26E6543
number → digits(.digits)? (E [+-]? digits)?

Describe language denoted by following RE.
1. a(a|b)*a - String of a's & b's that start & end with a.
2. ((ε|a)b*)* - String of a's & b's
3. (a|b)*a(a|b)(a|b) - String of a's and b's that the character third from the last is a.
4. a*ba*ba*ba* - String of a's & b's that contain three b.
5. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*
Strings of a's and b's that has even number of a and b.
6. Regular Expression that matches "\ — \"\\

Recognition of tokens.

Branching statement

if (expression) then stmt

or if (expression) then stmt else stmt

or ε

grammar

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| number

— tokens wrt lexical analyzer.

Pascal code – conditional statement
- then appears explicitly after condition.
- relop – comparision operators – =, <>
- terminals – if, then, else, relop, id, number
- Write pattern or RE.

Chomsky hierarchy

Hierarchy for classification of languages.

Chomsky classified languages and grammar into 4 types.

Type-3
Type-2
Type-1
Type-0

Type-3: If the grammar has all the productions of the form

$A \rightarrow \alpha B \mid B$

Where $A, B \in V$ (Non-terminal)
$\alpha, B \in T^*$ (Terminal)

If variable occuring at rightmost side then

grammar said to be Right Linear Grammar. (RLG)

i.e $A \to \alpha B \mid \beta$

└ variable rightmost side of grammar.

$A \to B\alpha \mid \beta$

└ variable on left side of grammar

so grammar called left linear grammar (LLG)

$A, B \in V$

$\beta, \alpha \in T^*$. (α, β) can be $-\varepsilon$ also

Eg:

$A \to aB \mid a$
$B \to aB \mid bB \mid a \mid b$] RLG

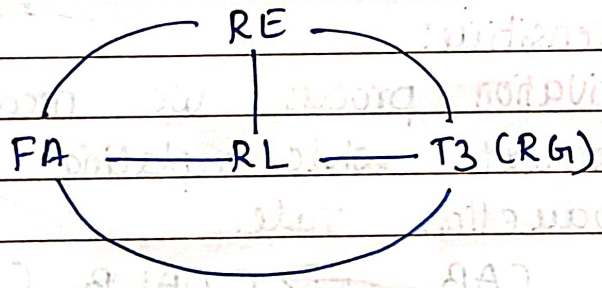
$A \to Ba \mid a$
$B \to Ba \mid Bb \mid a \mid b$] LLG

$A \to Ba \mid a \longrightarrow$ LLG Not Type 3.

$B \to aB \mid a \longrightarrow$ RLG

Note: Combination of LLG & RLG not allowed.

Language generated by Type 3 grammar are called regular language and Regular language can be represented using Finite Automata.



Equivalent in power.

Type 2: If all the productions in the grammar are of the form

$A \to \alpha$, $A \in V$

$\alpha \in (V \cup T)^*$, can be ε

Type 2 are called Context Free Grammar (CFG)

Regular Grammar are subpart of Context Free Grammar (Type 2) which can be either RLG or LLG

If a grammar is regular then its a Context Free grammar.

If a grammar is Context Free then it ~~might~~ may not be regular.

Eg: $A \to aAblab$

Deriving $A \to aAb$

$\to aabb$

Context Free vs context Sensitive

Context Free: Allow a variable (non terminal) to be replaced by a corresponding production rule whenever it appears in a derivation process. The replacement occurs irrespective of context (what lies before or after the non terminal)

Eg: $A \to OAI$ – whenever A found replace it with OAI.

Context sensitive:

In derivation process we need to think about the context while replacing a variable by its production rule.

Eg: $CAB \to COAIB$ (we can replace A by OAI whenever it is preceded by C and followed by B.

Language generated by CFG are called context Free Language

Machine which can be used as an acceptor for CFL is Push Down Automata. (PDA)

Type 3 : RG → FA → RL

Type 2 : CFG → PDA → CFL

Type 1 : Context sensitive Grammar – Restricted Grammar

$\alpha \rightarrow \beta$

α is $(V+T)^* V (V+T)^*$ atleast one variable

β is $(V+T)^*$, ε on LHS.

Restriction: Number of symbols in α should be less or equal to β.

$S \rightarrow AB$	$AB \rightarrow ABBC$
$AB \rightarrow abc$	$A \rightarrow bcA$
$B \rightarrow b$	$B \rightarrow b$

$S \rightarrow \varepsilon$ ('S' should not appear in RHS of the rule)

Context sensitive grammar are used for context sensitive language and machine used is Linear bounded automata.

Type 0: Recursively Enumerable Grammar. (Unrestricted)

$\alpha \rightarrow \beta$

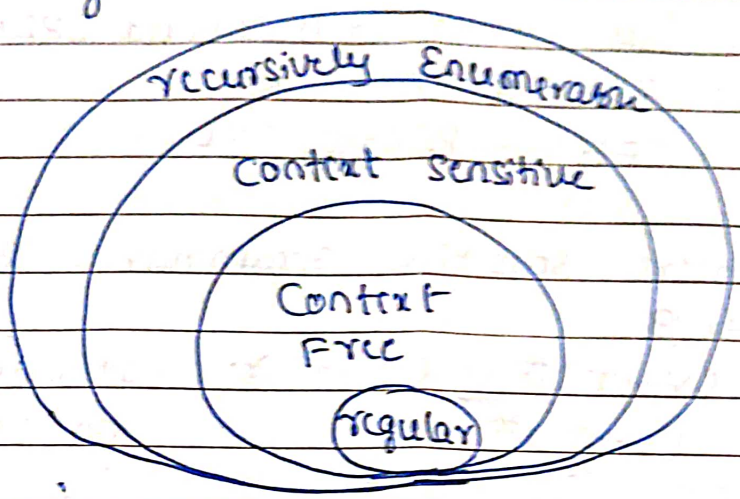
α is $(V+T)^* V (V+T)^*$ — one atleast variable onto LHS

β — $(V+T)^*$ — ε

α — can not be null.

$Sab \rightarrow ba$	$CB \rightarrow DB$
$A \rightarrow S$	$aD \rightarrow Db$
$S \rightarrow AcaB$	
$Bc \rightarrow acB$	

Recursively Enumerable Grammar for Recursively Enumerable language & machine used is Turing machine.



M/C	Grammar
Turing machine	Recursively Enumerable
Linear Bounded	Context Sensitive
PushDown Automata	Context Free
Finite state	Regular

Questions on RE. – Character Class.

1. The first ten letters (upto j) in either upper or lower
   [A-Ja-j]
2. LowerCase Consonants – [bcdfghjklmnpqrstvwxyz]
3. The digits in a hexadecimal number (either uppir or lower case for digits above 9
   [0-9a-f]
4. The character that can appear at the end of English sentence (exclamation, point)
   [.?!]