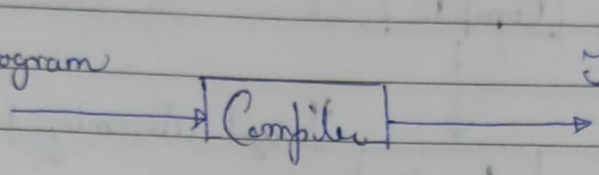
Compiler Design

Chapter 1 : Introduction : Lang processor, Struct of Compiler, Introduction & Lexical Analysis :-

* Language processor

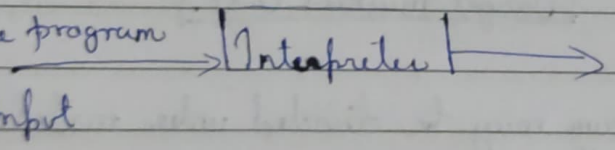
Compiler - is a

A Compiler program takes a program written in a source language & translates it into equivalent program in a target language.



Interpreter

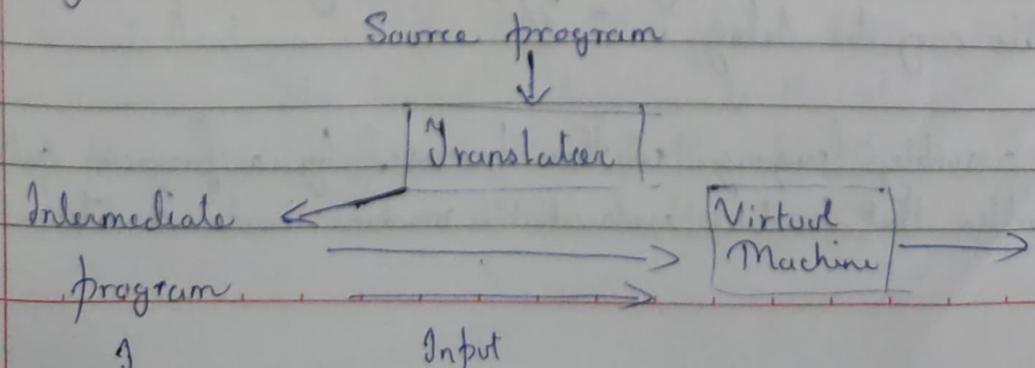
An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appear ite directly execute the operations specified in the program on the inputs supplied by the user.



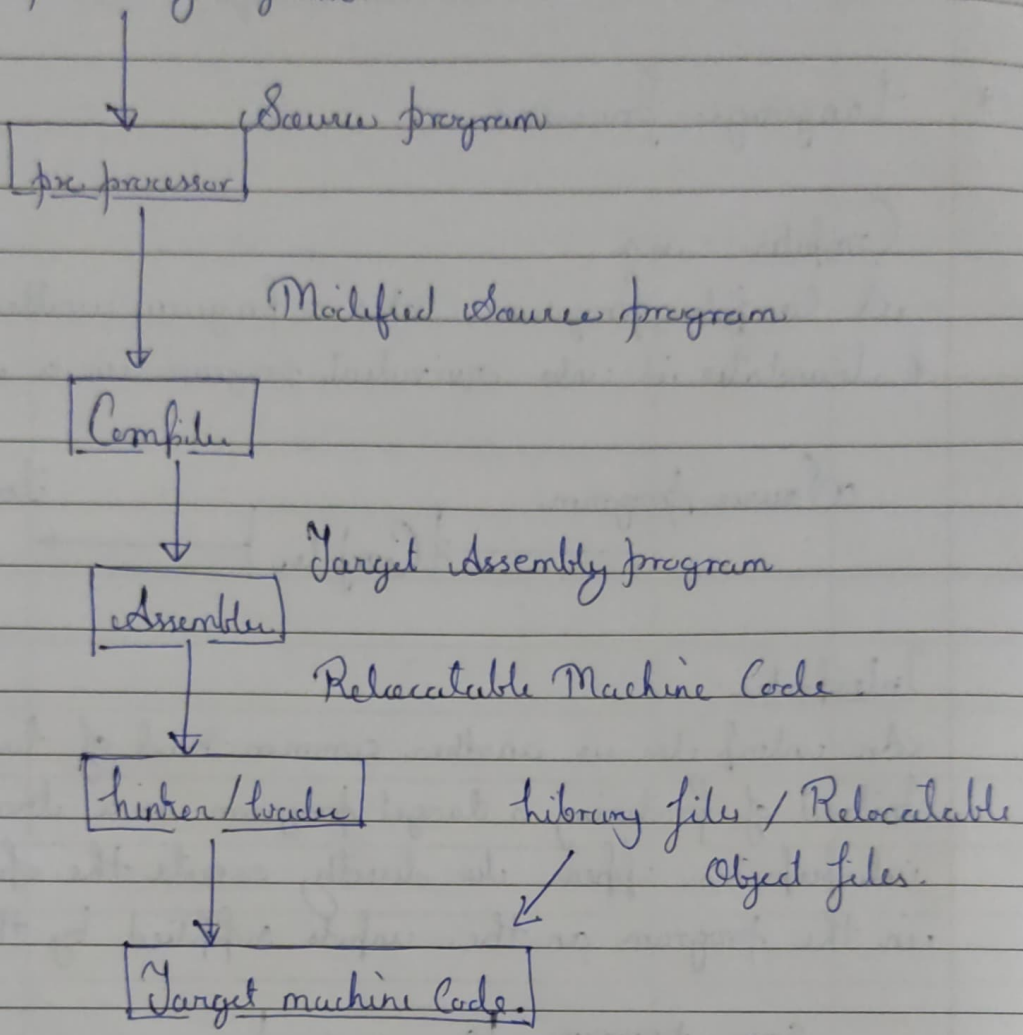
An Example of Compiler & Interpreter Combination

Java language processor combine compilation and interpretation.

A java Source program is first be compiled into an intermediate form called Byte Code. The byte Codes are then interpreted by a Virtual machine.



Language processing system



A source program may be divided into modules stored in seperate files. The task of collecting the source program is sometimes entrusted to a simple program called a preprocessor. The preprocessor may also expand short hands called macros

// * The modified source program is then fed into a compiler. The compiler may produce an assembly language program as its output because assembly language is easy to produce output and is easy to debug. * //

- The assembly language is then processed by a program called assembler that produces relocatable machine code as its output.

- Large programs are often compiled in parts, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The linker resolves the external memory addresses with the code in one file may refer the location in another file.
- The loader then puts together all of the executable object files into memory for execution.

* Structure of Compiler

Two parts of Compiler (Grouping of phases)

1) Analysis (Front End) 2) Synthesis (back end)

→ The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

It also collects info about the source program and stores it in data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part

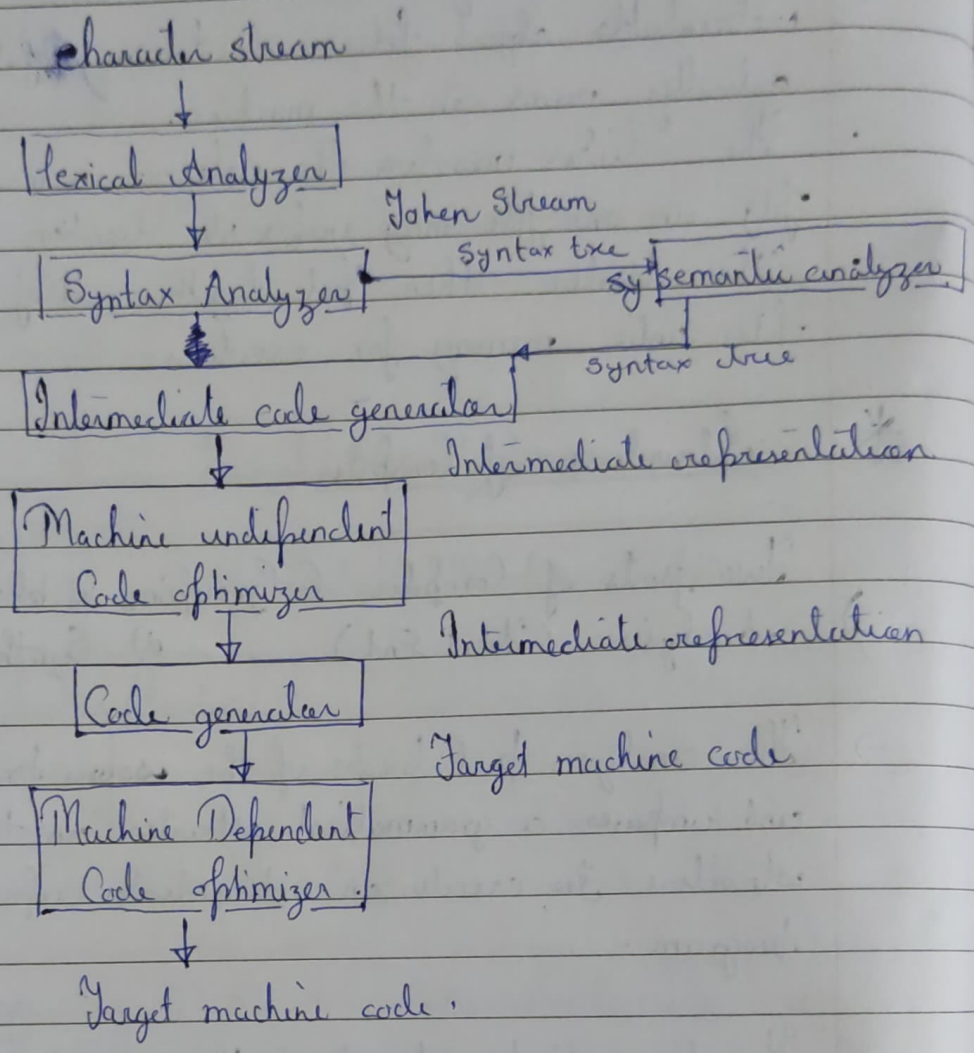
→ The Synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

1) Lexical Analyzer :- The first phase of a Compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as

3/03/21

Structure of a Compiler

output a token of the form, <token_name, attribute value>



Example.

position = initial + rate * 60;

i) position is a lexeme, mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position.

ii) '=' is mapped into <=> token. Since it doesn't need any attribute value, we omit the second component.

iii) Similarly

→ initial lexeme is mapped into token <id, 2>

→ '+' is mapped into <+> token

→ Rate is mapped into token <id, 3>
→ * is lexeme is mapped into token <*>
→ 60 is mapped into token <60>

∴ After lexical analysis, the token stream (o/p) is

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

2) Syntax Analyzer:

This is the 2nd phase of compiler, also known as parsing.
The parser uses the first components of tokens produced by the lexical analysis, create a tree like intermediate representation that shows the gramatical structure of the token stream

→ Typically each interior represents an operation
→ the children of the node represents the argument of the operation

3) Semantic Analysis:

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition

→ This phase involves type checking where the compiler checks that each operator has matching operands.

4) Intermediate Code Generator

In this the process of translating a source program into target code a compiler may construct one or more intermediate representations

which can have variety of forms.

One possibility of intermediate form called 3 address code which consists of a sequence of assembly like instructions each operand can act like a register.

5) Code optimization

The machine independent code optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster but other objectives may be designed such as shorter code or target code that consumes less power.

6) Code Generation:

The code ~~generation~~ generator takes an input an intermediate representation of the source program and maps it into target language. If the target language is machine code registers or memory locations are selected for each of the variables by the programs, then the intermediate instructions are translated into sequence of machine instructions that perform the same task.

7) Symbol Table

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

These attributes can provide the information about storage allocated for a name its scope and in case of procedure name such as number and types of its arguments the method of passing each argument and the type returned.

Example: position = initial + rate * 60

↓

[lexical Analyzer]

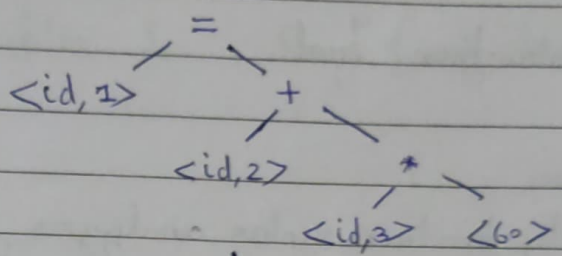
↓

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

↓

[Syntax analyzer]

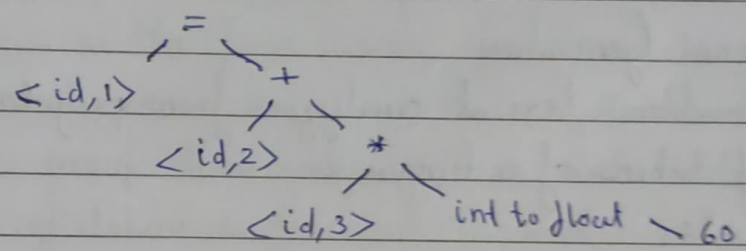
↓



↓

[Semantic analyzer]

↓



↓

[Intermediate Code Generator]

↓

i) t1 = int to float (60) ii) t2 = id3 * t1

iii) t3 = id2 + t2

iv) id1 = t3

↓

[code optimizer]

↓

t1 = id3 * 60.0

id1 = id2 + t1

↓

[Code generator]

↓

PTO

Date / /

```
LDF     R2, id3
MULF    R2, R2, #60.0
LDF     R1, id1
ADDF    R1, R1, R2
STF     id1, R1
```

Compiler Construction Tool

1) Parser Generator
It automatically produce syntax analyzers from a grammatical description of a programming language.
Eg. Yacc

2) Scanner Generator
It produces lexical analyzers from regular expressions description of tokens of a language.
Eg. LEX

3) Syntax directed translation Engines:
It produces collection of routines for walking a parse tree and generating intermediate code.

4) Code Generators Generators
It produces a code generator from a collection of rules for translating each operation of the intermediate language into machine language for a target machine.

5) Data flow analysis Engines:
It facilitate the gathering of information about how values are transmitted from one part of a program to each other part.

Data flow analysis is the key part of code optimization.

6) Compiler Construction tool kits:
It provides an integrated set of routines for constructing various phases of a compiler.

End of Chapter 1 - Easy peasy lemon

⇒ Chapter 2

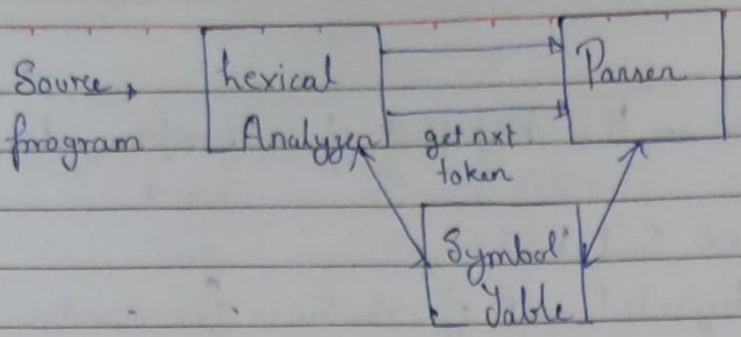
Lexical Analysis: The role of lex analyzer

The role of lexical analyzer:-

→ lexical analyzer is the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemas and produce as output the sequence of tokens as for each lexema in the source program.

→ The stream of tokens is sent to the parser for syntax analysis It is common for lexical analyzer to interact with the symbol table when the lexical analyzer identifies lexema constituting an identifier, it needs to enter that lexema into the symbol table.

→ The call suggested by the get next command shown in figure causes the lexical analyzer to read character from its input until it can identify the next lexema and produce for it the next token, which it returns to the parser.



lexical analyzer also performs the following additional fn's

i) produces a stream of tokens
ii) Eliminates blanks and comments
iii) generates symbol table which stores the info about identifiers constants encountered in the input.
iv) keeps tracks of line numbers
v) reports the error encountered while generating the tokens

lexical Analysis Vs Parsing

1) Simplicity of Design is the most imp consideration.
The seperation of lexical and syntatic analysis often allows us to simplify atleast one of these tasks.
eg. A parser that had to deal with comments and white spaces as syntatic units would be considerably more complex that one can assume white space and comments have already been removed by the lexical analyzer.

2) Compiler efficiency is improved:
A seperate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job parsing

3) Compiler portability is enhanced:
Input device specific peculiarities can be restricted to the lex analy

Tokens, Patterns & Lexemes

* **Tokens :-** A token is a pair consisting of a token name and an optional attribute value. The token name is a abstract symbol representing a kind of lexical unit.

Ex: A particular keyword or a sequence of input characters denoting a identifier.
The token describes the class or category of input stream.
ex: Identifiers, keywords, constants.

* **Patterns :-** Is a description of the form that the lexemes of a token may take.
In case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

For identifier and some other tokens, the pattern is a more complex structure that is matched by many string.

* **lexemes:** Is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

ex. 1

→ `printf("Total = %d\n", score);`

Total lexemes = 7.

i) printf & score are lexemes matching the pattern for token identifier
ii) "Total = %d \n" is a lexeme matching literals

→ `if (a<b)`
Here,

if, (a, <, b,) are all lexemas

and if is a keyword
(" is a opening parentheses
a is an identifier
< is a operator
And so on are all tokens.

Sample lexemes	Token
if	if
else	else
<=, !=	comparison operators
3.1415, 1.141	constant
"Not found", "Code dumped"	literals

Attributes for Tokens

* When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.

Ex. the pattern for token number matches both 0 & 1, but it is extremly important for the code generator to know which lexeme was found in the source program

* In many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token. The token name influences parsing decisions

Ex E = m * c ** 2

<id, pointer the symbol-table entry for E>
<assign_op>
<id, pointer the symbol-table entry for m>
<mult_op>
<id, pointer the symbol-table entry for c>
<exp_op>
<number, integer value 2>

Lexical Errors

It is hard for a lexical analyzer the tell, without the aid of other components, that there is a source code error.

eg if the string fi is encountered for the first time in a C program in the context

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

a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lex analyzer must return the token id to the parser and let some other phases of the compiler.

The simplest recovery strategy is 'panic mode' recovery.
→ Delete successive characters from the remaining input until the lexical analyzer can find a well formed token at the begining of what input is left.
→ This technique might confuse the parser

Other possible Error recovery actions

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input
3. Replace a character by another character.
4. Transpose two adjacent characters.

→ The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation.

→ A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes.

Input buffering → Sentinels lookahead Code (do after page no

```
switch (*forward++) {
    case eof:
        if (forward is at end of first buffer) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

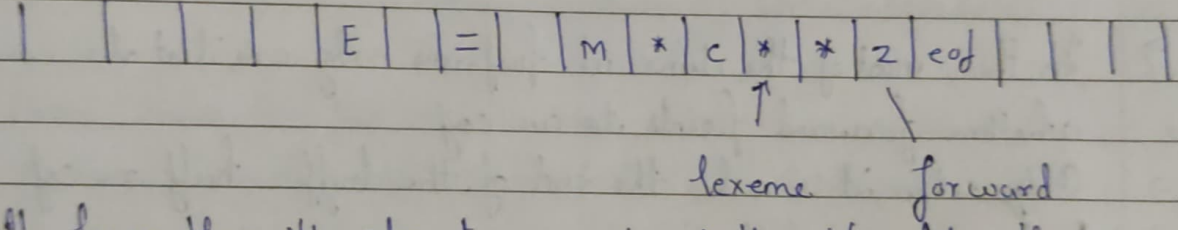
Input Buffering

Buffer pairs

- The amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized by buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

- An important scheme involves two buffers that are alternatively reloaded. Each buffer is of the same size n & n is usually the size of a disk block.

Example: 4096 bytes.



If fewer than N character remain in the i/p file, then a special character represented by EOF marks the end of the source file.

- Two pointers to be the inputs are maintained.
- Pointer lexeme begin, marks the beginning of the current lexeme whose extent we are attempting to determine.
2. Pointer forward, scans ahead ~~until~~ until a pattern match is found.

- Once the next lexema is determined, forward is set to the character at its right end, then after the lexema is recorded as an attribute value of a token returned to the parser, the lexema begin is set to the character immediately after the lexema just found:

Sentinels

→ In the previous scheme we must check each time the move forward pointer that have not moved off one half of the buffer. If we it is done, then we must reload the other half

→ Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

→ This can reduce the two tests to one if it is extend each buffer half to hold a sentinal character at the end.

→ The sentinal is a special character that cannot be part of the source program.

			E		=		M	*	eof	C	*	*	2	eof	eof		

→ In this, most of the time it performs only one test to see whether forward points to an eof.

→ Only when it reaches the end of the buffer half or eof, it performs more tests.

→ Since N input characters are encountered between eof's, the average number of tests per input character is very close to 1.

Specifications of Tokens

Strings & languages.

→ A string is a collection of finite number of alphabets or letters

→ language is a collection of string defined on Σ

Operations on languages

PTO

Operation	Definition and Notation
Union of L & M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
Concatination of L & M	$LM = \{ st \mid s \text{ is in } L \text{ & } t \text{ is in } M \}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular Expressions

→ Some notes Describing identifiers by giving names to sets of letters and digits by and using the language operators union, concatination and closure. The usefulness of this process has led to the excessive use of notations called Regular Expressions for all languages that can be described or built from these operators applied to the symbols of some alphabets.

Basic Rules for Reg Ex

1. ϵ is a regular expression & $L(\epsilon)$ is $\{\epsilon\}$.
2. If a is a symbol in Σ, then a is a regular expression, & $L(a) = \{a\}$

Induction

(r)(s) is a regex denoting $L(r) \cup L(s)$
(r)(s) is a regex denoting $L(r)L(s)$
$(r)^*$ is a regex denoting $(L(r))^*$
(r) is a regex denoting $L(r)$

Some dumb stuff that probably won't be asked

1. a, ϕ, ϵ are all primitive reg Ex
2. If R_1 & R_2 are regex then $R = R_1 + R_2$ is also a reg Ex
   $(R = R_1 | R_2)$
3. If R_1 & R_2 are regex, $R = R_1 \cdot R_2$

4. If R_1 is a regEx, then R_1^* is also a regEx.

Regular Definations

For notational Convinence:

If Σ is an alphabet of basic symbols, then the regular defination is a sequences of definations of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$d_n \rightarrow r_n$

where

1. each d_i is a new symbol, not in Σ & not the same as any other of the d's.
2. Each r_i is a RegEx over the alphabet $\Sigma \cup \{d_1, d_2, d_3 \ldots d_{i-1}\}$

eg 1: eg Unsigned numbers are strings such as 5280, 0.0123 etc
The regular definations are

digit → 0|1|...9

digits → (digit)(digit)*

optional fraction → .digit | ε

optional exponent → (E (+|-|ε) digits) | ε

number → (digits)(optional fraction)(optional exponent)

eg. 2. C identifiers are strings of letters, digits & _. The regex def.

letter_ → A|B|C|D.... Z|a|b|c|...z|_

digit → 0|1|2|3|4|...9

id → letter_(letter_|digit)*

Date / /

Extensions of Regular Expression

1) One or more instances:
The unary postfix operator '+' represents the positive closure of a regular expression, that is if r is a regular expression, then $(r)^+$ denote the language $(L(r))^+$.
The operator '+' has the same precedence and associativity as the operator '*'.

Two laws
$r^* = r^+ | \varepsilon$. $r^+ = r.r = r^*.r$

2) Zero of One instance:
The unary postfix operator '?' means zero or one occurrence, that is, r? is equivalent to $r | \varepsilon$
and
$L(r?) = L(r) \cup \{\varepsilon\}$
The ? operator has the same precedence and associativity as * & +

3) Character classes:
A regular expression $a_1 | a_2 | a_3 | \ldots | a_n$
where a_i are each symbols of the alphabet, can be replaced by short hands.
$[a_1 a_2 \ldots a_n]$
Eg. [abc] is shorthand for a|b|c
[a-z] is shorthand for a|b|c|...|z

→ Example:
Using the shorthands rewrite regex for unsigned int.
digit → [0-9]
digits → digit+
number → digits (.digits)? [E[+-]? digits)?

eg. 2 letter_ → [A-Za-z_]
digit → [0-9]
id → letter_(letter_ | digit)*

Recognization of Tokens

Algebraic laws for Regular Expressions.

laws	Description
$r \mid s = s \mid r$	| is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt$; $(s \mid t)r = st \mid tr$	Concatenation distributes over |
$\varepsilon r = r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r \mid \varepsilon)^*$	ε is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Recognisation of Tokens

Consider a Grammar for branching statements

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

expr → term relop term
| term

term → id | number

The terminals of the Grammar, which are if, then, else, relop, id and number are the names of token. The patterns for these tokens are described using regular definition

digit → [0-9]
digits → digit +
number → digits (. digits)? (E [+-]? digits)?
letter → : letter (letter | digit)*
if → if
else → else
then → then
relop → < | > | <= | = | <>

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" ws defined by:

ws → (blank | tab | newline)+

Tokens, their patterns & attribute value

lexemes	Token Name	Attribute value
any ws		
if	if	
then	then	
else	else	
any id	id	pointer to table entry
any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ

<>	relop	NE
>	relop	GT
>=	relop	GE

Transition Diagram

As an intermediate step, the construction of a lexical analyzer will first convert pattern into ~~transaction~~ its transition diagram.

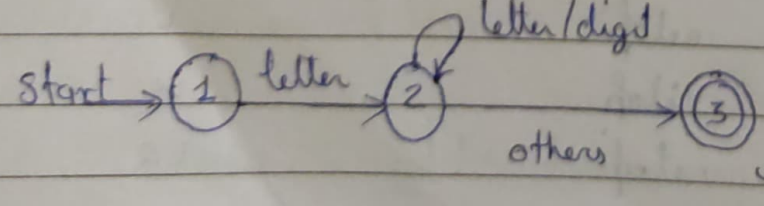
Transition diagram have a collection of nodes or circles called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges are directed from one state of the transition diagram to other. Each edge is labelled by a symbol or set of symbols. It should also know lexeme begin pointer and the forward pointer ~~End~~

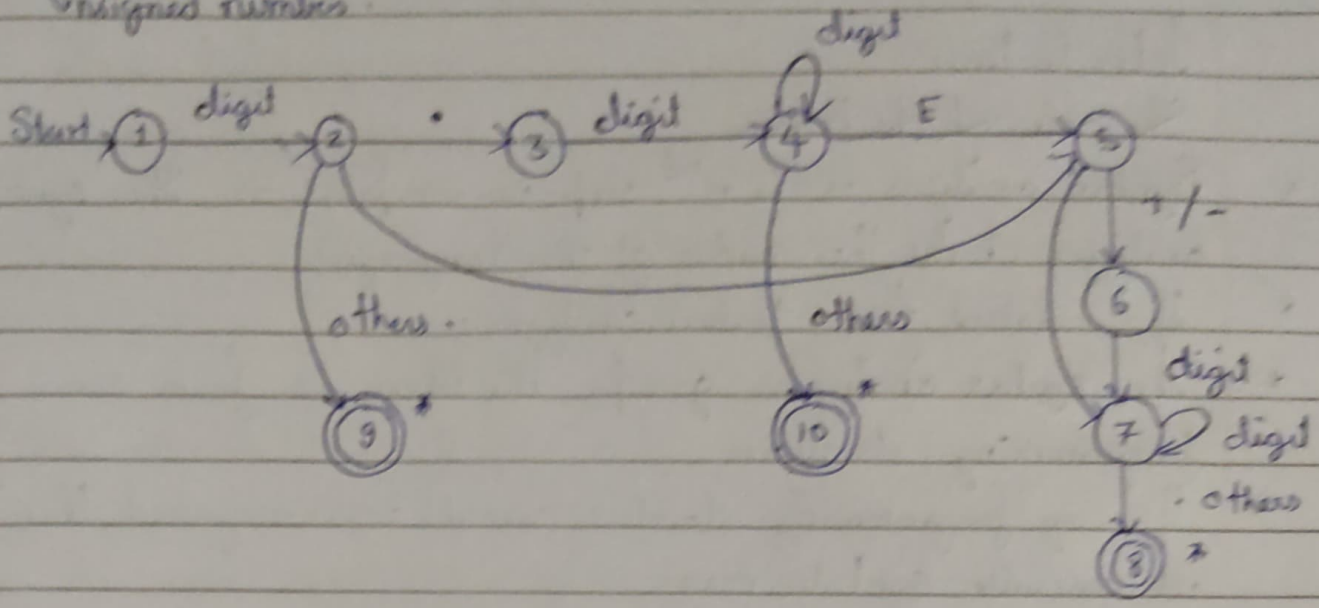
Certain states are said to be accepting or final indicated by double circle.

One state is designated the start state or initial state

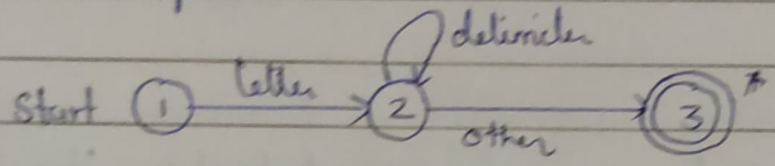
Example.
Construct a transition diagram for the following
ii) Identifiers / keywords



iii) Unsigned numbers



iii) white Spaces



iv) Relational op / relop