Principle of Compiler Construction

compiler → software system that translates one source language into another language

error identification easier in ~~compiler~~ interpreter

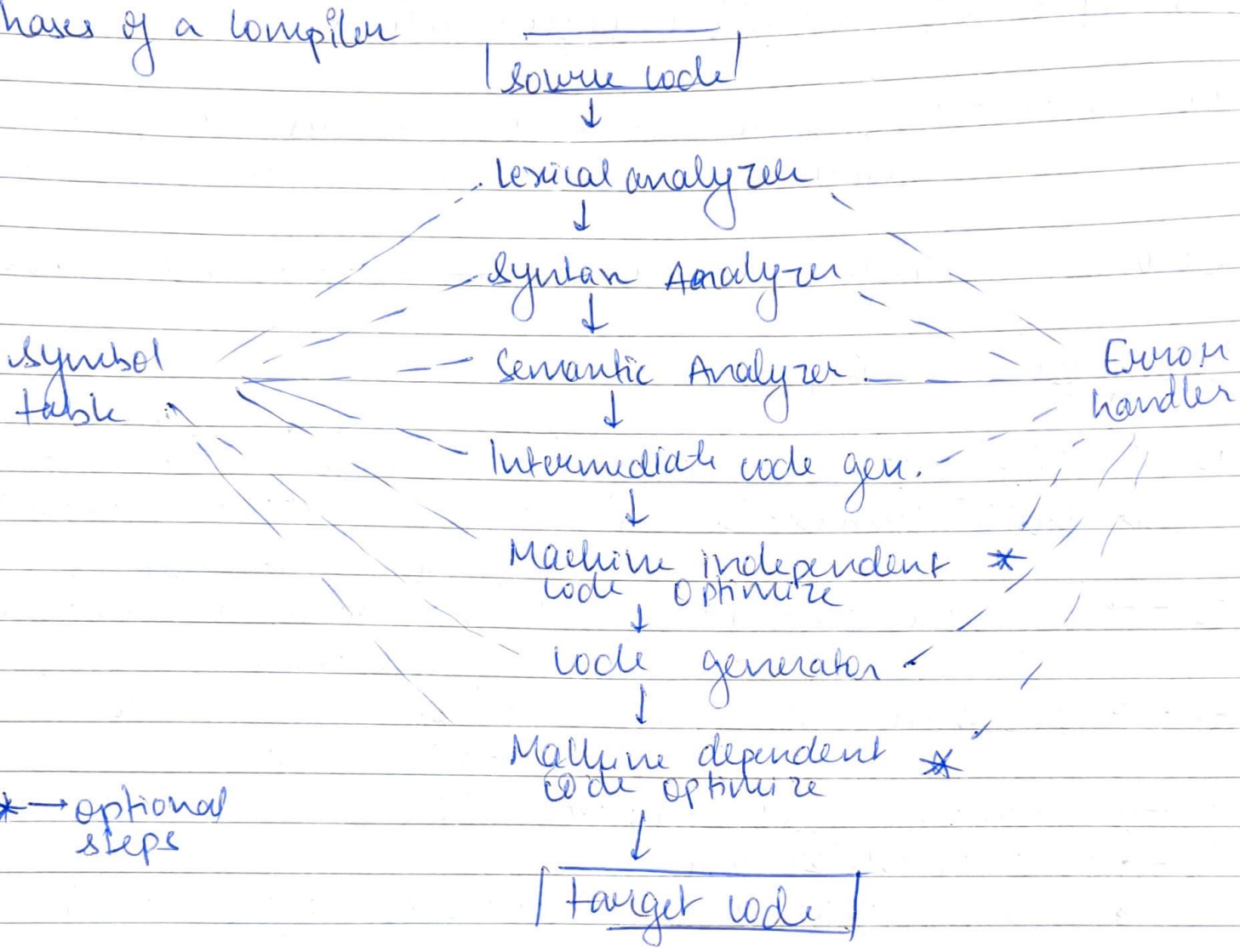
Compiler → 6 phases

1. ↳ lexical analysis phase (valid identifier, valid data type, valid exp)
   ↳ finite automata (reg ex pattern match)
2. → Syntax analyzing (whether statement is unambiguous / follows a production rule)
3. → Semantic analyzer (checks for semantic consistency)
4. → Intermediate code generator (role of computer organization)
5. → Code optimization
6. → Code generator (generates target code)

* Preprocessor, compiler, assembler, linkers

skeletal source prog
↓
preprocessor
↓
compiler
↓
assembler
↓
linker
↓
absolute machine code

phases of a compiler



eg. position = initial + rate * 60

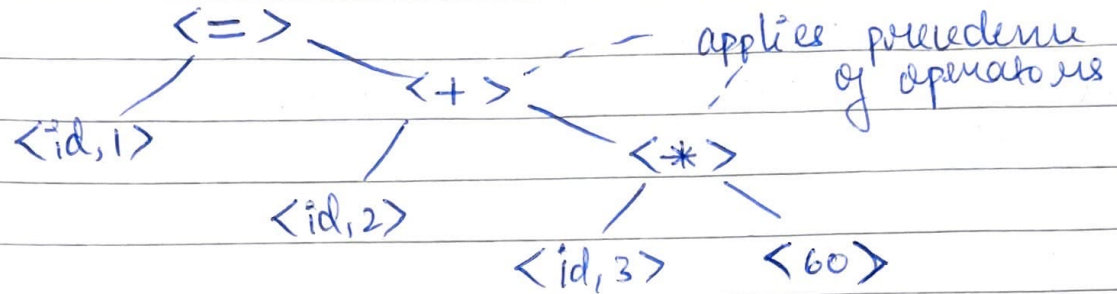
lexeme → token
<token-name, attribute/>
index value

=> <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

syntax analyzer → converts tokens to * syntax tree

condensed form of parse tree

→



- grammar used by syntax analyzer should be unambiguous in nature
- semantic analyzer ~~also~~ checks for semantic consistency. (

* intermediate code generator
  → max 2 operator and 1 output
  t_1 = int to float (60)
  t_2 = id_3 * t_1
  t_3 = id_2 + t_2
  id_1 = t_3

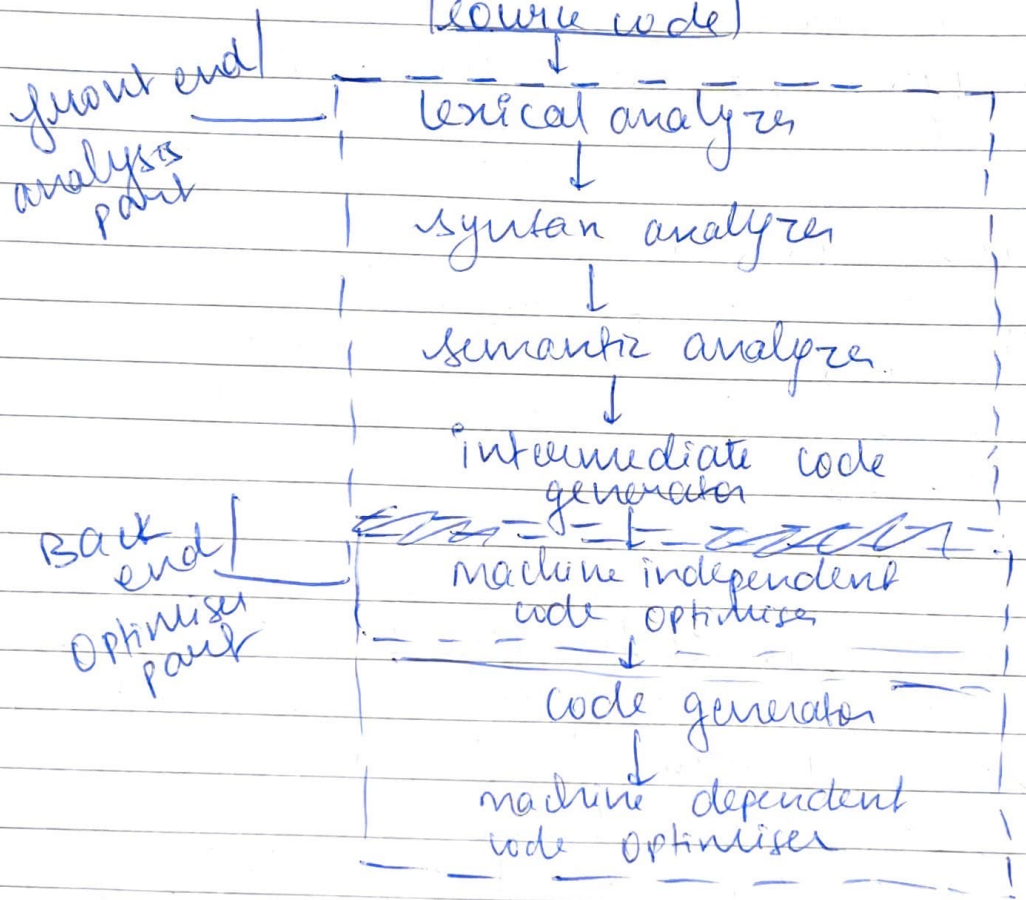
* machine ~~id~~ independent code optimizer optimizes intermediate code and is responsible for tasks like removing redundant variables
  eg $t_1 = id_3 * 60.0$
  $id_1 = id_2 + t_1$
  ] 2 line optimized code

* code generator
  → generates machine code
  eg LD R1, id3
  MUL R1, R1, #60.0
  LD R2, ~~R2~~ id_2
  SUM R_1, R_1, R_2
  ST ~~ID~~ ID$1, R1

* Machine dependent code optimizer
→ Optimizes the machine code generated by code generator

Passes

single pass and multi-pass compiler



multi-pass / 2 phase compilers generate machine code for a specific machine

No. of phases

eg

position = initial + rate * 60

↓

| Lexical analysis. |

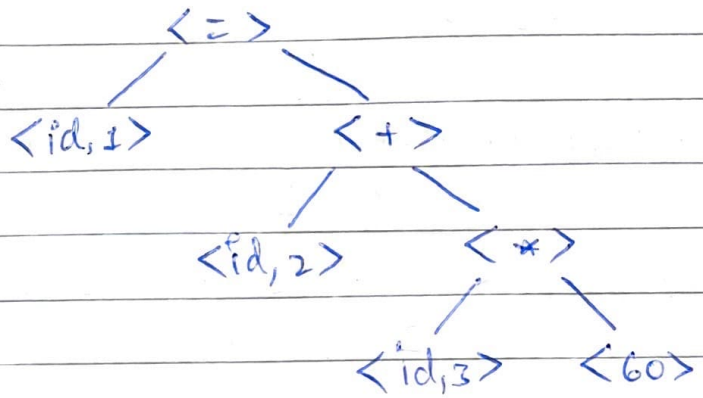
↓ token stream

<id,1> <=> <id,2> <+> <id,3> <*> <60>

↓

| Syntax Analysis |

↓ syntax tree



* Ambiguous and Unambiguous grammar

eg $E \rightarrow E+E \mid E*E \mid num \mid E=E$

to remove ambiguity, assign precedence to diff operators

lowest precedence

⇒ $E \rightarrow E=T \mid T$

$T \rightarrow T+F \mid F$

$F \rightarrow F*G \mid G$

$G \rightarrow num \mid <id>$

* To remove ambiguity:

① Precedence

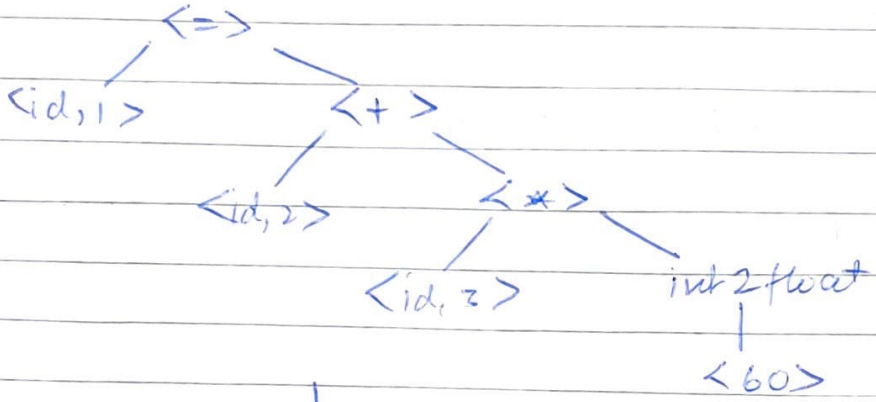
② Left to right (Associativity)

Syntax Tree

↓

Semantic Analysis

↓ syntax tree (semantically correct)



↓

Intermediate code generation (machine independent component)

↓

$t_1 = int2float\ (60)$
$t_2 = t_1 * id3$
$t_3 = t_2 + id2$
$id_1 = t_3$

→ right side has max 2 operands and 1 operator

↓

Code optimization → using DAG

↓

$t_1 = id_3 * 60.0$
$id_1 = id_2 + t_1$

↓

Code generator (machine dependent component)

↓

```
LD   R1, id3
MUL  R1, R1, #60.0
LD   R2, id2
ADD  R1, R1, R2
ST   id1, R1
```

Compiler Construction Tools

① Scanner Generator
   ↳ lexical analyser
② Parser generator
   ↳ syntax and semantic analyser
③ Syntax directed translation tool
   ↳ provides multiple routines to go through syntax trees and generate intermediate code
④ Data flow analysis generator (gives flow of program)
⑤ Code ~~optimizer~~ generator
⑥ Compiler construction toolkit

Evolution of Languages

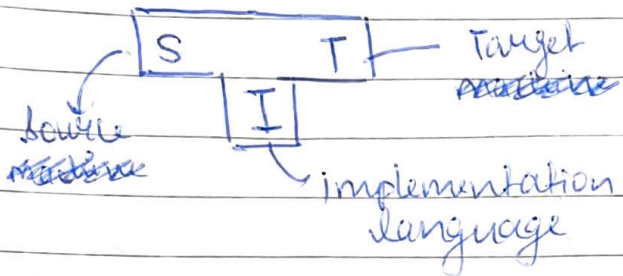
1940 → machine language → 1st generation
1950 → Assembly → 2nd generation
1955 → HLL (C, Pascal, etc) → 3rd gen
→ Problem specifying → 4th gen
(SQL, postscript)
→ Logic and Constraint based → 5th gen
(PROLOG, OPS 5)

based of nature / working culture

Imperative	Declarative
→ flow is defined for how the work is to be done	→ flow is not defined. Only work is defined
eg C, Pascal	eg Relational algebra, PROLOG

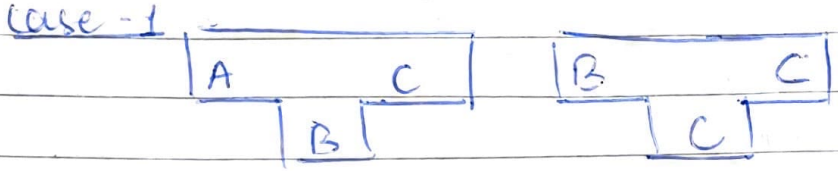
* Cross compiler

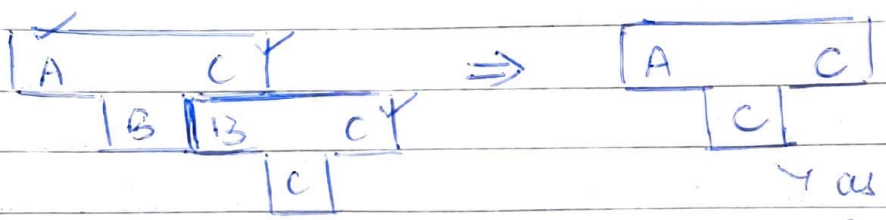
— a compiler which runs on one machine and generates target for another machine



→ T diagram

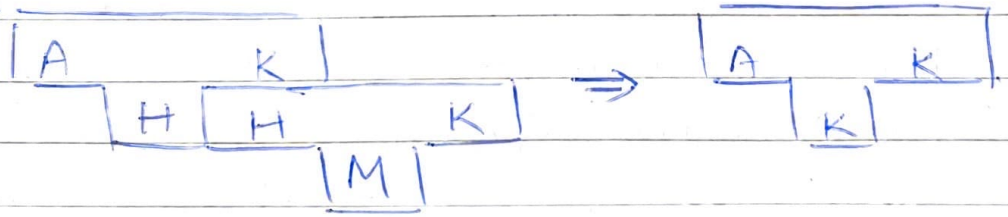
case - 1





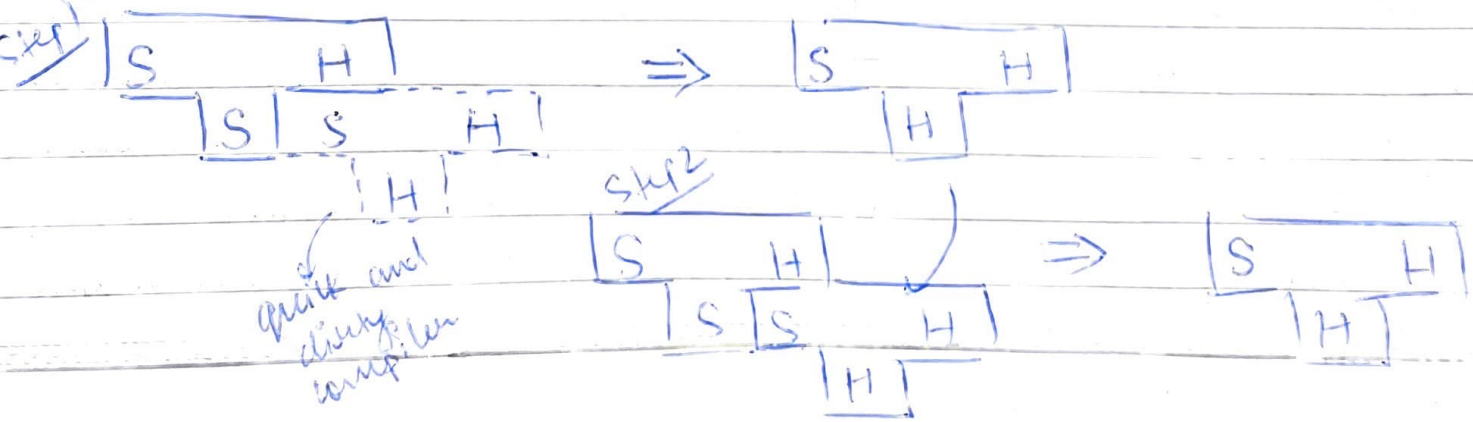
Y as B ultimately compiles to C, C is the implementation language

case - 2

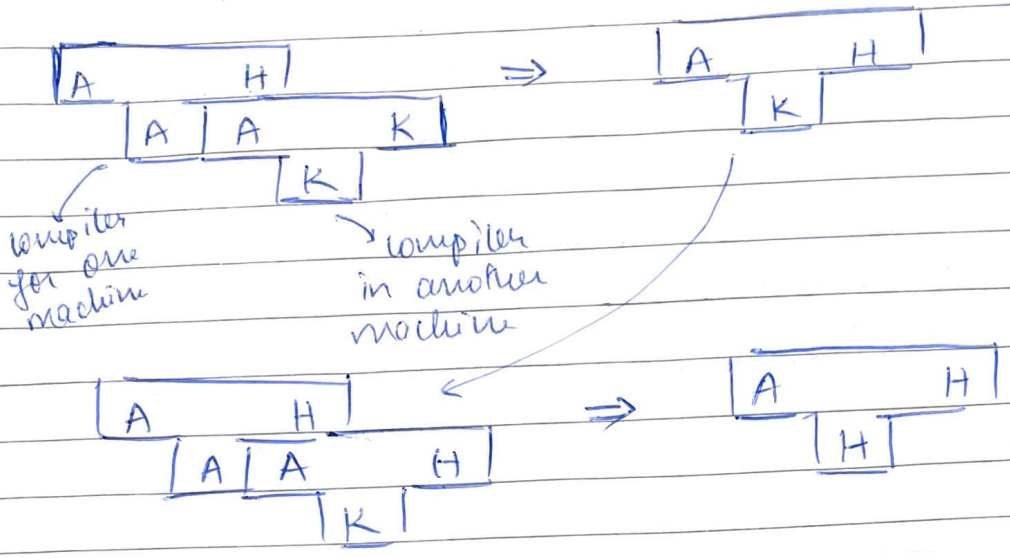


* Bootstrapping (self compiling compiler)

write a compiler in a simple language, which further generates compiler for a more complex language

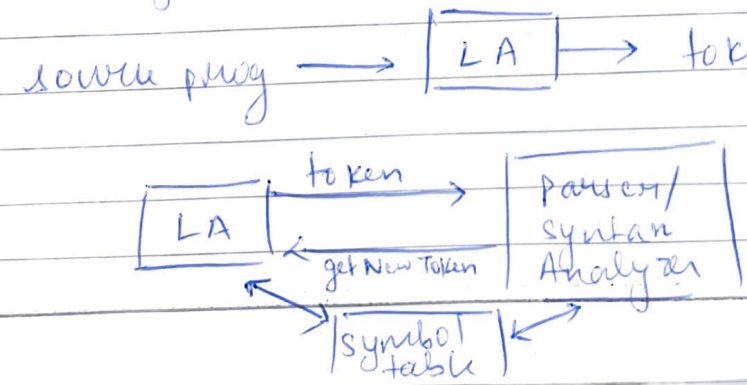


* Portability
Any generated program should be executable on multiple machines



Just In Time Compilation
(compilation process occurs at the time of execution.
eg. Java (JVM)
C# (Common language Runtime)

Lexical Analysis (Scanner Generators)
takes source programs, identifies Lexems in it and generates token stream



① Token generation

eg int a = b + c ;

T1 T2 T3 T4 T5 T6 T7

② Error Handling

↳ Tries to implicitly handle errors, if not then send information to error handler

③ Remove / Handles whitespaces

(blank, spaces, tabs, newline, comments etc)

④ Macro Expansion

* Tokens

identifiers

eg <id, pointer>

keywords

tokens are same as keywords

eg <'int'>

numbers

eg <number, pointer>

TBD

Operators

eg <op>

punctuations

eg <;>

Brackets

eg <(>, <{>

Token

↳ is an interpretation of valid lexeme

lexeme: atomic element which cannot be further divided and is useful to the program

to identify valid lexeme, patterns are used

patterns: are basically regular expressions

Input Buffering

E = m * c * * 2

(* * : to resolve this token, LA needs to look ahead)

⇒ operators as they come are put into an input buffer up until a valid lexeme is obtained / Error generated / invalid lexeme encountered.

* One buffer technique:
  - if buffer ~~exeed~~ is filled i.e lexeme overflows current buffer

* Two Buffer technique:
  - A second buffer is initialized

* to check if buffer is filled, sentinels are used

sentinels: a pointer at end of lexeme in a buffer is a sentinel

~~aeaeiou~~

* Regular expressions

5. All strings of lowercase letters that contains 5 vowels in order

→ C, no_vowel_set = (b+c+d+f+g+h+j+k+l+m+n+p+q+r+s+t+v+w+x+y+z)

⇒ no_vowel_set* a no_vowel_set* e no_vowel_set* i no_vowel_set* o no_vowel_set* u no_vowel

⇒ (non_vowel)* aa* (non_vowel)* ee* (non_vowel)* ii* (non_vowel)* oo* (non_vowel)* uu* (non_vowel)*

~~aba~~ ~~abaaba~~

(non_vowel + a)* a

(Good) ✱ (C+a)*a (C+e)*e (C+i)*i (C+o)*o (C+u)*u C*

6. All strings of as and bs, that donot contain the substring abb

(ϵ + b*)(aa*baa*)*

b*(a+ba+bb)*

b*(a+ba+(ϵ+b))*

b*(a+ab)*

b*(a+(ϵ+b))a*

7. no subsequence abab

b*(ϵ a* + ab) a*

b*a*(ab + ϵ) a*

b*(a*b)a*

b aaa b

1. RE → minimized DFA
2. NFA → DFA
3. RE → NFA

/* /// */

/a /***

Q Regex for comments consisting of string surrounded by /* */ without an intervening */ until it is inside " "

C → non /, *, /, "

/* C* (cc (C + * + /)* 99 + ϵ) C* */

/* (C*(*+ /+ϵ) C* (*+ /+ϵ))

/* (C (*+ /)* + ϵ)*

(/)* (*)*

////*

/* C*(*+ /)* C*

* Lex form of Regular Expression

Expression	matches	Examples
c	any non operator character	a, b, c, ...
\c	escape characters	*
"s"		"**"
.	any character except newline	a.*b
^	beginning of a line	^abc.
$	ending of a line	abc $
[s]	any one character from list	[abc]
[^s]	no character from set	[^abc]
r*	any 0 or more occurence	a*
r+	any 1 or more occurence	a+
r?	Zero or one	a?

$r\{m,n\}$ m to n occurrences a {1,2}

r_1, r_2 r_1 followed by r_2

$r_1 | r_2$ r_1 or r_2

(r_1) (a)

r_1 / r_2 r_1 when followed by r_2

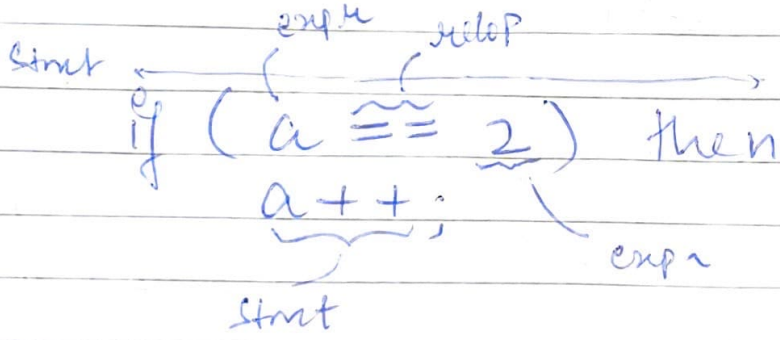
wordspace (blank / newline / tab)+

* Recognition of Tokens

stmt → if expr then stmt /
if expr then stmt else stmt / ϵ

expr → term relational operator term / term

term → id / num



regex for if statement.

~~ϵ~~ if \([a-z_]+.*