→ Unit 1

Q. Illustrate the value / various phases of compiler in brief, for the given expression.

a = b + c * d / e

Ans

a = b + c * d / e

↓

[lexical analysis]

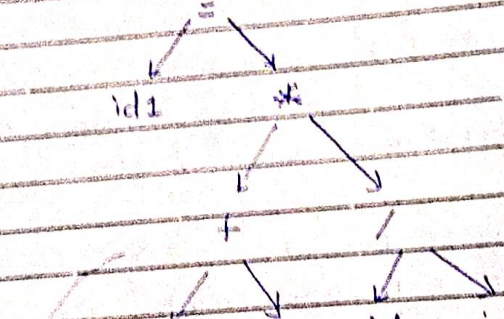
↓

id1 = id2 + id3 * id4 / id5

↓

[Syntax Analysis]

↓



↓

[Semantic analysis]

↓

Same (As their is no type conversion)

↓

[intermediate code generator]

↓

$t_1 = id_2 + id_3$

$t_2 = id_4 / id_5$

$t_3 = t_1 * t_2$

$id_1 = t_3$

↓

[Code optimization]

↓

$t_1 = id_2 + id_3$

$t_2 = id_4 / id_5$

$id_1 = t_1 * t_2$

↓

[Code generation]

↓

```
LOAD  id2, R1
LOAD  id3, R2
ADD   R1, R2
LOAD  id4, R3
LOAD  id5, R4
DIV   R3, R4
MUL   R1, R3
LOAD  R1, id1
```

Q Phases of Compiler.

Ans A compiler is a software program that converts the high level source code written in a programming language into low-level machine code that can be executed by the computer hardware. The process of converting the source code into m/c code involves several phases or stages, which are collectively known as phases of compiler.

The phases of compiler are :-

① Lexical Analysis :-
- First phase
- Also known as scanning phase
- Reads source code and breaks it into a stream of tokens → (basic units of programming language)

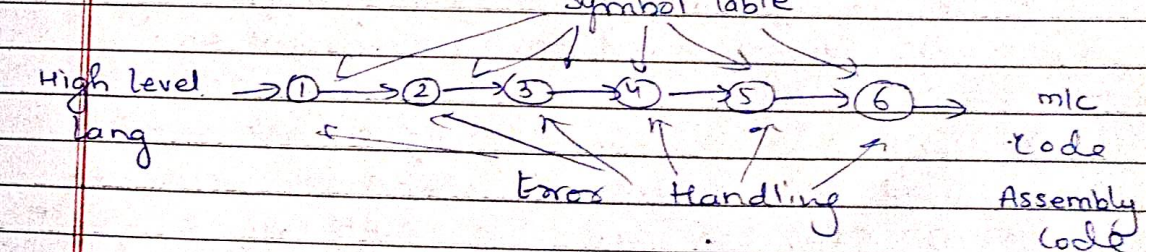
② Syntax Analysis :-
- Second phase
- also known as parsing
- takes stream of tokens from the lexical phase and checks whether they conform to the grammar of program L.
- Output of this phase is usually as AST

③ Semantic Analysis :-
- Third phase
- checks whether the code is semantically correct i.e whether it conforms to the language's type system & other semantic rules
- checks the meaning of the code to ensure it makes sence.
- type checking, semantic errors like undeclared variables & incorrect function call

④ Intermediate Code Generation :-
- Forth phase
- Generates intermediate representation of the source code that can be easily translated into m/c code.

⑤ Code Optimization
- Fifth
- Applies various optimization techniques to the intermediate code to improve performance of the generated m/c code.

⑥ Code Generation
- Final phase
- Takes optimized intermediate code and generates actual m/c code that can be executed by target hardware.



DATE: / /

Compiler & Interpreter Difference

	Compiler	Interpreter
①	The compiler is a software which converts high-lvl lang" into m/c level lang" so that it can be executed by target hardware	① Is a program that directly executes the instructions in a high-level lang" without converting into m/c code
②	Saves the m/c L in the form of m/c code on disks	② Does not save the m/c L
③	Compiled codes run faster than interpreter	③ Interpreted code run slower than compiler
④	The compiler generates on o/p in form of (.exe)	④ does not generate any output
⑤	Errors are displayed in compiler after compiling together at the current time.	⑤ Error is displayed in every line.
⑥	It does not require source code for later execution	⑥ It does require source code for later execution
⑦	CPU utilization is more in case of a compiler	⑦ CPU utilization is less in case of interpreter.
⑧	The use of compilers mostly happens in production enviornment.	⑧ in programming & Development env.
⑨	object code is permanently saved for future use	⑨ No object code is saved for future use
⑩	C, C++, C# are lang's that are C-based	⑩ MATLAB, Ruby, Perl, etc are p.l that are I-based.

DATE: / /

Criteria	Token	lexeme	Pattern
Def.	Token is basically a sequence of char" that are treated as a unit as it cannot be further broken down	It is a sequence of char" in source code that are matched by given predefined lang rules for every lexeme to be specified as a valid token	It specifies a scanner follows to create a token
Interpretation of type Keyword	all the reserved keywords of that lang (printf, main, etc)	int, goto	Sequence of char that make keyword
Interpretation of type Identifier	name of var, fun", etc	main, a	it must start with the alphabet followed by the alpha or digit
I.o.t operator	all the operators are considered as tokens	+, =	+, =
I.o.t punc"	each kind of punc" is considered as tokens (semicolon, bracket, comma etc)	(,), {, }	(), {, }
I.o.t literal	a grammar rule or boolean literal	"welcome to exam!"	any string of characters accept " x "

Q. How is finite automata useful for lexical analysis? And how many tokens are generated in the following statement:

```
Printf("%d%f%f%f", rollno, per_marks);
```

Ans In CD, lexical analysis (scanning) is the first phase that breaks down the source code into meaningful units called tokens. These tokens like keywords (printf), identifiers (rollno) operators (+,-) constraints (numbers) or punctuation symbol.

Finite automata are powerfull tools for implementing lexical analysis because they can efficiently recognize patterns in the input stream (source code)

A FA is a mathematical model consisting of states, transitions between states based on input symbol & a set of accepting states that represent valid tokens

Here's how FA helps in lexical analysis

① Regular Expressions
→ Lexical rules are often defined using R.E, describes legal token

② FA from R.E
→ A FA can be constructed using R.E. This FA is used to scan input stream & recognized tokens that match the R.E

③ Efficient Recognition:
→ FA-based scanners are efficient because they can quickly determine if the current input sequence can lead to a valid token.

⇒ Statement
:- printf("%d%f%f%f", rollno, per_marks);

① printf : keyword
② (: left parenthesis (open circular bracket)
③ " : left open

① printf :- keyword
② (:- open circular bracket
③ " :- Quotes
④ %d%f%f%f :- Treated as single unit
⑤ " :- Quotes
⑥ , :- Comma
⑦ rollno :- identifier
⑧ , :- comma

PAGE NO.:
DATE: / /

printf ("%d %f %c", rollno, per_mark);

① printf :- Identifier
② (:- Left parenthesis
③ "%d %f %c" :- String literal
④ , :- Comma
⑤ rollno :- Identifier
⑥ , :- comma
⑦ permark :- Identifier
⑧) :- Right parenthesis
⑨ ; :- semicolon.

Total 9 tokens.