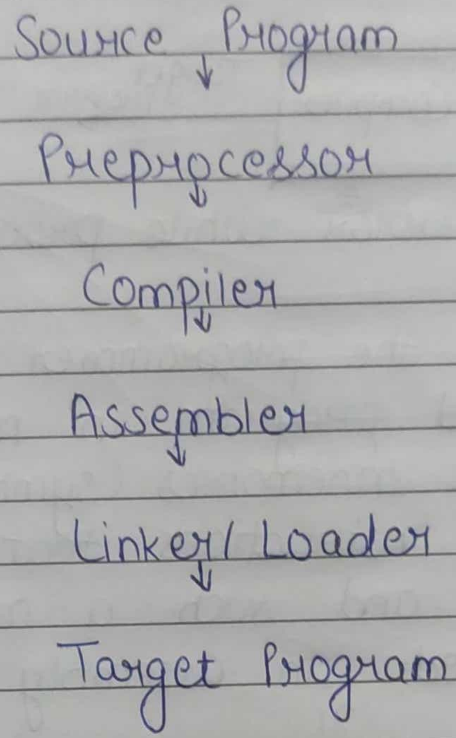
Compiler Design

17/02/22

CSE-602

Overview of Language Processing System



Preprocessor :- A preprocessor produce input to compiler they may perform the following functions.

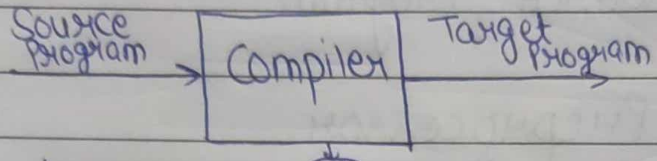
1. Macro Processing :- A preprocessor may allow a user to define macros that are shorthands for longer constructs.

2. File Inclusions :- A preprocessor may include header files into the program text.

3. Rational Preprocessor :- These processor add older languages with more modern flow of control and data structure facilities.

4. Language Extensions :- These preprocessor attempts to add capabilities to the language by certain amount to build macro.

Compiler- A compiler is a translator program that translates a program written in high level language and translate it into the equivalent program in machine level language as a target program.



A compiler display error while processing the program.

Assembler- Whenever the programmer find it difficult to write or read program in machine language they began to use mnemonics (symbols) for each machine ~~language~~ instructions that translate into machine language and such a mnemonics machine language is called an assembly language.

Interpreter- They convert high level language into the machine level language line by line such as BASICS, COBOL, FORTRAN.

Advantages:- 1. The modification of user program can be easily made and implemented as execution proceed.

2. The type of object that denotes at various level may change dynamically.

3. Debugging of program and finding errors is simplified task for a program used to interpretition.

4. Interpreter of the language makes it machine independent.

Disadvantages- 1. Slow execution of the program.

2. The memory consumption is high.

18 / 02 / 22

Linker / Loader:- All the files combined in the form of either linker or loader.

Without the help of Linker / Loader we can't execute our program.

List of compilers

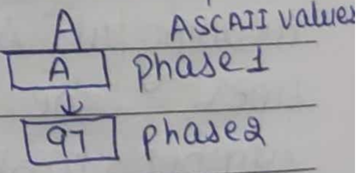
1. . ALGOL Compiler
2. . BASIC compiler
3. . C# Compiler
4. . C compiler
5. . C++ compiler
6. . COBOL Compiler
7. . D Compiler
8. . COMMON LISP compiler
9. . FORTRAN Compiler
10. . JAVA Compiler
11. . PASCAL Compiler
12. . PL/1 Compiler
13. . PYTHON Compiler
14. . SCHEME Compiler
15. . FELIX Compiler
16. . EFFIL Compiler
17. . ADA Compiler
18. . Small Talk Compiler

Structure of the Compiler Design

A phase is a logicaly inter-related operation that takes source program in one representation and produce Output in another representation.

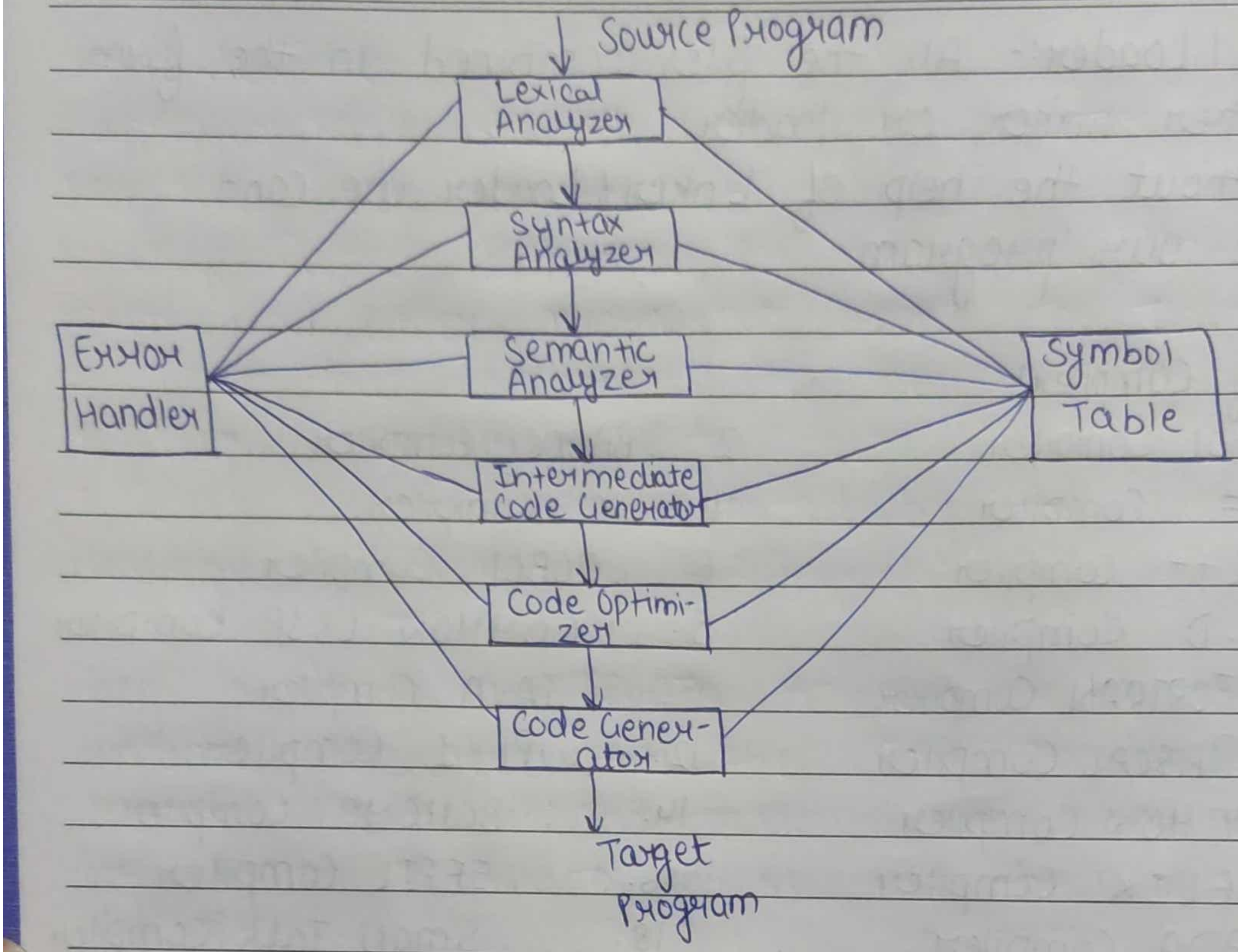
There are normally two phases in the compiler

1. Analysis
2. Synthesis



There are 6 phases of the compiler

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator



Lexical Analysis- Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character and group these characters into tokens. The character stream from the source program is grouped in meaningful sequences by identifying the tokens. Sometimes the entry of corresponding tokens into the symbol table and passes that token into the next phases of the compiler.

The primary function of these phase are
1. Identify the Lexical Unit in a source code.
2. Classify Lexical unit into classes like constant, reserve word or different type of sections we have
It will ignore comment section of the source program.

3. Identify token which is not a part of the language.
for Eg X = Y+10

Token	Description
X	Identifyier
=	Assignment Operator
Y	Identifier
+	Additional Operator
10	Number

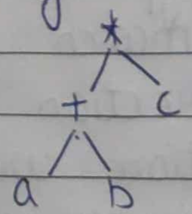
Syntax Analysis - Syntax Analysis is all about discovering structure in a code. It determines whether or not or text follow the aspected format. The main aim of these phase is to make sure that source code was written by the programmer is correct or not.

It is based on the rules regarding specific programming language by constructing the Parse Tree with the help of tokens.

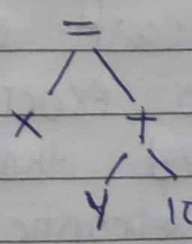
The list of tasks performed in this phase.

1. Obtain tokens from Lexical Analyzer.
2. Check if the expression is syntactily correct or not.
3. Report all syntax error.
4. Construct or Hierarchical structure which is known as Parse Tree.

For Eg. (a+b) * c



X = Y+10



		/ /

In Parse Tree

1. Interior Node - Record with and operator field and two fields for children.
2. Leaf - Record with two or more fields one for token and other information about the token.
3. Ensure that components of the program fit together meaningfully.
4. Check operands are permitted by the source language.

Semantic Analysis - Semantic Analysis checks the semantic consistency of the code. It use the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also check wheather the code is represent and appropriate meaning.

Semantic Analyzer will check type mismatch, incompatiable operands a function called within proper arguments, undeclared variable etc.

There are various functions used in semantic Analyzer.

1. It helps you to store type information gathered and save it in the system symbol table or syntax tree.
2. It allows you to perform type checking.
3. In the case of type mis-match if there are no exact type of correction rule which satisfied the desired operation will generate the semantic error occur error handler.
4. Collects type information and check for type

compatibility.

5. It also check if the source language permits the operand or not.

Eg- Float x= 20.2;
y= x* 30;

In this code semantic analyzer type caste to the integer 30 to float 30.0 before multiplication.

Intermediate Code Generation- Once the Semantic Analysis is over the compiler generates intermediate code for the target machine.

It generates a program for some abstract machine.

Intermediate code is between the high level language and machine level language.

The intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

y= x* 30;
variable= variable 2 * constant

The following function can be done on intermediate code generator.

1. It should be generated from the source program.

2. It holds the value computed during the process of translation.

3. It helps you to translate the intermediate code into target language.

4. It allow you to maintain precedence ordering of the source language.

5. It holds the correct no. of operands of

the instruction.

For Eg Total= Count + rate * 5

The following task will perform by the intermediate code with the help of the following address code method.

```
t1:=    int_to_float(5)
t2:=    Rate * t1;
t3 =    count + t2;
total=     t3;
```

Code Optimization- In this phase we removes un neccessary code lines and arranges the sequence of statement to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space. The primary function of this phase are -

1. It helps you to establish trade-off between execution and compilation speed.
2. It improves the running time of the target program.
3. Removing unreachable code and eliminate unused variables.
4. Removing statements which are not altered from the loop.

```
a= int to float (10);
b= c* a;                b= c* 10.0;
d = f+b;                k = f+b;
k = d;
```

Code Generation It is the last and final phase of the compiler. It gets input from code optimization phase and produce the object code or page code as a result. And the objective of this phase is to allocate storage and generate re-locatable machine code. All the memory locations and registers are also selected and alloted respectively.

for example

```
a=b+ 60.0;
MOVF b. R1;
MOVF #600, R2;
ADDF R1, R2;
```

Example- position := initial + rate * 60

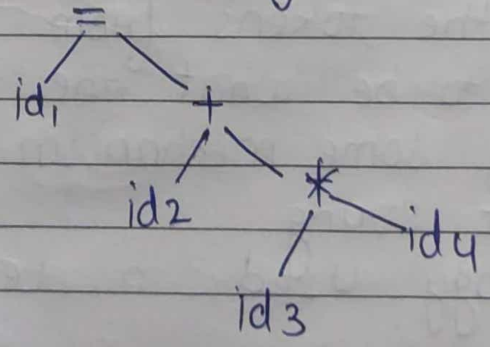
↓

Lexical Analyzer

$id_1 := id_2 + id_3 * id_4$

↓

Syntax Analyzer



↓

Semantic Analyzer

↓

X — 60

↓ int to real

intermediate code generator

↓

```
temp1 := int to real (60);
temp2 = id3 * temp1;
temp3 = id2 + temp2;
id1 : = temp3
        ↓
    code optimizer
        ↓
    temp 1 = id3 * 60.0
      id1 = id2 + temp1
        ↓
    code generator
        ↓
    MOVF id3, r1
    MOVF #60, r2
    MULF r2, r1
    MOVF id2, r2
    ADDF r2, r1
    MOVF r1, id1
```

1. To identify the tokens from an input string.
2. It is decide to be what the tokens are on there is a need of some mechanism to recognize token from the input string.
3. The Termonology used in Lexical Analysis-

(i) Token
(ii) Lexeme
(iii) Pattern

Token:- The Token is a sequence of characters that can be treated as single logic entity.
for Eg- Identifiers, Keywords, Operators, Special Symbol

/ /

constant etc.

Pattern :- It is a set of strings in the input
OR
The set of strings is described by a rule called pattern associated with the tokens.

Lexeme:- It is a sequence of the characters in the source program that is matched by the patterns for a token.

Token	Lexeme	Pattern
Const	Const	Constant
If	If	If
Relation	$<, <=, \neq$	Less than, Less than equals to, Not equal to
I	Pi	Any numeric constant
Literal	" "	

Lexical Errors -
Lexical Errors is the errors that can be generated by the lexer and after that we are not able to continue next process.
No way to recognize Lexeme as a valid token.

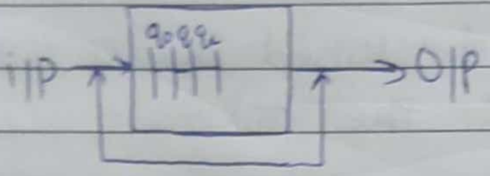
Regular Expressions:-
$\cdot + *$

R^* - $\wedge$, R, RR, RRR - - - - -

Even no. of zero's RE = $00 . (00)^*$

identifier= letter. (Digit + Letter)*

Automata



NDFA - Q, Σ, δ, q_0, F

Q - No. of states

Σ - input

δ - Transition function

q_0 - Initial state

F - final state

δ - Transition Table

" Diagram

" function

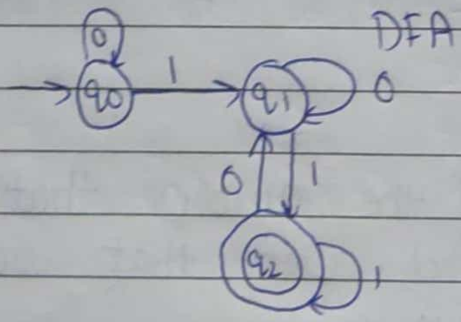
DFA - $Q \times \Sigma \to Q$

NDFA - $Q \times \Sigma \to 2^Q$

$\delta(q_0, 0) = q_0$

$(q_0, 1) = q_1$

	0	1
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_1	q_2



Lex specification - Lex program consists of three parts

1. Declaration xx
2. Translation Rule %.%
3. Auxiliary Procedure ÷ %.

Declaration - In the declaration section includes the declaration of variable and manifest constant and regular definitions.

Translation Rule- Translation Rule of a Lex program are statements of the form.

Pi {action 1}

P_2 {action 2}

Where P is a Regular Expression and each action is a program fragment describing what action the Lexical Analyzer should take when a pattern p match a lexeme.

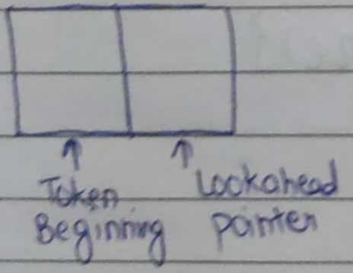
Auxiliary Procedure- The Third section holds whatever the Auxiliary procedures are needed by the action.

Input Buffering :- The lexical Analyzer scans the character of the source program one at a time to discover tokens. Because of large amount of time can be consume scanning character specialized buffering techniques had been developed to reduce the amount of overahead required to process input characters.

I. Buffering Techniques are -

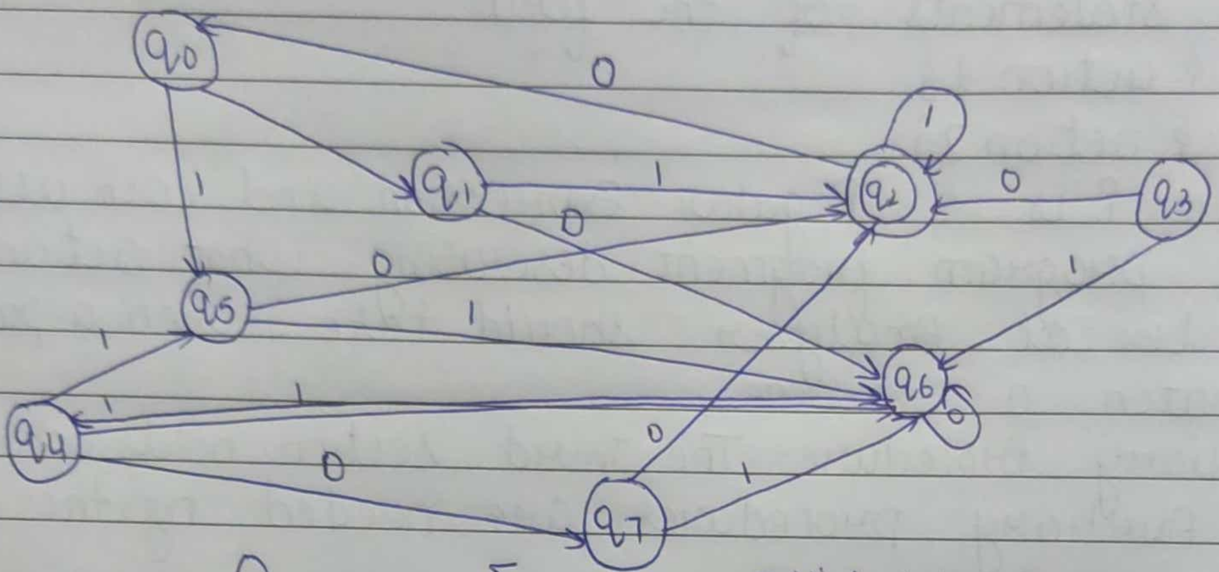
1. Buffer Pair

2. ~~Sentimentals~~ Sentinels



In this figure we take buffers divided into two halfs suppose 100 each one pointer marks at the beginning of the token being discovered a lookahead pointer scans ahead at the beginning point until the token is discovered.

Minimization of FA



$$M = \{ \overset{Q}{(q_0 - - q_7)}, \overset{\Sigma}{(0,1)}, \delta, \overset{\text{Initial state}}{q_0}, \overset{\text{final state}}{(q_2)} \}$$

$$\Pi_0 = \{ Q_1^0, Q_2^0 \}$$
$$= \{ (q_2), (q_0, q_1, q_3, q_4, q_5, q_6, q_7) \}$$

$$\Pi_1 = \{ (Q_1^1, Q_2^1) \}$$
$$= \{ (\overset{Q_1^1}{q_2}), (\overset{Q_2^1}{q_0, q_4, q_6}), (\overset{Q_3^1}{q_1, q_7}), (\overset{Q_4^1}{q_3, q_5}) \}$$

$$\Pi_2 = \{ (Q_1^2, Q_2^2) \}$$
$$= \{ (\overset{Q_1^2}{q_2}), (\overset{Q_2^2}{q_0, q_4}), (\overset{Q_3^2}{q_1, q_7}) (\overset{Q_4^2}{q_3, q_5}), (\overset{Q_4^5}{q_6}) \}$$

Q \ Σ	0	1
q0	q1	q5
q1	q6	q2
q2	q0	q2
q3	q2	q6
q4	q7	q5
q5	q2	q6
q6	q6	q4
q7	q6	q2

$$\Pi_3 = \{ Q_1^3, Q_1^3 \}$$
$$= \{ \underset{A}{(q_2)}, \underset{B}{(q_0, q_4)}, \underset{C}{(q_1, q_7)}, \underset{D}{(q_3, q_5)}, \underset{\epsilon}{(q_6)} \}$$

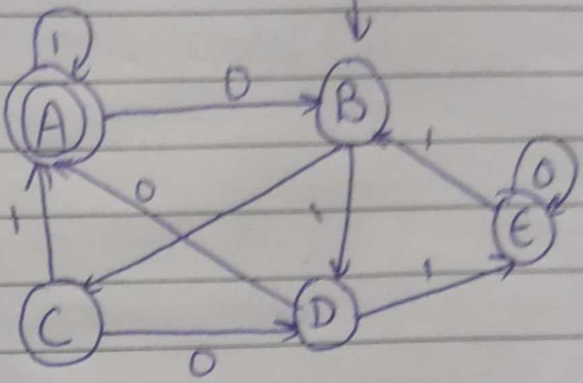
$$\pi_{n+1} = \pi_n$$

$$M' = \{ \overset{Q'}{(A, B, C, D, \epsilon)}, \overset{\Sigma}{(0,1)}, \delta', q_0', F' \}$$

Initial state →

Q \ Σ	0	1
B	C	D
C	E	A
D	A	E
E	E	B
A	B	A

↑ final state



A
B
C
D
E
0
1
0
1
1
0
1
0
1