16/1/18

MODULE - 1

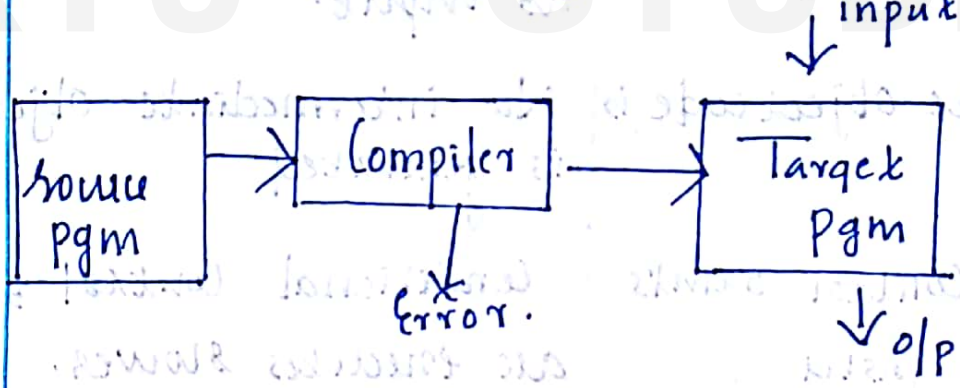
Compiler: It is a pgm that reads a pgm written in high level language (source language) and translates it into an equivalent pgm in another language [target language].

An imp. role of compiler is to report any errors in a source pgm that it detects during the translation process.

NOTE

Compiler is a system software

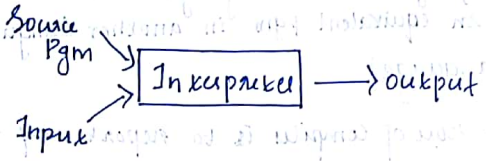
Compiler is a language translator



eg: C compiler, Java c, Dempo C, ...

Interpreter: An Interpreter another common kind of language processor. Instead of producing a target pgm an interpreter directly execute the operat^n specified by in the of source program.

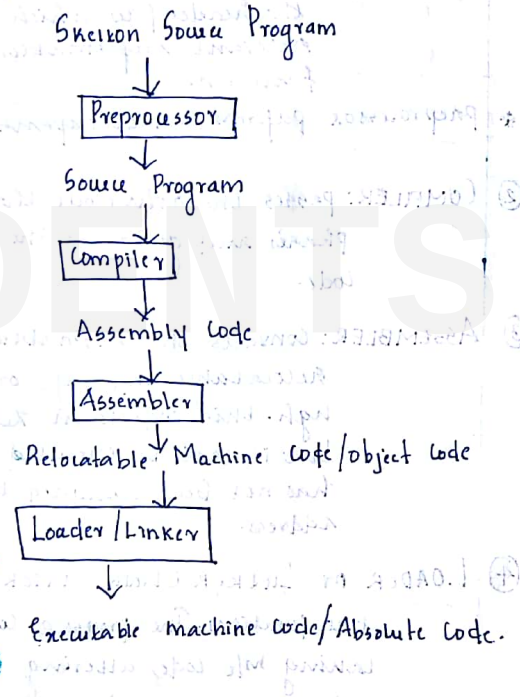
The m/c language target program produced by a compiler is usually much faster than an interpreter at maping inputs to outputs.



Difference b/w Compiler & Interpretor

Compiler	Interpretor
• It takes entire progm as input.	It takes single instruct as input.
• Intermediate object code is generated	No intermediate object code is generated
• Conditional control stmts are executes faster	Conditional control stmts are executes slowers.
• M/y requirement is more (Since object code is generated)	M/y requirement is less
• Pgm need not be compiled everytime	Everytime higher level pgms is converted into lower level Pgms
Errors are displayed after entire pgm is checked	Errors are displayed for every instruct by interpreter
eg: C, C++ languages uses compilers.	Pgming languages like Python, ruby uses interpreter

Context of a Compiler:



Assembler: is a language translate assembly code into machine language.

2 marks

The prgms which assist the compiler to convert a skeleton source code into executable form make the context of a compiler.

① PREPROCESSOR: Scans the sourcecode and includes the header files which contain relevant information for various functions.

* Preprocessor performs macroexpansion also.

② COMPILER: passes the sourcecode through various phases and generates the target assembly code.

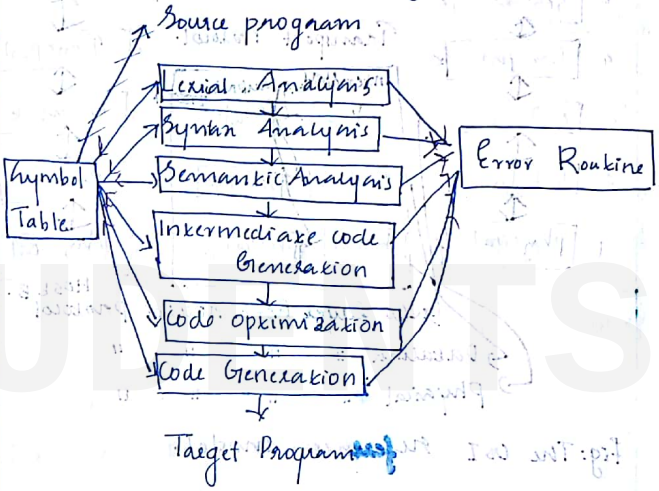
③ ASSEMBLER: converts the assembly code into relocatable m/c code or object code. Although, this code is in zero and one form. but it can't be executed Bcz this code has not been assigned the actual m/y address.

④ LOADER or LINKER (LINK EDITOR): It performs two functions. The process of loading consist of taking m/c code, altering the relocatable address & placing the altered instruction in Memory at proper location.

The linkeditor makes the single progm from several files from relocatable codes. These file are library files the pgm needs. The loader produces the executable or absolute machine code.

→ Phases of Compiler Design:-

10 marks



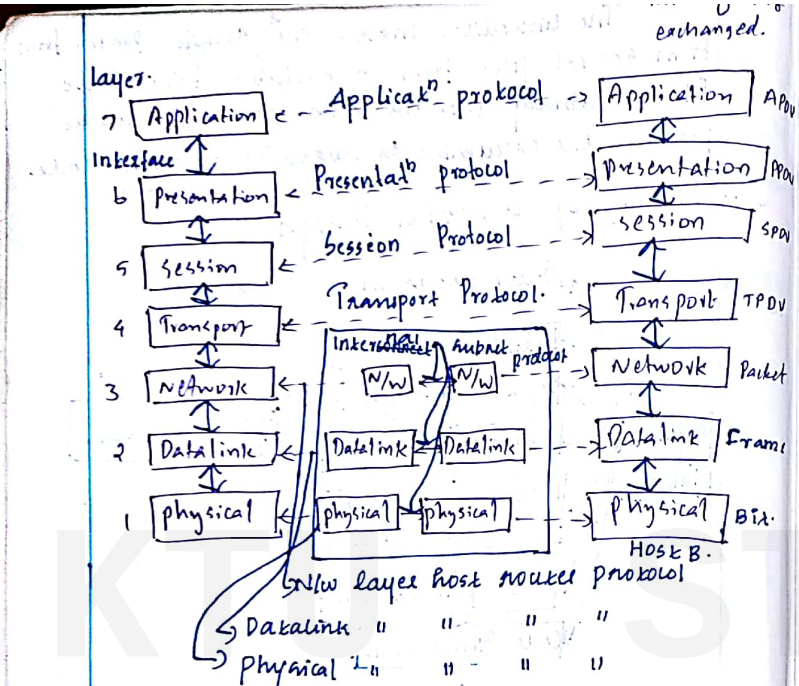


fig: The OSI reference model.

PDU- Protocol Data Unit (Session)

A compiler operates in phases. A Phase is a logically interrelated operation, that takes source program in one representation and produces output in another representation the phases of compiler are shown in above

There are two major phases of compilation.

1. Analysis phase: [Machine independent but language dependent]
2. Synthesis Phase: [Machine dependent and language independent]

→ Analysis phase consist of lexical analysis syntax analysis, Semantic analysis, intermediate code generation.

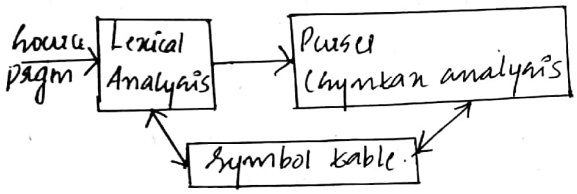
→ Synthesis phase consist of code optimization code generation.

Phase-1: Lexical Analysis

lexical analyser needs the stream of character making up the source program and groups the character into meaningful sequences of lexicans

For each lexeme lexical analyser produces a token of the form [token name, attribute value]

Tokens are pass to subsequence phase for syntax analysis.



Example:

Sample statement newval = oldval + 12

TOKENS:	lexeme	token
	newval	id 1
	=	Assignment operator
	oldval	id 2
	+	Add operator
	12	Number

lexican analyser truncate white space also removes error.

Phase-2: Syntax Analysis

Syntax analysis is also called parsing the parser uses token produced by the lexican analyser to create a tree in intermediate representation that depicts the grametical structure of the token string.

A typical represent is the syntax stream in which each internal node represent operation and children of the node represent argument of the operation.

eg: syntax correspond to

newval = oldval + 12

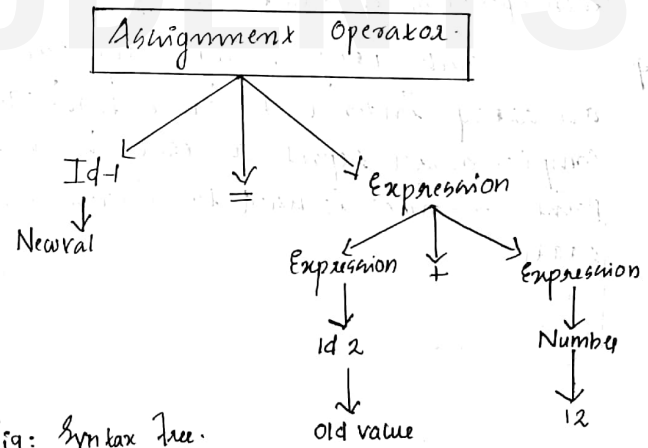


fig: Syntax Tree.

Phase-3 Semantic Analysis

The semantic analyser uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definitions.

It gathers information from either in the syntax string or symbol table and saves it in symbol table for subsequent use during intermediate code generation. An important part the semantic analysis is type checking. Where the compiler check that each operation an matching operations.

eg: Each programing language requir an array index to be an integer. The compiler must report an error if an floating point number is used to index an array.

Phase-4: Intermediate Code Generation.

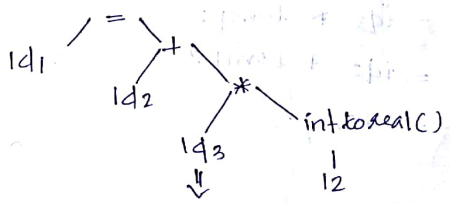
After the semantic analysis of source program many compilers an explicit low level on machine layer intermediate representation. This intermediate representation should have two important properties

1. It should be easy to produce
2. It should be easy to translate into the target machine.

The considered intermediate code for three address code, which consist of three operands per instruction each operand can act like a register. This phase bridges analysis & synthesis phase of translator.

eg: Newval = oldval + fact *12

id 1 id 2 + id3 * 12



temp-1 = int to real (12)
temp-2 = id3 * temp1
temp-3 = id2 + temp2
id1 = temp3

intermediate code.

Three address code
operand = 3

Phase-5: Code Optimization

The compiler looks at the large segment of the formula to decide how to improve performance. The machine independant on optimization attempt to increase the intermediate code so that better target code will present.

Usually better means: faster, shorter, target code that consumes less power

eg: The above intermediate code will optimize for

temp = id 3 * 12.0
id1 = id 2 + temp 2.

temp 1 = int to real (12
temp 2 = id3 * temp2
temp 3 = id2 + temp2
id1 = temp3

Phase 6: Code Generation

The last phase of translation is code generation it takes intermediate representation of source program and maps it into the target language.

eg:
id 1 = id 2 + id 3 * 12

```
mov  R1, id3
mul  R2, #12
mov  R2, id 2
ADD  R1, R2
mov  id2, R1
```

25/1/17

Example:

position = initial + rate *60

↓

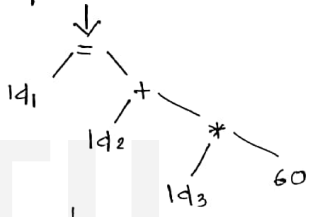
Lexical analyzer

↓

$id_1 = id_2 + id_3 * 60$

↓

Syntax analyser

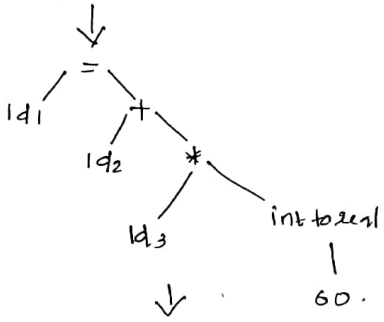


Symbol table

1	Position	id_1
2	Initial	id_2
3	rate	id_3

↓

Semantic Analyzer



↓

Intermediate code Generator

temp1 = inttoreal (60)

temp2 = id_3 * temp1

temp3 = id_2 + temp2

id_1 = temp3.

↓

Code ~~Generator~~ optimizer

↓

temp1 = id_3 * 60.

id_1 = id_2 + temp1

↓

Code Generator.

```
Mov  id3, R2
Mul  60 , R2
Mov  id2 , R1
Add  R2 , R1
Mov  R1 , id1
```

Symbol Table Management:

Symbol table is a datastructure containing a record for each variable name with fields for attributes of the variable name.

The datastructure should be designed to allow the compiler to find the record for each variable name quickly or to store / retrieve data from it quickly. These attributes may provide information about storage allocated for a name its datatype, its scope (where in the pgm is value may be used and in the case of procedure names the things such as no of arguments its type, the method of passing each argument

Position	id_1, ...
Initial	id_2, ...
Rate	id_3, ...
⋮	⋮

Error handling routines:

One of the most imp function of a compiler is the detection & reporting of errors in the source pgm. The error msg should allow the pgmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler. Whenever a phase of the compiler discovers an error, it must report the error to the error handler. Which issues an appropriate diagnostic messag Both of the symbol table mangment & error handler routines interact with all phases of the compiler

Lexical Errors:

eg: 5A (invalid variable name)
First character should be alphabet followed by alphabet/digit

Syntax Error:

missing of braces, ;

Semantic Error:

Invalid array declarath a[10.5]
Division by zero.

```
Char a;
int b,c
c = a+b
```

One pa

It passes through the source code of each compilation phase only once. Thus Efficiency

is limited bcz they don't produce inter mediate. OPC is very common bcz of it' is simplicity they are faster than multi pass compiler. Also known as narrow compiler

eg Pascal ML compiler.

→ Multipass compiler:

The i/p is passed through certain phases in one pass. Then the o/p of previous phases is passed through another phases in second pass until the desired o/p is generated It requires less m/y bcz each pass takes o/p of previous phase as i/p. It may create one/more intermediate code. Also known as wide compiler

eg Modulator 2.

Front End & back End of of a compiler.

The phases of compiler are collected into front end and back end (analysis phase)

The front end consist of those phases that depend primarily owned the source Program. These normally include lexical analysis, syntax analysis, semantic intermediate code generator.

- A certain amount of code optimisation can be done by front end aswell

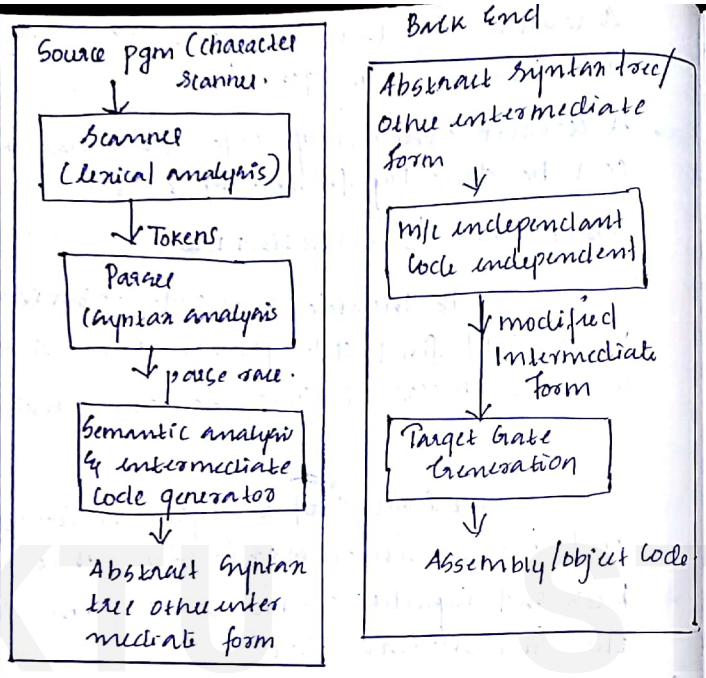
Back End / Synthesis phase

It include the code optimisath phase and final code generath phase along with the necessary error handling and symb

The front end analysis the source pgm and produces intermediate code while the back end synthesises the target pgm from the intermediate code.

%%

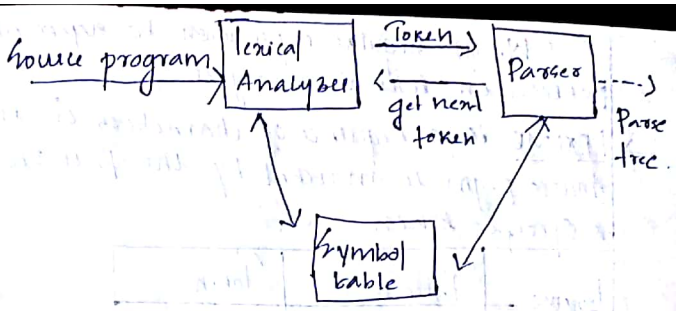
The front end phase consist of that primarily depend on source pgm & are independent of target machine. Vacant phase of a compiler consist of to those which depend on targe and are independent of source pgm.



3/11/17

Lexical Analyser : is the first phase of compiler. Its main task to read the i/p character (source program) and produce as output a sequence of tokens that the parser uses for syntax analysis. This interaction is summerised in below figure

function of Lexical analyser



lexical analyser is also called scanner

Seconclary functh performed by lexical analyser are as follows

1) Remove commands
2) Remove white spaces in the form of blank,
3) Tab and new line character.
3) Report lexical errors.

TOKENS, PATTERNS & LEXEME

→ TOKENS are basic building blocks of a source Program which are indivisible. Tokens can be classified into keywords, operators, identifiers, seperators, strings, numerical constants.

→ PATTERN is a set of rule to recognise tokens present in source program.

We use regular expression to represent pattern in lexical analyser.

→ LEXEME is a sequence of characters in the source prgm is matched by the pattern for a specific token

lexeme	Pattern	Token
785	$[0-a]^+$	number
Sum	$[a-z]$	id 1
a10	$[A-Z-a-3]$ $0-a$	id 2

Specification Of Token.

To specify a token in a lexical analyser by using pattern. Regular expression are an important notation for ~~pattern~~. Specifying pattern.

Definition of Regular Exp

Regular expression over alphabet Σ is defined as follows.

1. ϵ, a, b are primitive regular expression denotes the language $\{\epsilon\}$, $\{a\}$, $\{b\}$

Suppose 'R' and 'S' are regular expression denoting the language L(R) and L(S). Then

a) R.S is a regular expression
b) R/S is a regular expression
c) R^* " " " "
d) (R) " " " "

eg: Specification of Token, Identifier Using regular Expression is as following

id [Az a-z][A-Za-z 0-9]*

Recognise ~~digits no~~ decimal no:s using regular Expression.

Q Develop pattern to recognise fractional number 10.21

digit → [0-9]
num → [digit]
fnum → [digit]+ ("." [digit]+) ?

? => zero/~~more~~ one time.

Definition of finite automata
Review of finite automata
NFA & DFA
Context free grammar.
Derivation tree or Parse tree
left most derivation & ~~Right~~ Most derivation.

Ambigous grammar (TOC note)

Q Pattern specification for keywords

keyword → "int" / "float" / "while" / "if" / "else" / "for"

Q Pattern specification to identify comment lines.

cmt → "/*" . * "*/"

[here, . is a regular generator matches with all characters except newline]

cmt → "//" . * single line

Q Pattern to identify arithmetic operators.

optr → '+' / '-' / '*' / '/'