6th batch final

Answer to the question 1(a)

Analyze Static Vs Dynamic checking.

| Static | Dynamic |
| --- | --- |
| I. It happens at compile time | At run time |
| II. To detect errors before program run | II. To detect while program running |
| III. Faster checking before execution | III. Slower happens during execution |
| IV. Useful in early error detection | IV. Needed for checking user inputs |
| V. Tools used: compiler, static analyzer | V. Divided by zero, null pointer, |
| VI. Limitation: can't catch all runtime issue before | VI. can't catch error until they happen |

(b)

## Answer to the question 2(a)

Define type equivalence : Explain the role of Intermediate code generator in compilation process

Type equivalence refers to the rules used by a compiler to determine if two dat types are same

There are two main approach

Name equivalence : ~~They~~ Types are equivalent if they have the same name.

Structural equivalence : Types are equivalent if they have the same structure, even if their names differ.

Role of Intermediate code generator

The intermediate code generator translets the high level source code into an intermediate representation that is independent of machine arcebitecture

Simplifying optimization : Makes code easier to analyze and optimize

portability : Intermediate code can often run on different types of machines without needing major change

Translation: The compiler ~~task~~ takes the high level code and converts it into an intermediate form, which can be easier to analyze and manipulate

(b). construct a quadruple, triples for the following expression

$$a + a * (b-c) + (b-c) * d ?$$

Here,

$t_1 = b - c$

$t_2 = a \times t_1$

$t_3 = t_1 \times d$

$t_4 = a + t_2$

$t_5 = t_4 + t_3$

Quadraple:

| Location | operator | $arg_1$ | $arg_2$ | result |
|---|---|---|---|---|
| 0 ~~1~~ | − | b | c | $t_1$ |
| 1 ~~2~~ ~~2~~ | * | a | $t_1$ | $t_2$ |
| 2 ~~3~~ | * | $t_1$ | d | $t_3$ |
| 3 ~~4~~ | + | a | $t_2$ | $t_4$ |
| 4 ~~5~~ | + | $t_4$ | $t_3$ | $t_5$ |

Triple:

| Location | operator | Arg1 | Arg2 |
|---|---|---|---|
| ~~1~~ 0 | − | b | c |
| ~~2~~ 1 | * | a | (0) |
| 2 | * | (0) | d |
| 3 | + | a | 1 |
| 4 | + | (3) | (2) |

(c) Define peephole optimization. Explain machine dependent and machine independent optimization.

Peephole optimization: peephole optimization is a technique used in compiler design to improve code efficiency by (analyzing a small part of the code set of instruction) by making it consume less resources and deliver high speed

| Machine dependent | Machine Independent |
|---|---|
| Optimization tailored to specific hardware architecture | optimization applied to intermediate code, independent of hardware |
| Relies on hardware features | Relies on code structure |
| Target hardware | Intermediate code |
| maximize performance on specific hardware | Improve code efficiency |
| Optimization level: Machine code | Before code generation |
| Examples: Register allocation | dead code Removal |

## Answer to the question 3. (a)

Define tokens, patterns, and lexemes with examples. What is the function of the lexical analyzer? Demonstrate the interactions between the lexical analyzer and the parser:

**Tokens:** A token is a sequence of characters that represents a single source logical code unit in the source code. token is the smallest unit of meaningful data

Examples:

keyword int, char, float

Identifier Variable name

operator +, -, *, ÷

int sum = a+b

**patterns:** A pattern is a rule or syntax that designates (নির্দেশ করা) How tokens are identified in a programing Language.

Examples: For, identifier → valid token

the pattern is the predefined rules that it must start with alphabet, followed by alphabet or a digit.

sum →

Lexeme: A lexeme is sequence of source code that matches one of the predefined patterns and thereby forms a valid token.

Example

$x+5$ :

$x$ and 5 are lexeme

Interactions Between Lexical Analyzer and parser

Token passing: The lexer scans the source code and produces a stream of tokens, which it passes to the parser

Request driven: The parser requests tokens from the lexer as needed to build the parse tree according to the language grammar.

Feedback: parser detects ~~error~~ syntax error, It may rely on the lexer to provide context

Buffering: The lexer may buffer tokens to ensure efficient delivery to the parser

(b) What operations on Language? Let L be the set of letters {A, B, ...., Z, a, b .... z} and Let D be the set of digits {0, 1, 9...}. We may think L and D in two, essentialy equivalent, ways. One way is that L and D are, respectivley, the alphabets of uppercase and lowercase letters and of digits. Show some other Language that can be constructed from Language L and D, using the union, concatenation, kleene closure and positive ~~clos~~ closure operators.

A Language is a set of string over an alphabet, where an alphabet is a finite set of symbols.

Operation of Language

1. Union
2. concatenation
3. Kleene closure
4. positive closure

Given alphabets : $L = \{A, B \ldots Z, a, b \ldots z\}$

$D = \{0, 1, \ldots 9\}$

construction Language:

Union: $L \cup D$

$= \{A, B, \ldots Z\} \cup \{0, 1, \ldots 9\}$

$= \{A, B, C, D \ldots 0, 1, 2 \ldots\}$

Concatenation: $L \cdot D$

$= \{A, B, C \ldots Z\} \cdot \{0, 1, \ldots 9\}$

$= \{A1, Z0, B9\}$

Kleene closure: $D^*$ [consist (0) or many repetiti

$= \{0, \epsilon, 01, 007, 0\overset{12}{\cancel{AB}}, \cancel{01, B}\}$ no use (0)

positive closure: $L^+$

$= \{A, ab, xyz\}$ [All strings made by 1 or more repetati of elements from L

(e) Briefly explain the rules that define regular expression over some alphabet and the language that those expressions denote using basis and induction. How do you recognize the reserved words and identifiers? Show the transition diagram of retop.

Answer to the question 4.(a)

What is a dangling else?

The dangling else problem aries when an else statement is ambigious in a nestead of if structure, making it unclear which if statement it belongs to. This happens because most programing languages resolves this ambiguity by associating the else with nearest preciding if, ~~even if the programmer intends it for a different one~~.

This happens when braces { } are not used to clearly define code blocks

Example:

```
if (condition1) {
        if (condition2) {
                // some code
        }
    else {  // which if does this
            belong to ?
            // some code
    }
}
```

In this case, the else could be intended to match the inner if (condition2), or the outer if (condition1). The compiler resolves this ambiguity by associating the else with closest unmatched if.

(b)

How can the following grammar be ambigious for (id-id/id)?

$$E \rightarrow E-E \mid E/E \mid -E \mid (E) \mid id$$

Input: id-id/id

production rule

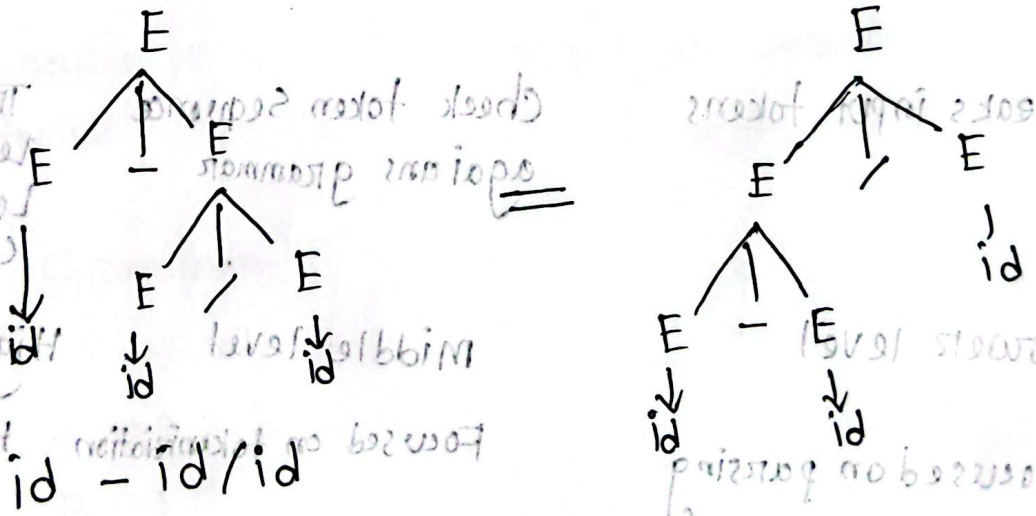
$E \rightarrow E-E$

$E \rightarrow E/E$

$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow id$




id - id/id

So this is ambigious

(Answer)

## Answer to the question 5. (a)

Differentiate the terms (i) yacc compiler (ii) Lex compiler, and (iii) c compiler

| | | |
|---|---|---|
| yacc compiler | Lex compiler | c compiler |
| yet another compiler-compiler | Lexical Analyzer Generator | c Language compiler |
| creats parser | creats analyzer | converts c code to machine code |
| Takes a context free grammar and produces a c program that performs syntax analysis | Takes regular expression and produces a c program that tokenizes input by identifying meaningful units | perform all compilation phases syntax analysis semantic analysis code optimization |
| Breaks input tokens | Check token sequence agai ans grammar | Translates high-level c cod into Low-level machine code |
| Lower level | middle level | High level |
| Focused on parsing | Focused on tokenization | Handles complex task |
| Used for syntax analysis in compiler design | Lexical analysis in compiler design | Final compilation to run on a machine |
| Input: Grammar rules (context free grammar) | token patterns (Regular expression) | C Source code |
| output: c code | c code | object code |

(b)

Discuss about handles

Handles is a part of an object. Handles are Small squares or points that allow users to interact with and manipulate object in a graphical interface. They are used to change the shape, size, position or other properties of graphical element.

Types of handles

1. Bounding Box handles
2. control points
3. Rotation handles

A handle is a substring of grammar symbols in a right-sentential from that matches a right-hand side of a production

Grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$a\underline{b}bcde$

$a\underline{Abc}de$

$a\underline{A}d\underline{e}$

$\underline{aABe}$

$S$

Handle

(e)

be following is a substring of grammar symbols

abbcde

Grammar

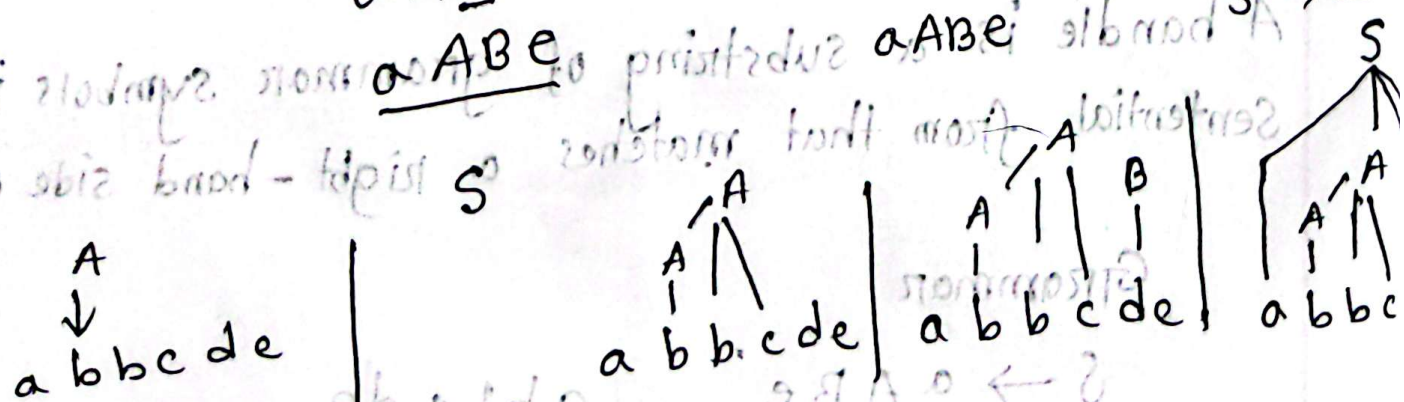
$S \to aABe$

$A \to Abc \mid b$

$B \to d$

prove that this substring show the property of "Handle".

| Right Sentential form | Handle | Replacing prod |
|---|---|---|
| a<u>b</u>bcde | b | $A \to b$ |
| a<u>Abc</u>de | Abc | $A \to Abc$ |
| aA<u>d</u>e | d | $B \to d$ |
| <u>aABe</u> | aABe | $S \to aA$ |
| S | | |



Answer to the question 6.(a)

Describe algorithm for LR parser

LR parsing is a bottom-up parsing method used in compiler design to analyze the syntax of programing Language. It works by building a parse tree from the bottom up, starting wit the input string and reducing it to the start symbol. LR stands for Left-to-right, rightmost derivation

Bottom-up: It starts from the input string and works start symbol, building a parse tree in reverse

Left-to-right: It reads the input string from left to right

Rightmost derivation in reverse: It constructs the parse tree by working backwards from rightmost derivation of the input string.

## LR parsing work:

**Stack and Input:** The parser maintains stack to store symbols

**Shift:** current input symbol is moved onto the stack

**Reduce:** If the top of the stack matches the right-hand side of a production rule, it is reduced to the left-hand side symbol, and the corresponding action is taken

**Accept:** Matching right side string when match the string then accept it.

## LR praser types:

- LR(0): The simpliest form of LR parsing, using no Lookhead.
- SLR(1): using one Lookhead symbol. Simple LR parser
- LALR (1): Look-ahead LR parser
- CLR: Canonical LR parser, using one Lookhead symbol

Example:

| Grammar | Right sentential | Handle | Replacing prod |
|---|---|---|---|
| $S \to aABe$ | abbcde | b | $A \to b$ |
| $A \to Abc \mid b$ | aAbcde | Abc | $A \to Ab$ |
| $B \to d$ | aAde | d | $B \to d$ |
| | aABe | aABe | $S \to aABe$ |

(b)

Based on LR parsing, for the following grammar, show all steps for the stack, input, and action.

| Grammar | Augment grammar |
|---|---|
| 1. $E \to E+T$ | $E' \to E$ |
| 2. $E \to T$ | $E \to E+T$ |
| 3. $T \to T*F$ | $E \to T$ |
| 4. $T \to F$ | $T \to F$ |
| 5. $F \to (E)$ | $F \to (E)$ |
| 6. $F \to id$ | $F \to id$ |

| | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| state | id | + | * | ( | ) | $ | E | T | F |
| | | | | | | | | | |

closure:

$I_0$

$E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

$\xrightarrow{E}$ $I_1$: $E' \to E.$

$\xrightarrow{E}$ $I_2$: $E \to E.+T$

$\xrightarrow{T}$ $I_3$: $E \to T.$

$\xrightarrow{F}$ $I_5$: $F \to id.$

From $I_0$, $I_1 \xrightarrow{E}$

$E' \to E.$

$E \to E.+T$ $\longrightarrow$ From $I_1$ $\longrightarrow$ $I_6$ $E \xrightarrow{+} E+.T$

$T \to .T*F$

$T \to .F$ $\to I_3$

$F \to .(E)$

$F \to .id$

$I_2$ $\xrightarrow{T}$ $T.$

$T \to T.*F$ $\longrightarrow$ From $I_2$

$I_3$ $T \xrightarrow{F} F.$

$I_7$ $T \xrightarrow{*} T*.F$

$F \to .(E)$

$F \to .id$

$I_{10}$ $\longrightarrow$ $T*F.$

$I_4$,

$F \xrightarrow{E} (.E)$

$E \to .E+T$ $\longrightarrow$ $I_8$ $F \to (E.)$ $\xrightarrow{E}$ $I_{11}$ $(E.)$

$E \to .T$ $\quad E \xrightarrow{+} E.+T$

$F \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

$F.$

~~Sta~~

$I_0$

$E' \to .E$

$E \to .E+T$

$E \to .T$

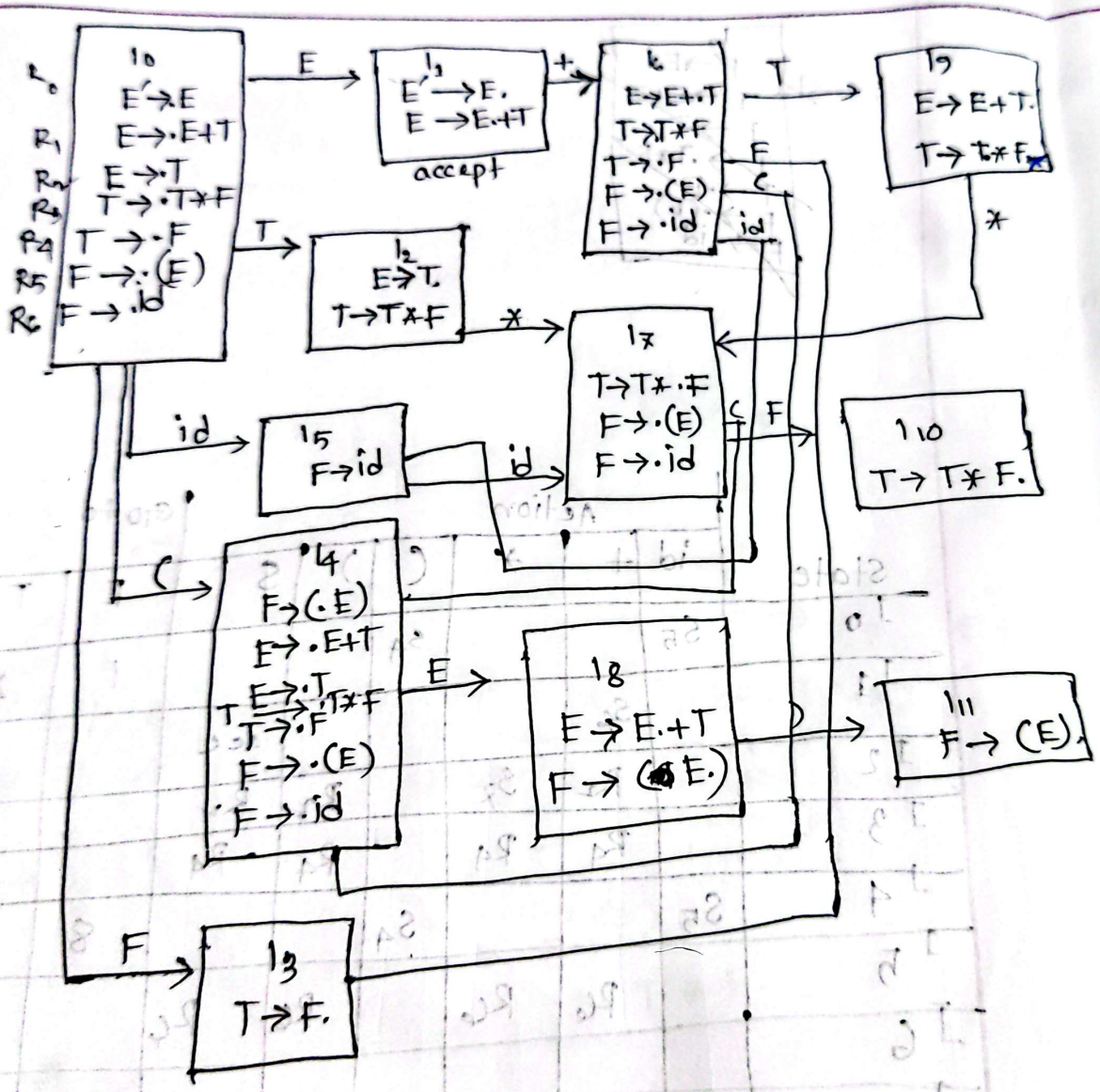
$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

| State | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| $I_0$ | $S_5$ | | | $S_4$ | | | 1 | 2 | 3 |
| $I_1$ | | $S_6$ | | | | acc | | | |
| $I_2$ | | $R_2$ | $S_7$ | | $R_2$ | $R_2$ | | | |
| $I_3$ | | $R_4$ | $R_4$ | | $R_4$ | $R_4$ | | | |
| $I_4$ | $S_5$ | | | $S_4$ | | | 8 | 2 | 3 |
| $I_5$ | | $R_6$ | $R_6$ | | $R_6$ | $R_6$ | | | |
| $I_6$ | | | | | | | | 9 | 3 |
| $I_7$ | $S_5$ | | | $S_4$ | | | | | 10 |
| $I_8$ | | $S_6$ | | | $S_{11}$ | | | | |
| $I_9$ | | $R_1$ | $S_7$ | | $R_1$ | $R_1$ | | | |
| $I_{10}$ | | $R_3$ | $R_3$ | | $R_3$ | $R_3$ | | | |
| $I_{11}$ | | $R_5$ | $R_5$ | | $R_5$ | $R_5$ | | | |




Ans

Distinguis
In LR pa

(1) pa
(II) ph

pa

1. A bass
techniqu
tokens
token is

II. Disco
synch

III- Sim
ou

IV.

V.

VI

Answer to the Question 7(a)

Distinguish the following terms for error recovery in LR parsing

(I) panic mode
(II) phrase-level recovery

| panic mode | phrase-level recovery |
|---|---|
| I. A bassic error recovery technique that skips input tokens untill a synchronization token is found | I. A method that corrects errors by modifying the input. |
| II. Discards input tokens untill a synchronization token is encountered | II. Analyzes the error and applies local fixes (inserting, deleting, substituting) |
| III. Simple to implement, but may skip over multiple errors | III. more sobisticated, can recover from more types of errors |
| IV. Relatively simple to implement | IV. more complex to design and implement. |
| V. Less sphisticated | V. more sophisticated |
| VI. Error Recovery by ignoring many tokens | VI. Error recovery by applying small, localized fixes. |
| VII. May lose a large amount of context | VII. Local context is largely preserved. |

convert grammar, left recursion

$A \to A\alpha \mid \beta$ immedi

$A \to \beta A'$

$A' \to \epsilon \mid \alpha A'$

$L \to L, S / S$

$A \quad A \quad \alpha \quad \beta$

(b)

construct the predictive parser for the following grammar

$S \to (L) / a$ $\xrightarrow{\text{convert}}$ $S \to (L) / a$

$L \to L, S / S$ $\quad L \to SL'$

$L' \to .SL' \mid \epsilon$

| | | first | follow |
|---|---|---|---|
| $S \to (L) / a$ | S | {(, a} | {$, .) } |
| $L \to L, S / S$ | L | {(, a} | {) } |
| | L' | {., ε} | {) } |

Parsing table:

| | a | ( | ) | . | $ |
|---|---|---|---|---|---|
| S | S→a | S→(L) | | | |
| L | L→SL' | L→SL' | | | |
| S' | | | L'→ε | L'→.SL' | |

(c) construct the LR parsing table for the following grammar

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

solve before

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Answer to the question 8.(a)

Let, you would like do parsing for syntax analyzer. write down a table to show follow (A)

$E \to TE_R$

$E_R \to +TE_R \mid \varepsilon$

$T \to FT_R$

$T_R \to *FT_R \mid \varepsilon$

$F \to (E) \mid id$

| | first | follow |
|---|---|---|
| $E$ | $\{(, id\}$ | $\{\$, ), id\}$ |
| $E_R$ | $\{+, \varepsilon\}$ | $\{\$, ), id\}$ |
| $T$ | $\{(, id\}$ | $\{+, \$, ), id\}$ |
| $T_R$ | $\{*, \varepsilon\}$ | $\{+, \$, ), id\}$ |
| $F$ | $\{(, id\}$ | $\{*, +, \$, ), id\}$ |

Ans

(b) give specific examples of static checking for the

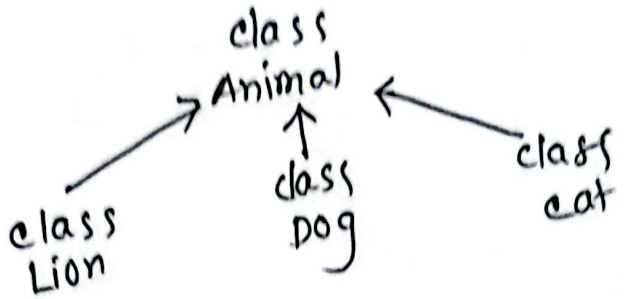
(a) Overloading: overloading refers to using the same name for multiple functions or methods in a program, but each with a different set of parameters. static checking ensures that these overload functions have distinct parameter types or numbers of parameters.

Example: print(int x) for numbers

correction: correction involves automatically converting one type of another. Static checking ensures that when different types are mixed, there's a clear and consistent way to convert one type into another

Example: If we try to add an integer and a floating-point number, the compiler checks that there's a well-defined way to convert the integer into a floating-point number before performing the addition.

**polymorphism:** polymorphism means 'many forms' and it occurs when we have many classes that are related to each other by inheritance.




**Flow-of-control checks:** The compiler checks to make sure that if a statement in my code causes the program to leave a particular part (like function, loop, condition), there must be a proper way for the program to continue its execution.