Compiler Design

Introduction to Compilers & Lexical Analysis Phase

Importance

Computer understands binary or machine or low level language. But language like C or C++ we write like english so how does computer understand?

We are interacting with system or phone. How small unit understanding our actions.

How we are able to interact with computer?

↓

With help of OS → interface between hardware and software

OS - Software → set of programs to do a task

↳ collection of programs that can perform a task.

Utility Software / System software

↳ We dont have much involvement, we just (user) initiate it & rest computer does it.

Ex: OS, compiler, interpreter, assembler, linker, loader

Application Software - user has to give input.

Set of instructions to perform a given task or particular task → program

↳ Task is smaller

Software → Task is bigger.

Compiler - converts high level language into low level language

Because computer understands low level or binary & we understand high level language.

Given one language → [Translator] → It produce another language

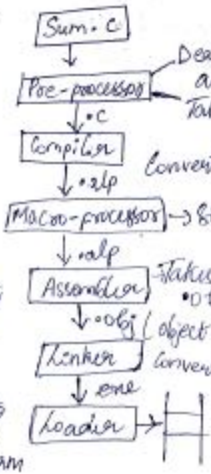
Such a translator is our compiler.

- Compiler is translator which translates high level language program into machine level language program or assembly level language program.

HLP → [Compiler] → MLP/ALP

```
#include <stdio.h>
int main()
{
    int a=10, b=20;
    printf("Sum of %d & %d is %d", a, b, a+b);
    return 0;
}
```

Wrote program as sum.c which is high level language program



It object file into executable file

Loads exe file into memory & initiate execution of program

Command prompt

$ cc sum.c → It produce .out file
by default it is a.out

$ ls
$ a.out → This means file is correctly processed
$ ./a.out ⇒ then sum of 10 & 20 is 30 is
output shown.

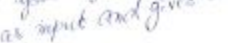
→ Here all preprocessor, compiler, macroprocessor, assembler, linker, loader processed & produce .out file.

Compiler → System Software

What is cc?
What is Compiler?
Individual unit of program - tokens
Divide entire program into token
Using Regular expression divide program into tokens.
Syntatically correct or not → Grammar
Combine tokens by using production & build parse tree
Machine to build parse tree is pda.
From parse tree convert into intermediate code (once parse tree generated then syntatically correct)
converted into intermediate code & optimized to generate Assembly level language program.

tokens
↓
Regular expression → Unit I
↓
Parse tree → Unit II & III
↓
Intermediate code - IV
↓
optimize
↓
Generate ALP } V

Another system software is Interpreter which takes HLP as input and gives directly output



Interpreter takes HLP directly as input and executes line by line. Interpreter HLP into low level language and executes and gives output.

Compiler compiles program at once.

↳ irrespective of errors it goes till end of program compiles entire program. If there are certain errors it list out all errors.

Compiler

① compiles whole program and identifies as many as errors possible.

② Compiler takes huge space as it needs to all variables used in program.

③ Once program is compiled it doesn't require original file to execute.

Ex: C, Java Compiler

Interpreter

① It stops at first error until end user rectifies doesn't move

② Interpreter takes (occupies) less space

③ It requires original file to execute

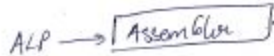
Ex: JVM, browsers (includes interpreter (Chrome, Internet Explorer etc)

Macro processor (System Software)

↳ It takes ALP as input & gives ALP as output

The input contains some macros these macros are expanded in output

Assembler (System Software)



Diff object programs given to linker & it links them and gives executable file as output (.exe or .out file)

↓

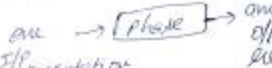
a.out file

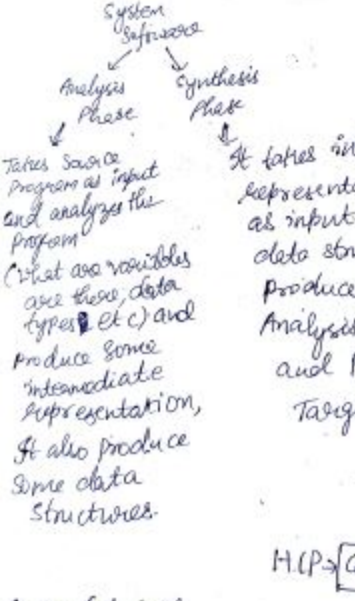
↓

Takes executable file and keeps it in memory & starts executing

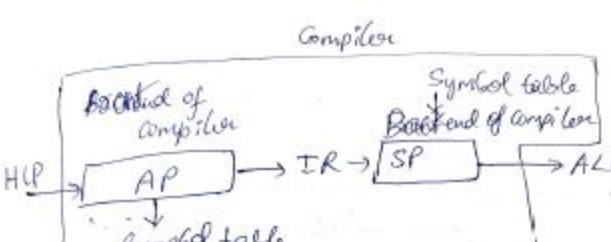
Two phases in System Software while designing

- Phase : A phase is a logically cohesive operation that takes one input representation & produces another output representation.

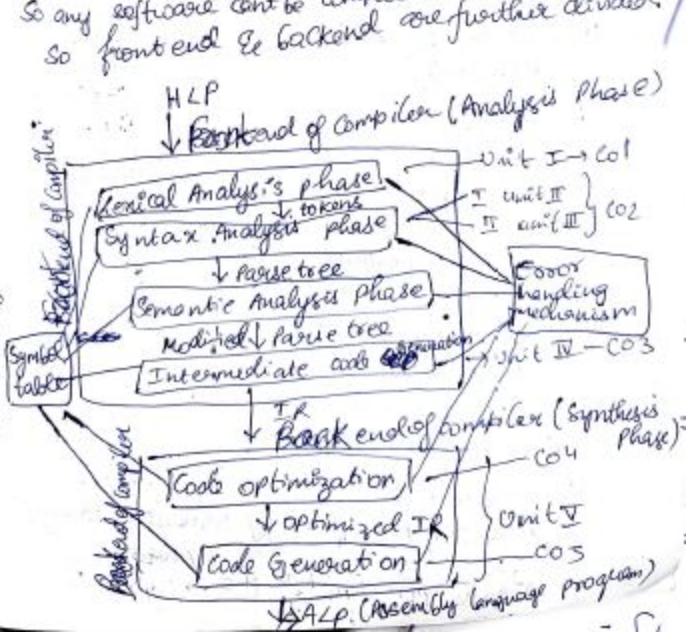




- DS is Symbol table



So any software can be completed in single flow
So front end & backend are further divided.



Lexical

It scans entire source program & divides entire program into individual unit known as token. output is token.

So token taken by Syntax Analysis phase & combine tokens by means of some rules & produce output called Parse tree.

Syntax → look for syntax of rule doesn't see meaning, checks only form correct or not.

```
void swap(char a, char b);
int a=10, b=20;
swap(10, 20);
```

```
int a=10
float b=5.5;
float c=a+b;
```

Syntax wise Correct but semantic (meaning wise error) bcz adding float & integer we don't do.

But in C we can do bcz compiler we do coercion (internal typecasting) → done by Semantic Analysis Phase

Depends upon datatype & e

* compiler sees whether function call & function definition same name and same no. of characters so ✓ correct Syntax but ✗ meaning wrong Semantic wrong bcz we passed integer during call but char during definition.

So output of Semantic Analysis phase is modified parse tree.

Intermediate Code Generation → converts high level language statements into some Intermediate representation close to target program.

$c = a + b * c - d/e;$

If my ALP should contain atmost one operation

So at c has 4 operation we want it to be close to ALP

So $T_1 = b * c$
$T_2 = a + T_1$
$T_3 = d/e$
$T_4 = T_2 - T_3$
$c = T_4$

} Apply precedence & associativity to be in form of IR so that can be easily converted into ALP or target

Some redundancy exist in IR which are not useful, if eliminate that redundancy convert into efficient code (either less space or less time but both accomplishing not possible) which is Optimized IR as output.

Code Generation takes optimized IR & converts into ALP using registers.

ALP is directly linked with machine as it processor, so ALP can't do multiplication

For compiler to remember everything uses Symbol table or some data structure. So Symbol table can interact with all phases.

Compiler when sees error does some repair before moving next line → Error handling mechanism
|| Interacted by all phases

Pass → Collection of phases which takes one representation as input & produces another representation as output.

Depends upon designer ability we can design all phases in either one pass, two pass etc

Single pass compiler → combine all 6 phases into single pass

Two pass compiler → Combine all 4 phases into one pass & 2 phases into one pass

Multi-pass compiler → combine lexical analysis and Syntax Analysis phase into one pass

Regular expression

int a, b, c.;
↑ ↑ ↑ ↑
token token token token

(RE) helpful to divide entire line into individual tokens

, → token
; → token
int → token

⇒ Combined using rules Grammar

Semantic, Intermediate Code Generation → one pass
Code optimization, Code Generation → one pass

If different phases then lexical Analysis phase produces file containing tokens as output ip given to Syntax Analysis phase which produces parse tree.

So when diff passes, intermediate files are produced due to which compiler gets heavy.

First compiler → Find out
↳ How much time it took to design it?
find out various types of compilers available?
Speciality of C compiler?

C → procedural language
C++ → C + Object oriented language
Java → Completely object oriented programming language

In order to design phases → automatic tools are present otherwise huge amount of code to be written

These automatic tools are called scanners bcz used to scan & divide into individual tokens.

One such scanner is lex tool.

Produces lexical Analysis phase

YACC Tool → Produce another syntax Analysis phase

Yet Another Compiler Compiler (But for what lang we need to specify)

Phases of Compiler:

HLP → Lexical Analysis Phase → Tokens

Int main()
{
int a=10, b=20;
printf("%d

Scan program from beginning & form into some called lexems

Initially scans 1 lexeme as i
Next n lexeme as in
Next t lexeme as int

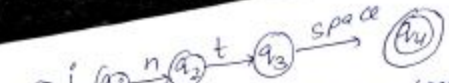
First token ↙ ↳ lexem recognized first & then what type of lexem

Space not considered as lexem

main → Another token
(→ one token
) → one token
{ → one token
int → one token
a → one token
= → one token
10 → one token

Entire program divided into series of tokens.

Mechanism used to divide into tokens is automata (finite automata)



int recognized by finite automata

↓

String recognized by FA → Regular expression

• Recognize token use RE (Regular expression) using finite automata & RE can recognize regions

types of tokens:

- int keyword
- A, main, printf } identifier (variable name, function name)
- =, ; → Symbols
- %, (,), + → operator
- 10, 20 → Constants

} Diff type of tokens (Types of tokens)

Tokens can be represented as pair

(Token type, Token value)

⇕

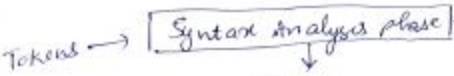
Depends upon type of token

Keyword can take 1 to 32

Identifier → Location where identifier placed in symbol table

value ↙

② Syntax Analysis phase (only looks at form not meaning)



d = a + b * c ; → Statement correct

form of statement or instruction correct or not
↳ Syntax Analysis phases

form correct or not depends upon some rules

while (cond) ; → Run infinite times
↳ no error as Syntatically correct but meaning wrong

```
while (cond) {
    Statement 1
    :
    Statement n;
}
```

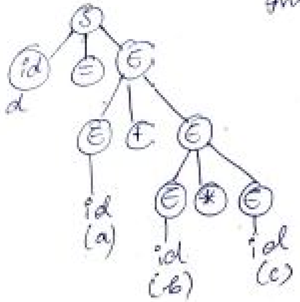
CFG used to define rules to validate each & every instruction of a program.

Based on productions given in CFG

If statement d = a + b * c ; written appropriately then we can produce parse tree

$S \rightarrow id = E$
$E \rightarrow E*E \mid E+E$
$E \rightarrow id$

leaf nodes → terminals
Internal → non terminal



(BODMAS rule followed)

d = a++bc As statement not correct so cant produce parse tree

Semantic Analysis phase (It does look meaning of statement)

Parse tree → [Semantic Analysis Phase] → Modified Parse tree

① c = a + 10; It should be of same data type but have diff data type
c, a → float

Compiler internally makes 10 as 10.0 and doesnt throw error

Modification done by compiler & produces modified parse tree

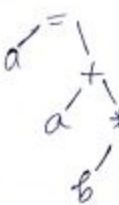
① swap(10, 20);
void swap(char a, char b);
} Compiler internally converts 10 & 20 into characters (ASCII values)

Parse Tree
↓
Intermediate Code Generation
↓
Intermediate representation

It should be simple statement and close to Assembly language program

Gap btwn high level & assembly language
⇓
very high
(Conversion → diff directly)

- Postfix Notation
All statements can be converted in some form of postfix notation
- Syntax Tree



(Instead of id placed values)

- From parse tree obtain syntax tree

Infix exp
operator must be placed btwn operands
But should know BODMAS rule for precedence to evaluate

(not worried about precedence)
↓
Already taken care

Prefix exp
+ab

Postfix exp
ab+

- Three Address Statements

$d = a + b * c$

At most one operator

$T_1 = b * c$
$T_2 = a + T_1$
$d = T_2$

Three address C2 mean of 3 operands in each statement

=) From now on' deal with three address statement

IR → [Code optimization] → optimized IR

Remove redundant statements Without Comprimising meaning is optimization (code improvement)

$T_2 = a + T_1$
$d = T_2$

$T_1 = b * c$
$d = a + T_1$

Done by Code optimization as ICG cont do this thing as about here

local optimization
(Focused only on blocks)

loop optimization
(Focus on block in loops)

Global optimization
↓
Depends upon more than one block

IR → [Code Generation] → ALP

ALP → low level language (Tightly bound with hardware)

Know about hardware, registers is very much important

More time R_1, R_2 are two registers available freely

These may take more memory cycles 3

- MOV R1, b
- MUL R1, C
- MOV R2, a

} Direct mode Dealing with memory location

ADD R2, R1 } Both register So register mode

Takes only 1 memory cycle → Less time

How effectively use registers & it should know hardware structure to produce ALP

Error handling mechanism → Routines to deal with errors & fix the errors

Phases of Compiler - An Example

Amount = Principle + rate * team/100;

→ High level language statement

↓

Lexical Analyser (LAr)

↓ tokens $id_1 =, id_2, +, id_3, *, id_4, /, \langle int \rangle, ;$

Syntax Analyser (SAr)

↓

First combines id3, *, id4.

then /

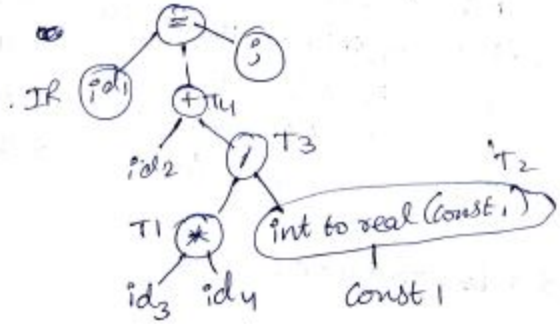
Parse tree

Amount, principle, rate, term are real and 100 is constant So diff data types

So SeAr adds One function to const that is int to real which will take 100 as Parameter & convert to 100.0

Modified Parse tree

(1) Syntax Tree → Eliminates E in such a way that operators become internal nodes & operands become leaf nodes. whose depth is very high that operation performed first.
It is simplied parse tree



(4) Three Address Statements

$T_1 = id_3 * id_4$

$T_2 = inttoreal(const\ 1)$

$T_3 = T_1 / T_2$

$T_4 = id_2 + T_3$

$id_1 = T_4$;

(5)

$T_1 = id_3 * id_4$

$T_3 = T_1 / 100.0$

$id_1 = id_2 + T3$

Empty Registers
R_1, R_2

(6)

MOV	R_1, id_3	$R_1 \leftarrow id_3$	3R
MUL	R_1, id_4	$R_1 = R_1 * id_4$	3R
DVI	$R_1, \#100.0$	$R_1 \leftarrow R_1 / 100.0$	4R
MOV	R_2, id_2	R_2, id_2	3R
Add	R_2, R_1	$R_2 \leftarrow R_2 + R_1$	1R
STORE	R_2, id_1	$id_1 \leftarrow R_2$	3R

Division ← immediately ⇓ Passes directly the operand into instruction

$d = b * b - 4 * a * c$;

LEX TOOL → It is for scanner
It is given RE & it will give us into tokens

YACC tool → Produce a parser program.

Yet Another Compiler Compiler

Different ways or The Science of a building a Compiler.

There are 2 methods:

1. Boots trapping
2. Cross Compiler

→ Building of a compiler on its own action is known as bootstrapping.

Cross Compiler - The Compiler that runs on one machine & produce an output that can run on another machine.

Build a compiler for particular language

What are needed

These are keywords

Define way constants, symbols, etc that programming language accepts

Define Syntax

Define tokens → RE

Define rules → CFG

metal detector → chip

Automatic washing machine → chip → Not a computer

Automatic geared bike → Not a computer

Compiling on one machine but output on another machine

① Bootstrapping

T_{X}^{YZ}

X → The machine that runs (Program runs on that machine) / Source language of compiler

Y → Input lang (lang in which I want to design compiler) / in Program

Z → The machine that runs / Target lang of compiler.

Internal hardware architecture of machine (we should have that knowledge)

Interested to develop a compiler for language L

small set of features ✓ S C L

with S features we can build compiler for L

Using S we can write compiler program for machine A.

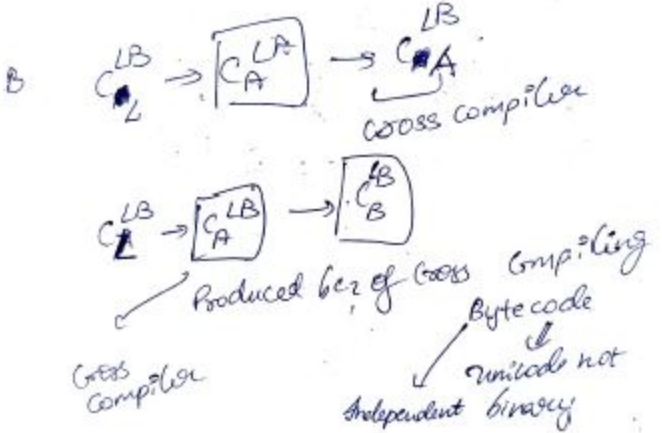
C_{A}^{SA}

Bootstapping

Compiler runs on machine A & accept any programming language & produce target that runs on machine A

Eg: Java C is written in Java itself

B



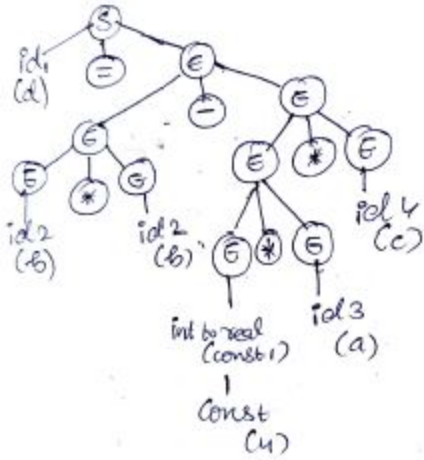
$d = b * b - 4 * a * c$

① ① ② ③

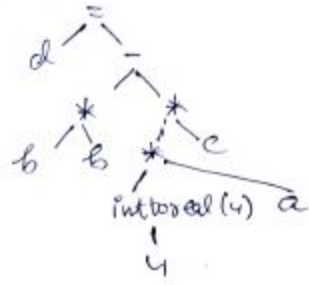
① $id_1, =, id_2, *, id_2, -, const1, *, id_3, *, id_4$

New token
↳ Checks in Symbol table
If same gives same identifier

②



③



$T_1 = b * b$

$T_2 = \text{int to real}(4)$

$T_3 = T_2 * a$

$T_4 = T_3 * c$

$T_5 = T_1 - T_4$

$d = T_5$

(3)

$T_1 = b * b$

$T_3 = 4.0 * a$

$T_4 = T_3 * C$

$d = T_1 - T_4$

(6) R_1, R_2

```
MOV  R1, b
MUL  R1, R1
MUL  R2, #4.0
MUL  R2, a
MUL  R2, C
Sub  R1, R2
STORE R1, d
```

4 * a * b

Left associative

4 ^ 2 ^ 2

Two same precedence associativity

Right associative

Register mode Reg Reg (First operand in Register, second operand in register)

Reg Memory location (Direct location / From memory location fetch & use)

(Allocate memory to stack memory location) Stack location

direct value (Immediate value)

ML ML (Memory direct mode)

Lexical Analysis Phase

High level language → Lexical Analysis Phase → Tokens

How to design lexical Analysis phase?

Consider 3 aspects

Scan entire source program and character by character & divide into tokens → Main work

1) A mechanism to describe tokens

(How token appears at end of output

Each token has a different type of characteristic (pattern known as Regular Expressions.

⇒ Tokens in form of RE

Pattern for operator, symbols etc

2) A mechanism to recognize the token

- Finite Automata

Deterministic as it can be implemented

3) It produce a mechanism to perform certain action

- produce o/p if token recognized

Otherwise error!

If output produce give it to parser

② Role of Lexical Analyser

Two roles:

→ ~~If~~ As a single phase produce all tokens (of src program)

If doesn't scan entire program at a time)

→ It acts as coroutine/subroutine to produce a token & give it back to parser. (Gives one token at time)

(Scans when parser requests & give it to parser, again if parser requests scan remaining program & give it to parser)

③ Need of lexical Analyzer

From HL to produce parse tree → very difficult

If not individual tokens then difficult for parser to convert into parse tree

) To simplify function of parser (simplify task of parser (Syntax Analyzer))

```
c = a + b;
c = a+b;
c=a+b;
```
} Additional spaces skipped by lexical Analyzer

```
/* Sum.c developed by CS */
/* Date created: 9th Dec, 2012 */
/* Updated on 12th Dec, 2023 */
```

Any preprocessing done in the task of Lexical Analyzer

(2) Preprocess the HLP (to give exactly required code)

Input given to lexical analyzer in form of file.

Preprocessing to be done to file is very much difficult to insert, delete between lines so use buffer to do Preprocessing.

Buffer ⇓ synchronize Processor & memory

Fetches lot of instruction & stored in buffer In buffer we can delete add easily

Buffer

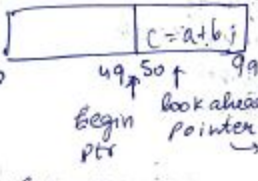
Copy instructions & fetches instruction from memory to buffer so for faster access

fetches into buffer first & then reads it from file

If comment there in buffer removes all Comments. If there are spaces in input buffer they are removed.

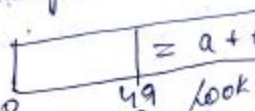
If buffer → 100 characters

Split into two parts



String between look ahead pointer & begin ptr is a token

So look ahead first points at C & then moves forward, space recognized as delimiter & look ahead moves back & C is recognised as token removed, space is squeezed & remaining present



Look ahead like circular queue after moving last on move to first part.

Token → Smallest individual part of program
↳ collection of character that recognizes as single unit
Always represented as string as it is present in file

Token types
- Keywords
- Identifiers
- Constants/literals
- Operators
- Symbols

Each token has diff pattern

Each keyword has unique pattern

All unique pattern
- int → one pattern
- float → another pattern
- break
- else
- if
- while

Pattern for integer → int
inta doesn't match anything
↳ so not keyword

Identifier → Name given to programming constant
↳ Require pattern, to describe identifier
↓
They are rules (Regular expression)

① Character or _ → First character

② Remaining can be character or digits

This pattern uniquely defines all names
Known as RE.

Pattern

('A') Character const → Enclosed in single quotes (' ')

("Gitam") String const → enclosed in double quotes (" ")

Operator → 9 diff operator
↳ diff types, diff pattern

General term for pattern → RE → 3 operations
Concatenation
Kleene closure
union

l	name for RG	Corresponding RE	Note: Square Brackets ⇓ Take only single character
l		[a..z A..Z]	RE
d		[0..9]	

Defined RE based on names

id → l(l/d)* → Kleene closure
→ RE
→ union
→ Concatenation

3 operations one applied

int or float

int → dd* dd* + dd*.d* → RE

Diff types of token → described in terms of RE

+, ++, --, <=, ==, !=	RE for operators
int, float, break, else, if, while	RE for keywords

Lexem → Sequence or group of characters

During lexical analysis state scanning the string formed → token

while returning it classify token.

int a, b

↓

Scanned character by character

Stored whenever delimeter comes & it can be called as token & while returning it classify & return.

lexeme → represented as pair

Pair (token type , token/attribute value)

↳ Parser uses it

Used by lexical Analysis

token value/attribute → used by many phases
↓
first used by parser (Syntax Phase)

For keyword return as it is

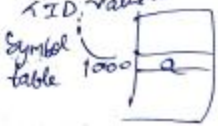
For identifier token value is location where it is placed in Symbol table.

For const also value where it is placed in symbol table that location number is assigned

Operator → each operator, a number is assigned that is taken as value

Symbol also number is assigned. So that is taken as value.

In symbol table only number is assigned to each symbol / so lexical analysis gives token value



(2) Recognise token

FA

↳ FSM that consists of finite set of states & has finite set of transitions.

$M(Q, \Sigma, \delta, q_0, F)$

Special kind of flowchart consisting of states as circle

Describe token in form of transition diagram/ state transition diagram

transition as arrows

input symbols over arrows

$l(l/d)^*$

↓

ISR

③ When reaches > recognizes as identifier but we need to go back as ...

moving backward size of delimeter with help

* The process of taking the look a head pointer moving backwards the size of delimeter in order to recognize exact token is called as Retraction.

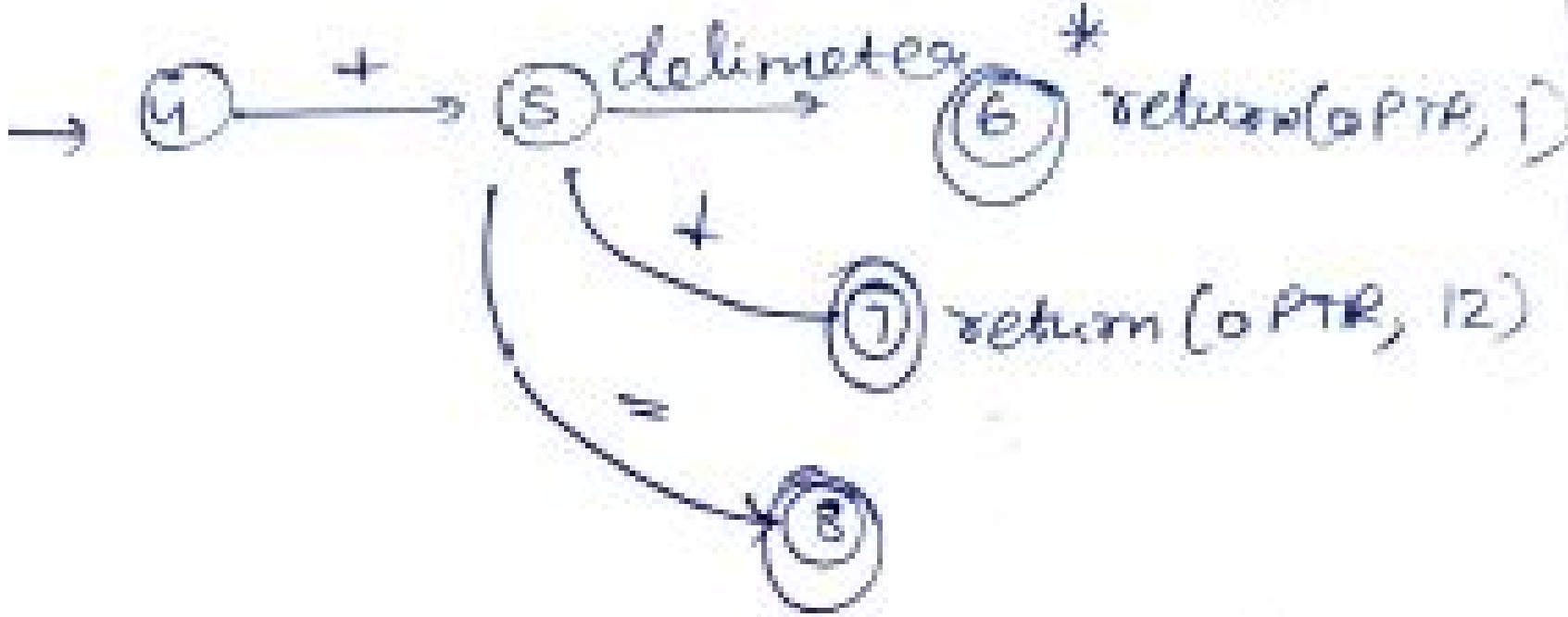
When * is there in final state retraction required how much delimeter is there that much size go back.

∴ We need to return token

return (ID, Install()) identifier there in Symbol table fetches otherwise place & fetch from Symbol table

* Install() Procedure which finds the identifier in symbol table & if found get the address of identifier & return it, otherwise places the identifier

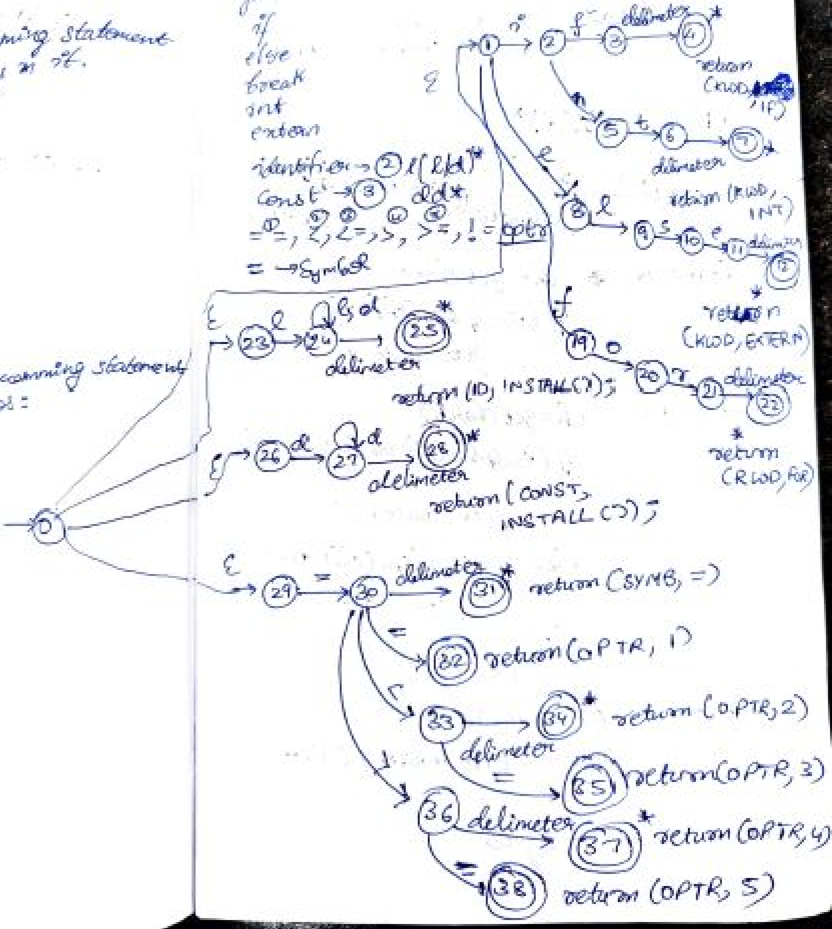
into Symbol table & returns its address.



...ning statement
...s is %.

...scanning statement
...as:

Keywords → ①
if
else
break
int
extern
identifier → ② l(l/d)*
Const → ③ dd*
==, <, <=, >, >=, != optr
= → Symbol

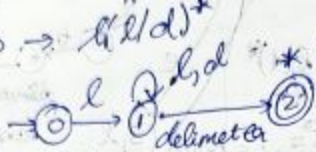


ε-NFA → DFA → minimize DFA

Leads to redundancies

so we will minimize DFA

ID → l(l/d)*



State 0; ch = getchar()
if (isalpha(ch)) then
goto state 1;

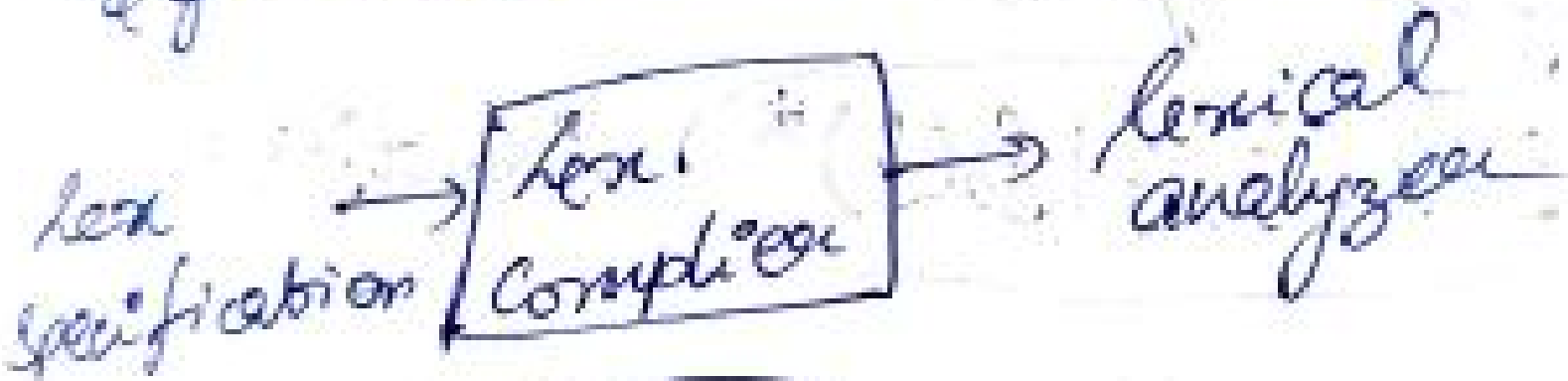
State 1: ch = getchar()
if (isalpha(ch) or isdigit(ch)) then
goto state 1;
else if (isdelim(ch)) then
goto state 2;
else
error();

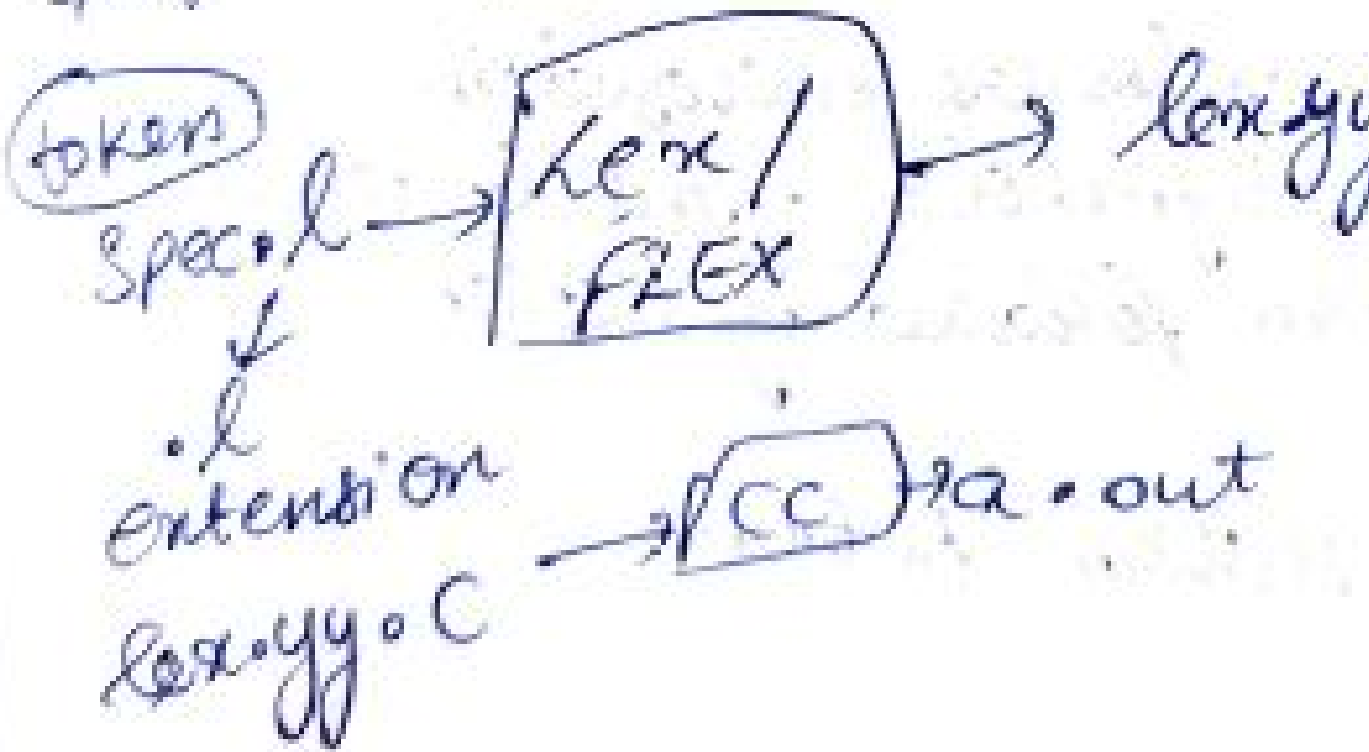
State 2: RETRACT();
return (ID, INSTALL());

First Compiler → 20 yrs

Lex Tool

* It is used to generate lexical analyzer.
For lex compiler give input lex specification
& give output lexical analyzer





Lex specification — Declaration section — Auxiliary declaration
Regular expression definitions or declaration
Transition rules
Auxiliary function → Diff function used by Transition rules
Auxiliary declaration → Just like c declaration which going to use in c

Reg. exp declaration

```
%{
int a = 10;
%}
```

Auxiliary declaration

l [a..zA..Z]
↓
Stands for only one symbol

d [0...9]
↓
Only one symbol

id ([l]([l]|[d])*) → identifier

Transition rules are enclosed in double %. symbols. Inside we can give pattern followed by action.

```
%%
    Pattern Action.
%%
```

Whatever defined put in curly braces
what not defined we can give in
[ ]

```
%%
  [a-z]+   { printf("%s is an id", yytext); }
%%
```

Global variable declared by lex Compiler

Auxiliary functions
↳ Simple c functions
↳ whatever functions written in this copied to lex.yy.c

Transition rules

~~%% [a-z]+ { printf("welcome"); printfun(yytext); }~~

```
%%
  [a-z]+   { printf("welcome"); printfun(yytext); }
%%
```

Auxiliary function

```
void printfun(char *str)
{
  printf("%s is identifier", str);
}
```

Calling lex compiler

int main()

{

yylex();

}

Declared Transition Annotations

↓

specification.l

↳ If pattern matches only pointed function called