SYNERGY INSTITUTE OF ENGINEERING & TECHNOLOGY
Department of Computer Science & Engineering
Academic Session 2023-24

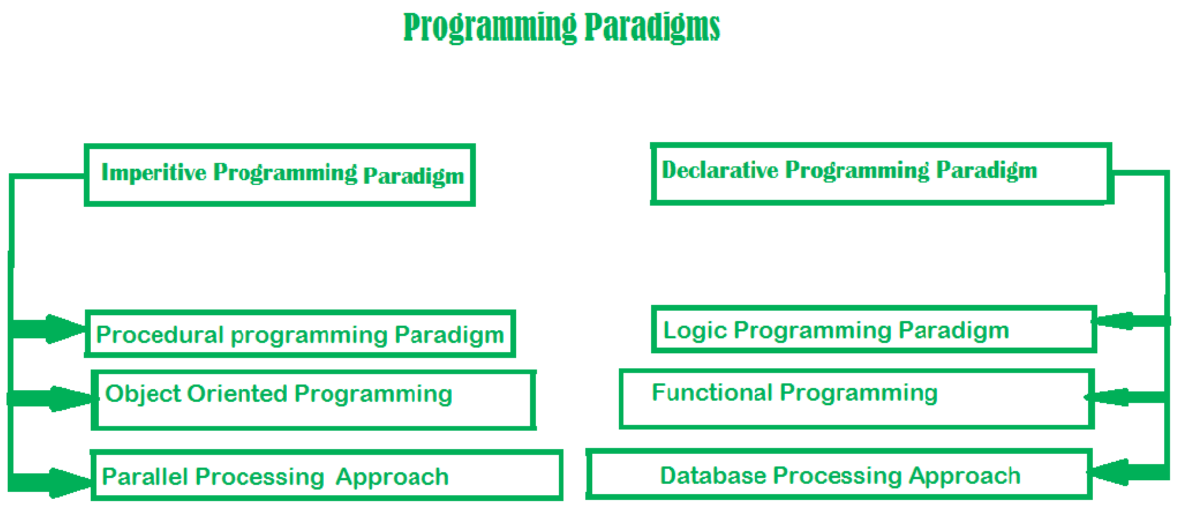
LECTURE NOTE

Nameof Faculty	: Mr. Ashis Kumar Acharya
Name of Subject	: Principle Of Programming Language
Subject code	: RCS4D003
Subject Credit	: 3
Semester	: 4th
Course	: B.Tech
Branch	: Computer Science And Engineering
Admission Batch	: 2022-26

Module-1

A) Introduction:-(overview of different programming paradiagram)

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand. They are discussed below:


Programming Paradigms
Imperitive Programming Paradigm
Procedural programming Paradigm
Object Oriented Programming
Parallel Processing Approach
Declarative Programming Paradigm
Logic Programming Paradigm
Functional Programming
Database Processing Approach

1. Imperative programming paradigm: It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consist of several statements and after execution of all the result is stored.

Advantages:

1. Very simple to implement
2. It contains loops, variables etc.

Disadvantage:

1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

Object oriented programming –

The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. It can handle almost all kind of real life problems which are today in scenario.

Advantages:

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

Parallel processing approach –

Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.

2. Declarative programming paradigm:

It is divided as Logic, Functional, Database. In computer science the *declarative programming* is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic.It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code is actually doing. It just declares the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.

Logic programming paradigms –

It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism.

In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

Functional programming paradigms –

The functional programming paradigms has its roots in mathematics and it is language independent. The key principle of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions.The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like perl, javascript mostly uses this paradigm.

Database/Data driven programming approach –

This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

B) compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.

  - When you execute a program which is written in HLL programming language then it executes into two parts.
  - In the first part, the source program compiled and translated into the object program (low level language).
  - In the second part, object program translated into the target program through the assembler.

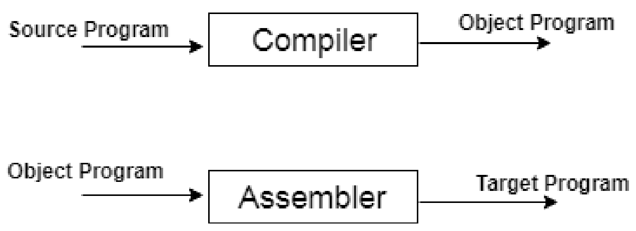


Fig: Execution process of source program in Compiler

C) Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:

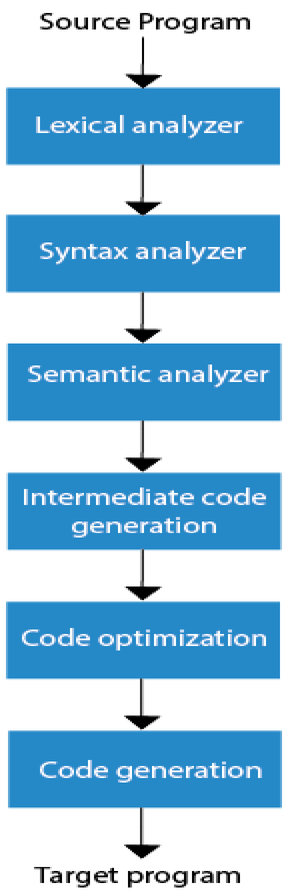


Fig: phases of compiler

- Lexical Analysis: Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.
- Syntax Analysis:- Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.
- Semantic Analysis:- Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

- Intermediate Code Generation:- In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.
- Code Optimization:- Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.
- Code Generation:- Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

D) Finite state machine

- Finite state machine is used to recognize patterns.
- Finite automata machine takes the string of symbol as input and changes its state accordingly. In the input, when a desired symbol is found then the transition occurs.
- While transition, the automata can either move to the next state or stay in the same state.
- FA has two states: accept state or reject state. When the input string is successfully processed and the automata reached its final state then it will accept.

A finite automata consists of following:

Q: finite et of states
∑: finite set of input symbol
q0: initial state
F: final state
δ: Transition function

Transition function can be define as $\delta: Q \times \sum \rightarrow Q$

FA is 2 types 1. DFA (finite automata) 2.NDFA (non deterministic finite automata)

DFA:- DFA stands for Deterministic Finite Automata. Deterministic refers to the uniqueness of the computation. In DFA, the input character goes to one state only. DFA doesn't accept the null move that means the DFA cannot change state without any input character.

DFA has five tuples {Q, ∑, q0, F, δ}

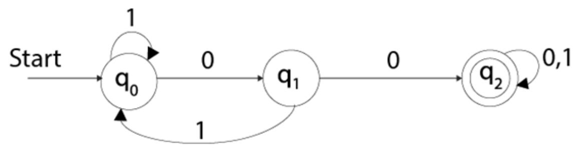
Q: set of all states
∑: finite set of input symbol where $\delta: Q \times \sum \rightarrow Q$
q0: initial state

F: final state
δ: Transition function

Example

See an example of deterministic finite automata:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q3}



NDFA:- NDFA refer to the Non Deterministic Finite Automata. It is used to transit the any number of states for a particular input. NDFA accepts the NULL move that means it can change state without reading the symbols.

NDFA also has five states same as DFA. But NDFA has different transition function.

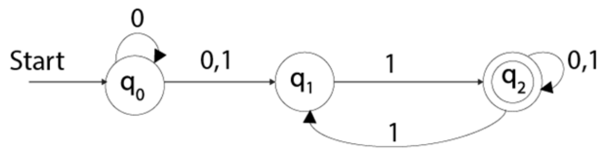
Transition function of NDFA can be defined as:

$\delta: Q \times \sum \rightarrow 2^Q$

Example

See an example of non deterministic finite automata:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q3}



LEX:-

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

E) Formal grammar:-

- Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.
- Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- Formal grammar is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.

Formal grammar G is written as follows:

1. G = <V, N, P, S>

Where:

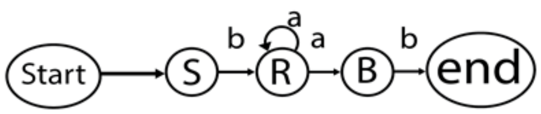
N describes a finite set of non-terminal symbols.
V describes a finite set of terminal symbols.
P describes a set of production rules
S is the start symbol.

Example:

1. L = {a, b}, N = {S, R, B}

Production rules:

1. S = bR
2. R = aR
3. R = aB
4. B = b



This production describes the string of shape ba^{n}ab.

F) Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where, G describes the grammar T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:
L= {wcw^R | w € (a, b)*}
Production rules:
S → aSa
S → bSb
S → c
Now check that abbcbba string can be derived from the given CFG.
S ⇒ aSa
S ⇒ abSba

S ⇒ abbSbba
S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

G) Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation:- In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:
Production rules:
S = S + S
S = S - S
S = a | b |c
Input:
a - b + c
The left-most derivation is:
S = S + S
S = S - S + S
S = a - S + S
S = a - b + S
S = a - b + c

Right-most Derivation:- In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.
Example:
S = S + S
S = S - S
S = a | b |c
Input:
a - b + c
The right-most derivation is:
S = S - S
S = S - S + S
S = S - S + c
S = S - b + c
S = a - b + c

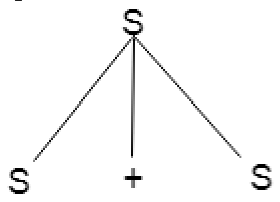
H) Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

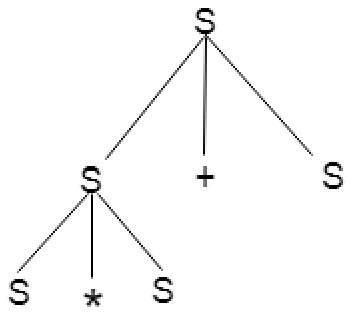
The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

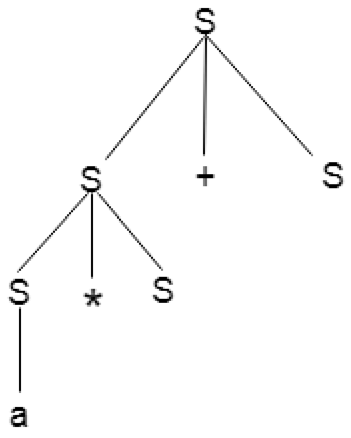
Example:
Production rules:
T= T + T | T * T
T = a|b|c
Input:
a * b + c
Step 1:



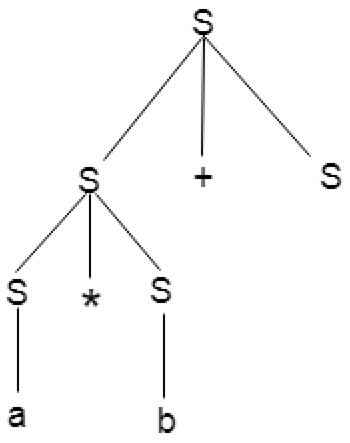
Step 2:



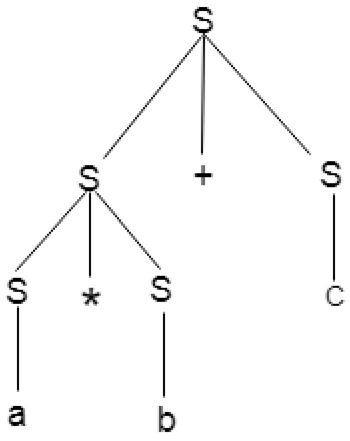
Step 3:



Step 4:



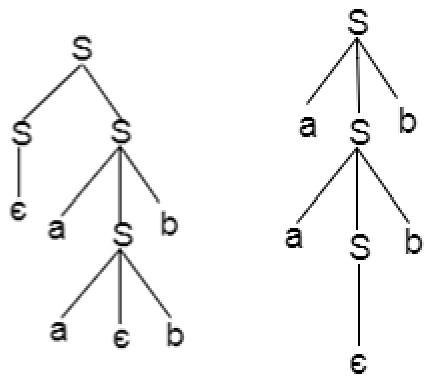
Step 5:



I) Ambigus Grammar:-

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:
S = aSb | SS
S = ∈
For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

J) syntax specification and semiformal semantic specification using attribute grammar.

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

```
E → E + T
```

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

```
CFG + semantic rules = Syntax Directed Definitions
```

For example:

```
int a = “value”;
```

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking
-

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

K)Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

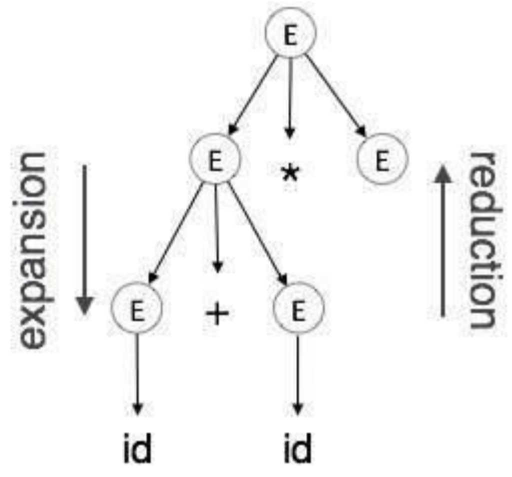
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

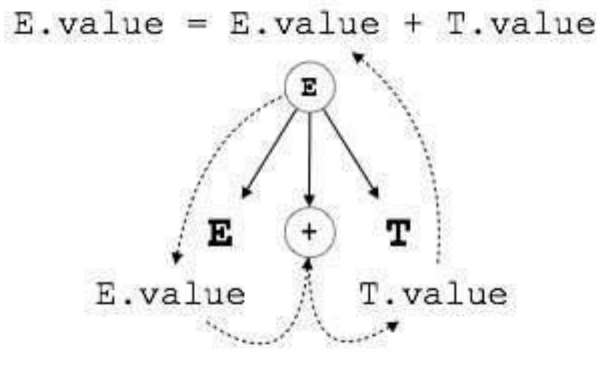
For example:

```
int value  = 5;
<type, “integer”>
<presentvalue, “5”>
```

For every production, we attach a semantic rule.

L)S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

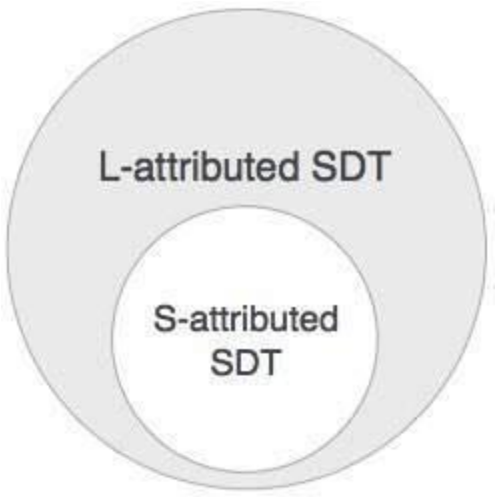
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Module-2

A) Names :- Variables, subprograms, labels, user defined types, formal parameters all have names.
Design issues for names:-

What is the maximum length of a name? –
Are names case sensitive or not? –
Are special words reserved words or keywords?

Length:-
– If too short, they cannot be connotative
– Language examples: • Earliest languages : single character
• FORTRAN 95: maximum of 31 characters
• C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31 characters
• C#, Ada, and Java: no limit, and all are significant
• C++: no limit, but implementers often impose one

Name Forms
• Names in most PL have the same form:
– A letter followed by a string consisting of letters, digits, and underscore characters – In some, they use special characters before a variable's name
• Today "camel" notation is more popular for C-based languages (e.g. myStack)
• In early versions of Fortran
– embedded spaces were ignored. e.g. following two names are equivalent
Sum Of Salaries
SumOfSalaries

Special characters
– PHP: all variable names must begin with dollar signs
– Perl: all variable names begin with special characters ($, @, %), which specify the variable's type
– Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Case sensitivity
– In many languages (e.g. C-based languages) uppercase and lowercase letters in names are distinct
• e.g. rose, ROSE, Rose

– Disadvantage: readability (names that look alike are different)
• Names in the C-based languages are case sensitive
• Names in others are not
• Worse in C++, Java, and C# because predefined names are mixed case (e.g. IndexOutOfBoundsException)
– Also bad for writability since programmer has to remember the correct cases 9 Names (continued)
• Special words
– An aid to readability; used to delimit or separate statement clauses
• A keyword is a word that is special only in certain contexts, e.g., in Fortran
real VarName (Real is a data type followed with a name, therefore Real is a keyword)
Real = 3.4 (Real is a variable)
INTEGER REAL
REAL INTEGER
This is allowed but not readable
Special words
– A reserved word is a special word that cannot be used as a user-defined name
• Can't define for or while as function or variable names.
• Good design choice
• Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)
Special Words
• Predefined names: have predefined meanings, but can be redefined by the user
• Between special words and user-defined names.
• For example, built-in data type names in Pascal, such as INTEGER, normal input/output subprogram names, such as readln, writeln, are predefined.
• In Ada, Integer and Float are predefined, and they can be redefined by any Ada program. 12 13 Variables

B)Scope:
» The region of program text where a binding is active is called its scope
» Notice that scope is different from lifetime Two major scoping disciplines:♣
» Static or lexical: binding of a name is given by its declaration in the innermost enclosing block
• binding of a name is determined by rules that refer only to the program text
• Typically, the scope is the smallest block in which the variable is declared
• Most languages use some variant of this
• Scope can be determined at compile time Most languages use some variant of this
» dynamic: binding of a name is given by the most recent declaration encountered at runtime
• binding of a name is given by the most recent declaration encountered during run-time
• Used in SNOBOL, APL, some versions of LISP

C) Binding time:-
A binding is an association between two things, such as a name and the thing it names .
In general, binding time refers to the notion of resolving any design decision in a language implementation (e.g., an example of a static binding is a function call: the function referenced by the identifier cannot change at runtime).
Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made. There are many times when decision about the binding are taken:

language design time: the control flow constructs, the set of fundamental (primitive) types, the available constructors for creating complex types, and many other aspects of language semantics are chosen when the language is designed language implementation time: precision (number of bits) of the fundamental types, the coupling of I/O to the operating system's notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow

program writing time: programmers choose algorithms and names

compile time: compilers plan for data layout (the mapping of high-level constructs to machine code, including the layout of statically defined data in memory)

link time: layout of whole program in memory (virtual addresses are chosen at link time), the linker chooses the overall layout of the modules with respect to one another, and resolves intermodule references

load time: choice of physical addresses (the processor's memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time)

Run time: is a very broad term that covers the entire span from the beginning to the end of execution:

program start-up time
module entry time
elaboration time (point a which a declaration is first "seen")
procedure entry time
block entry time
statement execution time

The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time.

D)Binding Types:-

- Static Binding
- **Static Binding is also known as Compile Time Binding.**
- Static binding occurs when the connection between a function call and its definition is set when your program is being prepared for running, called Compile Time.
- Think of default settings as how something works, like how a tool behaves when you use it. In programming, the default way of connecting a function call to its definition in C++ is called Static Binding. It's like having a pre-set rule that says, "Hey, link these things together when the program is being made."
- Static binding is all about making this link between a function call and its definition while the program is being prepared, usually before it starts running. This is why it's also called Compile time binding or Early Binding.
- You can think of it as putting labels on things beforehand. You can match things up quickly if you know what the labels on the items say.
- Static binding is used automatically for regular function calls in your program. It is also used when using multiple iterations of a function or when using operators in unique ways.
- When methods are marked as static, private, or final, their linking is always done during compile-time. This is because the class type is known beforehand, so the connection between calls and definitions can be decided immediately.

Some of the advantages of using the Static Binding are as follows:

- Static binding is faster than dynamic binding. The program runs faster because the computer is already aware of all the methods in a class. This also implies that the computer will not require additional time to make sense of things while running the program.
- Programs run more quickly and efficiently because they are faster.

However, static binding has a small downside and is a bit less flexible. Before the program executes, all decisions about the function's call and input values are made. Changing anything while the program is running is difficult.

Examples of Static Binding are:

- Function Overloadin
- Operator Overloading

Dynamic Binding

When the connection between a function call and its definition happens while the program is running, this is called Dynamic Binding.

At compile time or during the preparation of our program, the computer sometimes cannot understand all the information in a function call. Instead, it sorts things out while the program is actually running. This kind of linking is known as dynamic binding.

It is also known as Run time binding or Late binding because everything is decided while the program is running.

Some of the advantages of using dynamic binding are given here.

- The fact that the same function can handle a variety of objects gives it a lot of flexibility.
- It can help make programs smaller and easier to understand.

There exist a drawback to it as well, as all the calculations must be made while the program is running, which can slow down operations. This is one of the less desirable aspects of dynamic binding.

Examples of dynamic binding is:

- Virtual Function

Static Binding VS Dynamic Binding

Static Binding	Dynamic Binding
• It happens at compile time. • This type of binding is also called early binding. • It takes place when all the information needed to call a function is known during compile time. • It is achieved during normal function calls, like function overloading or operator overloading. • It is faster in execution, and the function call is resolved before run time. • It provides less flexibility as compared to dynamic binding.	• It happens at run time. • This type of binding is also called late binding. • It takes place when all the information needed to call a function is not known during compile time. • It is achieved with the help of virtual functions. • It is slower in execution, but the function call is resolved during run time. • It provides more flexibility as compared to static binding.

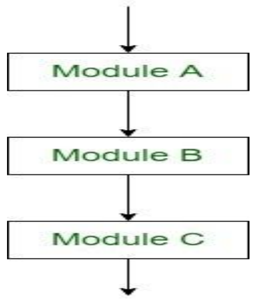
Control flow:

Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures. It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

Let us see them in detail:

1. **Sequential Logic (Sequential Flow)**
   Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



Selection Logic (Conditional Flow)

Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as **Conditional Structures**. These structures can be of three types:

1.

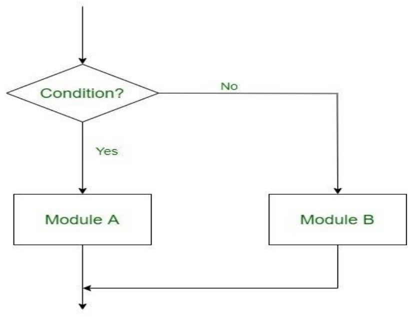
- **Single Alternative**This structure has the form:
  If (condition) then:
  ```
          [Module A]
  ```
  [End of If structure]
  Implementation:
    - C/C++ if statement with Examples
    - Java if statement with Examples
- **Double Alternative**This structure has the form:
  If (Condition), then:
  ```
          [Module A]
  ```
  Else:
  ```
          [Module B]
  ```
  [End if structure]
  Implementation:
    - C/C++ if-else statement with Examples
    - Java if-else statement with Examples
- **Multiple Alternatives**This structure has the form:
  If (condition A), then:
  ```
          [Module A]
  ```
  Else if (condition B), then:
  ```
          [Module B]
          ..
  ```

```
..
Else if (condition N), then:
        [Module N]
[End If structure]
```

Implementation:

- C/C++ if-else if statement with Examples
- Java if-else if statement with Examples

In this way, the flow of the program depends on the set of conditions that are written. This can be more understood by the following flow charts:



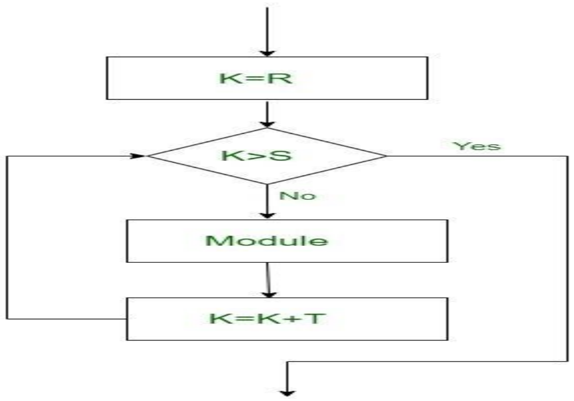
Double Alternative Control Flow

2. **Iteration Logic (Repetitive Flow)**
The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.
The two types of these structures are:

- **Repeat-For Structure**
This structure has the form:

```
Repeat for i = A to N by I:
        [Module]
[End of loop]
```

Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B. K increases or decreases according to the positive and negative value of I respectively.



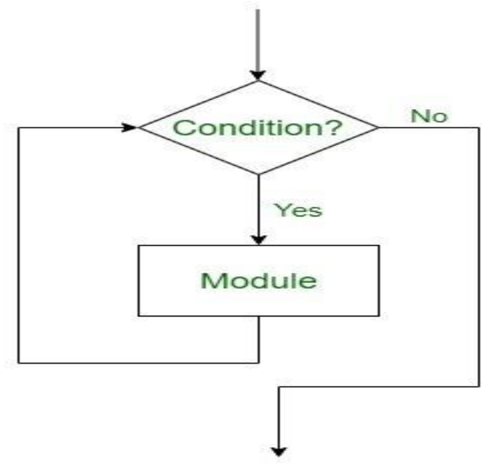
Repeat-For Flow

Implementation:

- C/C++ for loop with Examples
- Java for loop with Examples

- **Repeat-While Structure**
It also uses a condition to control the loop. This structure has the form:

```
Repeat while condition:
        [Module]
[End of Loop]
```



Repeat While Flow

Implementation:

- C/C++ while loop with Examples
- Java while loop with Examples

In this, there requires a statement that initializes the condition controlling the loop, and there must also be a statement inside the module that will change this condition leading to the end of the loop.

Control Abstraction:-

Subroutine:

Control abstraction is association of a name with a program fragment that performs an operation, and is thought of its purpose or function, rather than its implementation.

Subroutine is a principal mechanism for control abstraction.

Procedure is a subroutine that does not return a value.

Function is a subroutine that returns a value.

Parameter Passing

Parameter names that appear in the declaration of a subroutine are formal parameters. Expressions that are passed to a subroutine in a particular call are actual parameters or arguments.

```
int max (int a, int b)
{ … }
…
x = max(y+5, 10);
```

Parameter Passing Modes

We will talk about three common modes. For a language with a value model of variables

• Call by value

• Call by reference For a language with a reference model of variables

• Call by sharing

Call by value

Each actual parameter is assigned into the corresponding formal parameter. The two are independent. (A copy of actual parameter's value is passed).

Call by reference

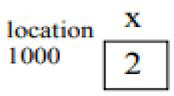
Each formal parameter is a new name for the corresponding actual parameter. The two refer to the same object. (The l-value of the actual parameter is passed.)

Some languages provide only one mode (e.g. C - value, Fortran - reference). Some provide both (e.g. Pascal, Ada, C++).

Call by Value (for Value Model of Variables)

Value of an actual parameter is assigned to a formal parameter. Both parameters are independent.

```
(* Pascal *)
var x : integer;      (* global *)

procedure foo(y : integer)
  y := 3;
  println(x);    (* print 2 *)
end;
…
x := 2;
foo(x);          (* Value 2 is passed. *)
println(x);      (* print 2 *)
```

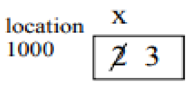


Call by Reference (for Value Model of Variables)

Formal parameter is a new name for actual parameter; the two are the same object.

If change is made on the object through the formal parameter within the subroutine, the change is visible through the actual parameter outside the subroutine.

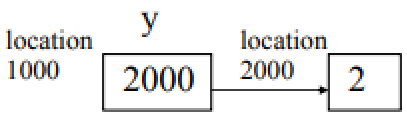
```
(* Pascal *)
var x : integer;      (* global *)

procedure foo(var y : integer)
  y := 3;
  println(x);    (* print 3 *)
end;
…
x := 2;
foo(x);          (* Location 1000 is passed. *)
println(x);      (* print 3 *)
```



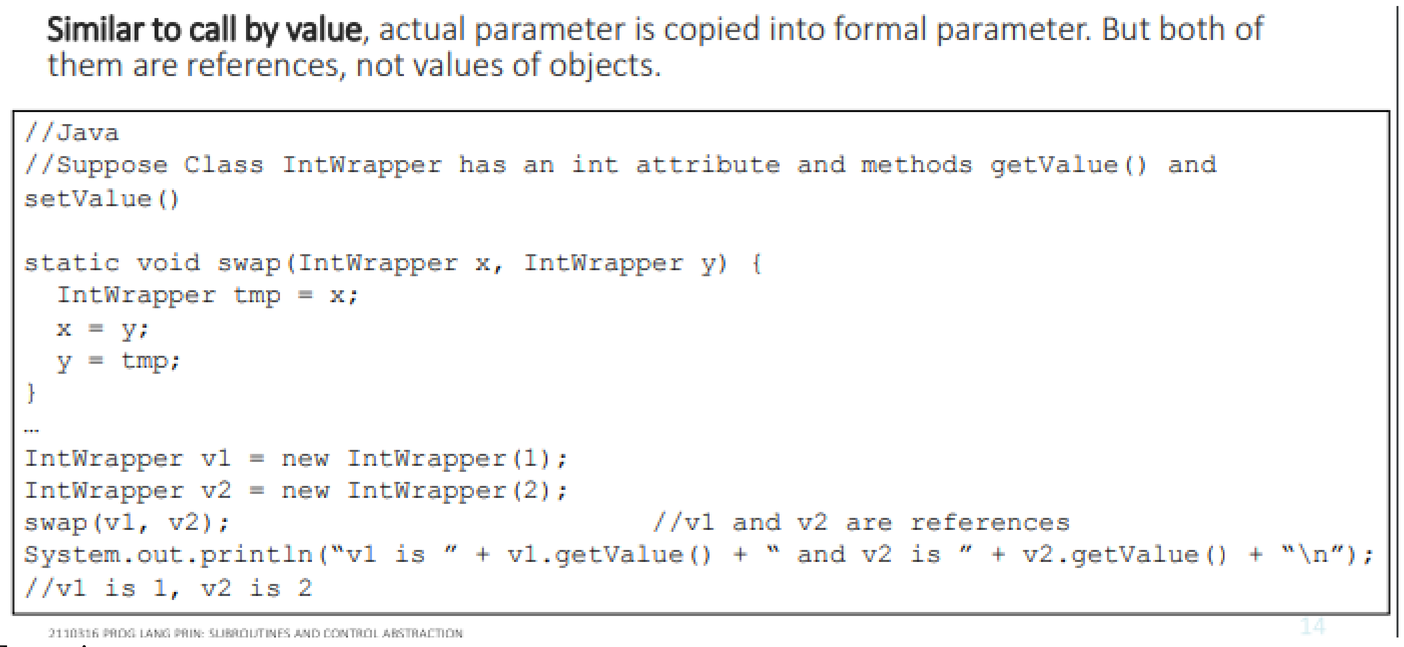
Call by sharing

- Since an actual parameter is already a reference to an object, it is copied to the formal parameter (e.g. passing location 2000).

```
y = 2
```



- The actual parameter and formal parameter refer to (or share) the same object.
- Many people mix it up with call by value and call by reference since the value that is passed is a reference to an object.



Similar to call by value, actual parameter is copied into formal parameter. But both of them are references, not values of objects.

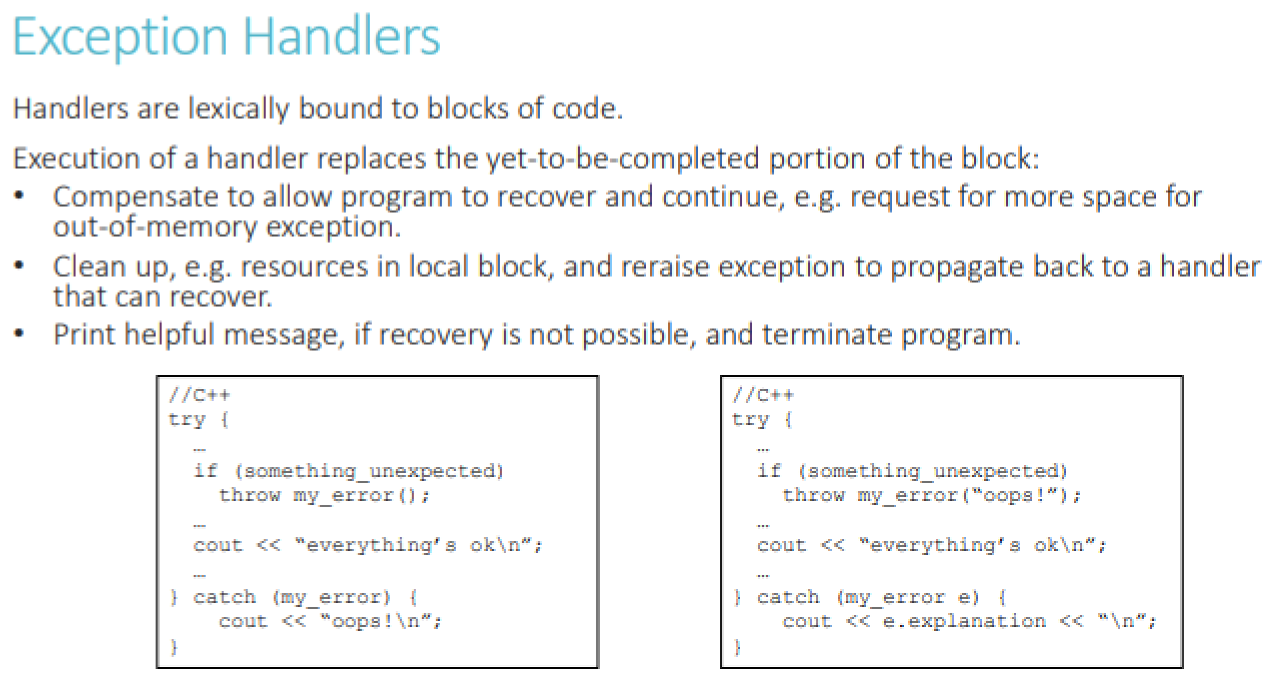
```
//Java
//Suppose Class IntWrapper has an int attribute and methods getValue() and
setValue()

static void swap(IntWrapper x, IntWrapper y) {
  IntWrapper tmp = x;
  x = y;
  y = tmp;
}
…
IntWrapper v1 = new IntWrapper(1);
IntWrapper v2 = new IntWrapper(2);
swap(v1, v2);                          //v1 and v2 are references
System.out.println("v1 is " + v1.getValue() + " and v2 is " + v2.getValue() + "\n");
//v1 is 1, v2 is 2
```

2110316 PROG LANG PRIN: SUBROUTINES AND CONTROL ABSTRACTION 14

Exception

Exception is unexpected or unusual condition that arises during program execution. It may be detected automatically by language implementation, or program may raise it explicitly, e.g.

• Unexpected end of file when reading input
• Reading in a string when expecting a number
• Arithmetic overflow
• Division by zero
• Subscript error
• Null pointer dereference Exception handling facility is provided by more recent languages, e.g. Ada, C++, Java, C#, ML, Python, PHP, Ruby.



Exception Handlers

Handlers are lexically bound to blocks of code.

Execution of a handler replaces the yet-to-be-completed portion of the block:

- Compensate to allow program to recover and continue, e.g. request for more space for out-of-memory exception.
- Clean up, e.g. resources in local block, and reraise exception to propagate back to a handler that can recover.
- Print helpful message, if recovery is not possible, and terminate program.

```
//C++
try {
  …
  if (something_unexpected)
    throw my_error();
  …
  cout << "everything's ok\n";
  …
} catch (my_error) {
    cout << "oops!\n";
}
```

```
//C++
try {
  …
  if (something_unexpected)
    throw my_error("oops!");
  …
  cout << "everything's ok\n";
  …
} catch (my_error e) {
    cout << e.explanation << "\n";
}
```

We can also represent exception in –

Multiple catch statement.

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use multi-catch block.

Catching multiple exception.

Catching multiple exceptions using the catch statement in Java 7 allows developers to *handle multiple exception types in a single catch block*, reducing code redundancy and making the code more concise.

```
catch(FirstException | SecondException | ThirdException e)
{
……
…..
}
```

Data Types:-

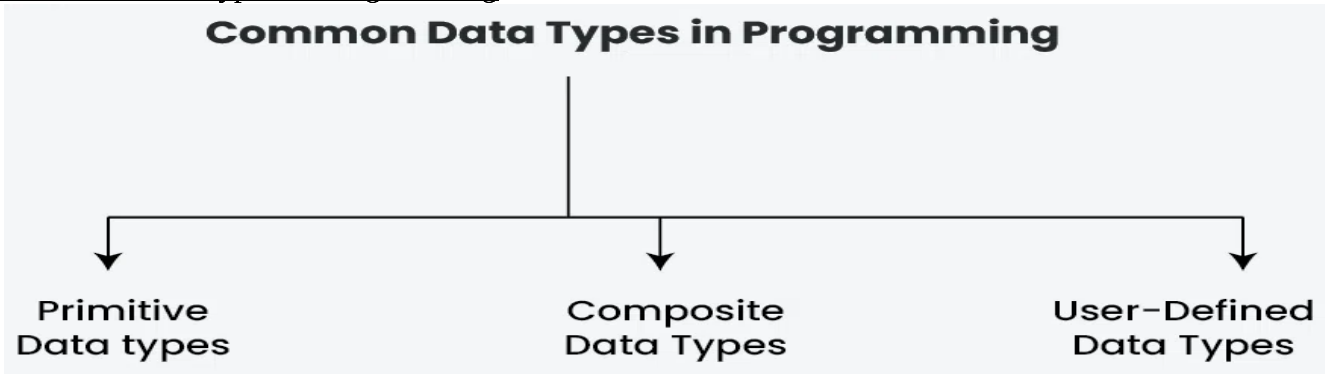
An attribute that identifies a piece of data and instructs a computer system on how to interpret its value is called a data type.

The term "data type" in software programming describes the kind of value a variable possesses and the kinds of mathematical, relational, or logical operations that can be performed on it without leading to an error. Numerous programming languages, for instance, utilize the data types string, integer, and floating point to represent text, whole numbers, and values with decimal points, respectively. An interpreter or compiler can determine how a programmer plans to use a given set of data by looking up its data type.

The data comes in different forms. Examples include:

- your name – a string of characters
- your age – usually an integer
- the amount of money in your pocket- usually decimal type
- today's date – written in date time format

Common Data Types in Programming:



1. Primitive Data Types:

Primitives are predefined data types that are independent of all other kinds and include basic values of particular attributes, like text or numeric values. They are the most fundamental type and are used as the foundation for more complex data types. Most computer languages probably employ some variation of these simple data types.

2. Composite Data Types:

Composite data types are made up of various primitive kinds that are typically supplied by the user. They are also referred to as user-defined or non-primitive data types. Composite types fall into four main categories: semi-structured (stores data as a set of relationships); multimedia (stores data as images, music, or videos); homogeneous (needs all values to be of the same data type); and tabular (stores data in tabular form).

3. User Defined Data Types:

A user-defined data type (UDT) is a data type that derived from an existing data type. You can use other built-in types already available and create your own customized data types.

Common Primitive Data Types in Programming:

Some common primitive datatypes are as follow:

Data Type	Definition	Examples
Integer (int)	represent numeric data type for numbers without fractions	300, 0 , -300
Floating Point (float)	represent numeric data type for numbers with fractions	34.67, 56.99, -78.09
Character (char)	represent single letter, digit, punctuation mark, symbol, or blank space	a , 1, !
Boolean (bool)	True or false values	true- 1, false- 0
Date	Date in the YYYY-MM-DD format (ISO 8601 syntax)	2024-01-01
Time	Time in the hh:mm:ss format for the time of day, time since an event, or time interval between events	12:34:20
Datetime	Date and time together in the YYYY-MM-DD hh:mm:ss format	2024 -01-01 12:34:20

Common Composite Data Types:

Some common composite data types are as follow:

Data Type	Definition	Example
String (string)	Sequence of characters, digits, or symbols—always treated as text	hello , ram , i am a girl
array	List with a number of elements in a specific order—typically of the same type	arr[4]= [0 , 1 , 2 , 3]
pointers	Blocks of memory that are dynamically allocated are managed and stored	*ptr=9

Common User-Defined Data Types:

Some common user defined data types are as follow:

Data Type	Definition	Example
Enumerated Type (enum)	Small set of predefined unique values (elements or enumerators) that can be text-based or numerical	Sunday -0, Monday -1
Structure	allows to combining of data items of different kinds	struct s{ …}
Union	contains a group of data objects that can have varied data types	union u {…}

Static vs. Dynamic Typing in Programming:

Characteristic	Static Typing	Dynamic Typing
Definition of Data Types	Requires explicit definition of data types	Data types are determined at runtime
Type Declaration	Programmer explicitly declares variable types	Type declaration is not required
Error Detection	Early error detection during compile time	Errors may surface at runtime
Code Readability	Explicit types can enhance code readability	Code may be more concise but less explicit
Flexibility	Less flexible as types are fixed at compile time	More flexible, allows variable types to change
Compilation Process	Requires a separate compilation step	No separate compilation step needed
Example Languages	C, Java, Swift	Python, JavaScript, Ruby

Abstraction:
Abstraction is used to hide background details or any unnecessary implementation about the data so that users only see the required information. It is one of the most important and essential features of object-oriented programming.

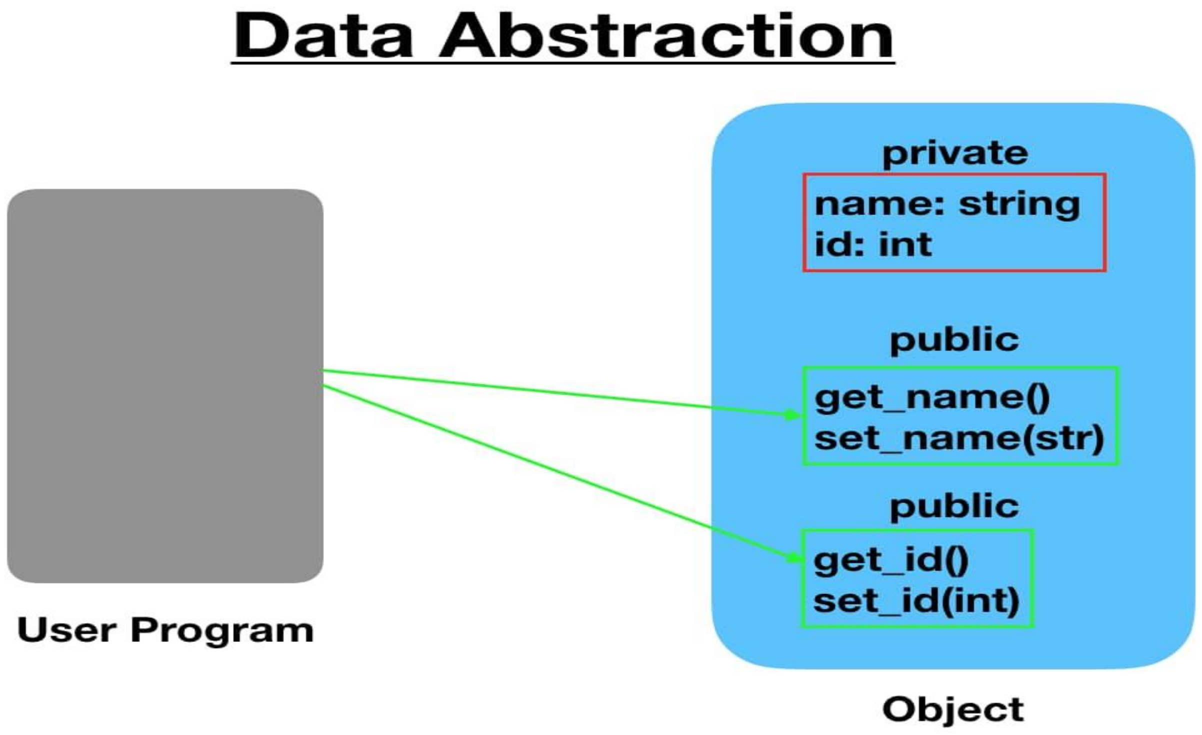
Types of Abstraction

Abstraction using classes

Abstraction can be implemented with classes. Classes have private and public identifiers to limit the scope of any variable or a function.

Abstraction in header files

Header files of many languages store some pre-defined function, for example, the pow() function in C++, .sort(), etc. A user knows how and when to use them; however, their workings are kept hidden in these header files/libraries.



An example of Data Abstraction

Users cannot access the private variables in the above illustration. However, they can be accessed and modified by the set() and get() methods. Let's look at an example below:

Although members declared as public in a class can be accessed from anywhere in the program, members declared as private in a class can only be accessed from within the class.

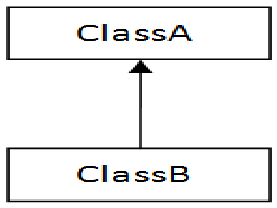
Inheritance:-

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

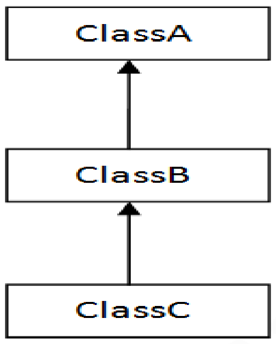
The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

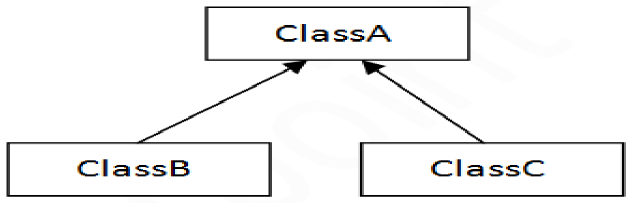
Types of inheritance:-



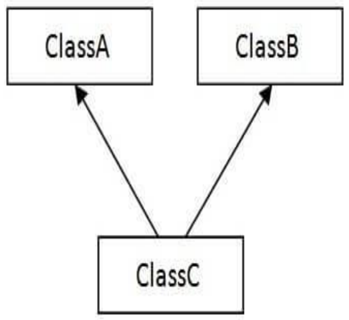
1) Single



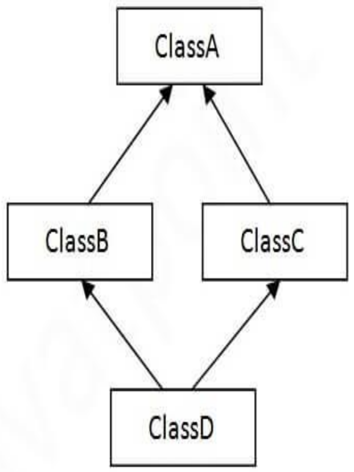
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

Module-3

Lambda Calculus:

The Lambda-calculus is the computational model the functional languages are based on.
It is a very simple mathematical formalism. In the theory of the λ-calculus one is enabled to formalize, study and investigate properties and notions common to all the functional languages, without being burdened by all the technicalities of actual languages, useless from a theoretical point of view.
In a sense, the λ-calculus it is a *paradigmatic* and extremely simple functional language.

The concepts the Lambda-calculus is based on are those that are fundamental in all functional programming languages:

variable (*formalisable by* $x, y, z, \ldots$)
abstraction (anonymous function) *(formalisable by* $\lambda x.M$
where M is a term and x a variable)
application (*formalizable by* MN, *where M and N are terms*)

We have seen that these are indeed fundamental notions in functional programming. It seems, however, that there are other important notions: basic elements, basic operators and the possibility of giving names to expressions.
Even if it could appear very strange, this notions actually are not really fundamental ones: they can be derived from the first ones.
This means that our fucntional computational model can be based on the concepts of variable, functional abstracion and application alone.
Using the formalization of these three concepts we form the lambda-terms. These terms represent both *programs* and *data* in our computational model (whose strenght in fact strongly depends on the fact it does not distinguish between "programs" and "data".)

The formal definition of the terms of the Lambda-calculus (lambda-terms) is given by the following grammar:

$$\Lambda ::= X \mid (\Lambda\Lambda) \mid \lambda X.\Lambda$$

where Λ stands for the set of lambda-terms and X is a metavariable that ranges over the (numerable) set of variables (x, y, v, w, z, x2, x2,...)
Variable and *application* are well-known concepts, but what is abstraction?
Abstraction is needed to create anonymous functions (i.e. functions without a name). We have seen that being able to specify an anonymous function is very important in fucntional programming, but it is also important to be able to associate an identifier to an anonymous function. Nonetheless we can avoid to introduce in our model the notion of "giving a name to a function". You could ask: how is it possible to define a function like fac without being able to refer to it trought its name??? Believe me, it is possible!.

The term λx**.** M represents the anonymous function that, given an input x, returns the "value" of the body M.
Other notions, usually considered as primitive, like basic operators (+, *, -, ...) or natural numbers, are not strictly needed in our computational model (indeed they can actually be treated as *derived* concepts.)

Bracketing conventions

For readability sake, it is useful to abbreviate nested abstractions and applications as follows:

$(\lambda x_1.(\lambda x_2. \ldots . (\lambda x_n.M) \ldots)$ is abbreviated by $(\lambda x_1 x_2 \ldots x_n.M)$

$(\ldots ((M_1M_2)M_3) \ldots M_n)$ is abbreviated by $(M_1M_2M_3 \ldots M_n)$

We also use the convention of dropping outermost parentheses and those enclosing the body of an abstraction.

Example:

$(\lambda x.(x\ (\lambda y.(yx))))$ can be written as $\lambda x.x(\lambda y.yx)$

In an abstraction $\lambda x.P$, the term P is said to be the **scope** of the abstraction.
In a term, the occurrence of a variable is said to be **bound** if it occurs in the scope of an abstraction (i.e. it represents the argument in the body of a function).
It is said to be **free** otherwise.

These two concepts can be formalized by the following definitions.

Definition of bound variable
We define BV(M), the set of the Bound Variables of M, by induction on the term as follows:

$BV(x) = \Phi$ (the emptyset)
$BV(PQ) = BV(P) \cup BV(Q)$
$BV(\lambda x.P) = \{x\} \cup BV(P)$

Definition of free variable
We define FV(M), the set of the Free Variables of M, by induction on the term as follows:

$FV(x) = \{x\}$
$FV(PQ) = FV(P) \cup FV(Q)$
$FV(\lambda x.P) = FV(P) \setminus \{x\}$

Substitution

The notation **M [L / x]** denotes the term M in which any **free** (i.e. not representing any argument of a function) occurrence of the variable x in M is replaced by the term L.

Definition of substitution (by induction on the structure of the term)
1. If M is a variable (M = y) then:

$$y\,[L/x] \equiv \begin{cases} L & \text{if } x=y \\ y & \text{if } x\neq y \end{cases}$$

2. If M is an application (M = PQ) then:

$PQ\,[L/x] = P[L/x]\ Q[L/x]$

3. If M is a lambda-abstraction (M = λy.P) then:

$$\lambda y.P[L/x] \equiv \begin{cases} \lambda y.P & \text{if } x=y \\ \lambda y.(P\,[L/x]) & \text{if } x\neq y \end{cases}$$

Notice that in a lambda abstraction a substitution is performed only if the variable to be substituted is free.

Great care is needed when performing a substitution, as the following example shows:

<u>*Example*</u>*:*

Both of the terms (λx.z) and (λy.z) obviously represent the constant function which returns z for any argument

Let us assume we wish to apply to both of these terms the substitution [x / z] (we replace x for the free variable z). If we do not take into account the fact that the variable x is free in the term x and bound in (λx.z) we get:

(λx.z) [x / z] => λx.(z[x / z]) => λx.x representing the identity function
(λy.z) [x / z] => λy.(z[x/ z]) => λy.x representing the function that returns always x

This is absurd, since both the initial terms are intended to denote the <u>very same thing</u> (a constant function returning z), but by applying the substitution we obtain two terms denoting very different things.

Hence a necessary condition in order the substitution M[L/x] be correctly performed is that free variables in L does not change status (i.e. become not free) after the substitution. More formally: FV(L) and BV(M) must be <u>disjoint</u> sets.
Such a problem is present in all the programming languages, also imperative ones.

<u>*Example*</u>*:*

Let us consider the following substitution:

(λz.x) [zw/x]

The free variable z in the term to be substituted would become bound after the substitution. This is meaningless. So, what it has to be done is to rename the bound variables in the term on which the substitution is performed, in order the condition stated above be satisfied:

(λq.x) [zw/x]

now, by performing the substitution, we correctly get: (λq.zw)

The example above introduces and justifies the notion of *alpha-convertibility*

The notion of alpha-convertibility

A bound variable is used to represent where, in the body of a function, the argument of the function is used. In order to have a meaningful calculus, terms which differ just for the names of their bound variables have to be considered identical. We could introduce such a notion of identity in a very formal way, by defining the relation of **α-convertibility**. Here is an example of two terms in such a relation.

$\lambda z.z =_\alpha \lambda x.x$

We do not formally define here such a relation. For us it is sufficient to know that two terms are α-convertible whenever one can be obtained from the other by simply renaming the bound variables. Obviously alpha convertibility mantains completely unaltered both the meaning of the term and how much this meaning is explicit. From now on, we shall implicitely work on λ-terms modulo alpha conversion. This means that from now on two alpha convertible terms like λz.z and λx.x are for us the **very same** term.

Higher Order Function:-

Functions are values just like any other value in OCaml. What does that mean exactly? This means that we can pass functions around as arguments to other functions, that we can store functions in data structures, that we can return functions as a result from other functions.

Let us look at why it is useful to have higher-order functions. The first reason is that it allows you to write general, reusable code. Consider these functions `double` and `square` on integers:

```
let double x = 2 * x

let square x = x * x
```

Let's use these functions to write other functions that quadruple and raise a number to the fourth power:

```
let quad x   = double (double x)

let fourth x = square (square x)
```

There is an obvious similarity between these two functions: what they do is apply a given function twice to a value. By passing in the function to another function `twice` as an argument, we can abstract this functionality:

```
let twice f x = f (f x)

(* twice : ('a -> 'a) -> 'a -> 'a *)
```

Using `twice`, we can implement `quad` and `fourth` in a uniform way:

```
let quad   x = twice double x

let fourth x = twice square x
```

Higher-order functions either take other functions as input or return other functions as output (or both). The function `twice` is higher-order: its input `f` is a function. And—recalling that all OCaml functions really take only a single argument—its output is technically `fun x -> f (f x)`, so `twice` returns a function hence is also higher-order in that way. Higher-order functions are also known as *functionals*, and programming with them could be called *functional programming*—indicating what the heart of programming in languages like OCaml is all about

Evaluation Strategies

The parameter evaluation strategy adopted by a programming language defines when parameters are evaluated during function calls. There are two main strategies: strict and lazy evaluation.

Strict Evaluation

The strict evaluation strategy consists in the full evaluation of parameters before passing them to functions. The two most common evaluation strategies: by-value and by-reference, fit into this category.

Call-by-Value: The call-by-value strategy consists in copying the contents of the actual parameters into the formal parameters. State changes performed in the formal parameters do not reflect back into the actual parameters. A well-known example of this type of behavior is given by the swap function below, implemented in C:

```
void swap(int x, int y) {
  int aux = x;
  x = y;
  y = aux;;
}
int main() {
  int a = 2;
  int b = 3;
  printf("%d, %d\n", a, b);
  swap(a, b);
  printf("%d, %d\n", a, b);
}
```

Once the swap function is called, the contents of variables a and b are copied to the formal parameters x and y respectively. The data exchange that happens in the body of swap only affect the formal parameters, but not the actual ones. In other words, the swap call is innocuous in this program. In order to circumvent this semantics, the language C lets us use pointers to pass the address of a location, instead of its contents. Thus, the function below swaps the contents of two variables, as intended:

```
void swap(int *x, int *y) {
  int aux = *x;
  *x = *y;
  *y = aux;
}
int main() {
  int a = 2;
  int b = 3;
  printf("%d, %d\n", a, b);
  swap(&a, &b);
  printf("%d, %d\n", a, b);
}
```

The call-by-value strategy is very common among programming languages. It is the strategy of choice in C, Java, Python and even C++, although this last language also supports call-by-reference.

Call-by-Reference: whereas in the call-by-value strategy we copy the contents of the actual parameter to the formal parameter, in the call-by-reference we copy the address of the actual parameter to the formal one. A few languages implement the call-by-reference strategy. C++ is one of them. The program below re-implements the swap function, using the call-by-reference policy:

```
void swap(int &x, int &y) {
  int aux = x;
  x = y;
  y = aux;
}
int main() {
  int a = 2;
  int b = 3;
  printf("%d, %d\n", a, b);
  swap(a, b);
  printf("%d, %d\n", a, b);
}
```

In C++, parameter passing by reference is a syntactic sugar for the use of pointers. If we take a careful look into the assembly code that g++, the C++ compiler, produces for the function swap, above, and the function swap with pointers, we will realize that it is absolutely the same.

The call-by-reference might be faster than the call-by-value if the data-structures passed to the function have a large size. Nevertheless, this strategy is not present in the currently main-stream languages, but C++. Parameter passing by reference might lead to programs that are difficult to understand. For instance, the function below also implements the swap function; however, it combines three xor operations to avoid the need for an auxiliary variable.

```
void xor_swap(int &x, int &y) {
  x = x ^ y;
  y = x ^ y;
  x = x ^ y;
}
```

This function might lead to unexpected results if the formal parameters x and y alias the same location. For instance, the program below, which uses the xor_swap implementation zeros the actual parameter, instead of keeping its value:

```
int main() {
  int a = 2;
  int b = 3;
  printf("%d, %d\n", a, b);
  xor_swap(a, a);
  printf("%d, %d\n", a, b);
}
```

Lazy Evaluation

The strict evaluation strategies force the evaluation of the actual parameters before passing them to the called function. To illustrate this fact, the program below, implemented in python, loops.

```
def andF(a, b):
  if not a:
    return True
  else:
    return b

def g(x):
  if g(x):
    return True
  else:
    return False

f = andF(False, g(3))
```

There are parameter passing strategies that do not require the parameters to be evaluated before being passed to the called function. These strategies are called *lazy*. The three most well-known lazy strategies are *call by macro expansion*, *call by name* and *call by need*.

Call by Macro Expansion: many programming languages, including C, lisp and scheme, provide developers with a mechanism to add new syntax to the core language grammar called macros. Macros are expanded into code by a macro preprocessor. These macros might contain arguments, which are copied in the final code that the preprocessor produces. As an example, the C program below implements the swap function via a macro:

```
#define SWAP(X,Y) {int temp=X; X=Y; Y=temp;}
int main() {
  int a = 2;
  int b = 3;
  printf("%d, %d\n", a, b);
  SWAP(a, b);
  printf("%d, %d\n", a, b);
}
```

This macro implements a valid swap routine. The preprocessed program will look like the code below. Because the body of the macro is directly copied into the text of the calling program, it operates on the context of that program. In other words, the macro will refer directly to the variable names that it receives, and not to their values.

```
int main() {
  int a = 2;
  int b = 3;
  printf("%d, %d\n", a, b);
  { int tmp = (a); (a) = (b); (b) = tmp; };
  printf("%d, %d\n", a, b);
}
```

The expressions passed to the macro as parameters are evaluated every time they are used in the body of the macro. If the argument is never used, then it is simply not evaluated. As an example, the program below will increment the variable b twice:

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
int main() {
  int a = 2, b = 3;
  int c = MAX(a, b++);
  printf("a = %d, b = %d, c = %d\n", a, b, c);
}
```

Macros suffer from one problem, called *variable capture*. If a macro defines a variable v that is already defined in the environment of the caller, and v is passed to the macro as a parameter, the body of the macro will not be able to distinguish one occurrence of v from the other. For instance, the program below has a macro that defines a variable temp. The call inside main causes the variable temp defined inside this function to be captured by the definition inside the macro's body.

```
#define SWAP(X,Y) {int temp=X; X=Y; Y=temp;}
int main() {
  int a = 2;
  int temp = 17;
  printf("%d, temp = %d\n", a, temp);
  SWAP(a, temp);
  printf("%d, temp = %d\n", a, temp);
}
```

Once this program is expanded by the C preprocessor, we get the code below. This program fails to exchange the values of variables temp and a:

```
int main() {
  int a = 2;
  int temp = 17;
  printf("%d, temp = %d\n", a, temp);
  {int temp=a; a=temp; temp=temp;};
  printf("%d, temp = %d\n", a, temp);
}
```

There are a number of lazy evaluation strategies that avoid the variable capture problem. The two best known techniques are call-by-name and call-by-need.

Call by Name: in this evaluation strategy the actual parameter is only evaluated if used inside the function; however, this evaluation uses the context of the caller routine. For instance, in the example below, taken from Weber's book, we have a function g that returns the integer 6. Inside the function f, the first assignment, e.g., b = 5, stores 5 in variable i. The second assignment, b = a, reads the value of i, currently 5, and adds 1 to it. This value is then stored at i.

```
void f(by-name int a, by-name int b) {
  b=5;
  b=a;
}
int g() {
  int i = 3;
  f(i+1,i);
  return i;
}
```

Very few languages implement the call by name evaluation strategy. The most eminent among these languages is Algol. Simula, a direct descendent of Algol, also implements call by name, as we can see in this example. The call by name always causes the evaluation of the parameter, even if this parameter is used multiple times. This behavior might be wasteful in referentially transparent languages, because, in these languages variables are immutable. There is an evaluation strategy that goes around this problem: *the call by need*.

Call by Need: in this evaluation strategy, a parameter is evaluated only if it is used. However, once the first evaluation happens, its result is cached, so that further uses of the parameter do not require a re-evaluation. This mechanism provides the following three guarantees:

- The expression is only evaluated if the result is required by the calling function;
- The expression is only evaluated to the extent that is required by the calling function;
- The expression is never evaluated more than once, called applicative-order evaluation.

Haskell is a language notorious for using call by need. This evaluation strategy is a key feature that the language designers have used to keep Haskell a purely functional language. For instance, call by need lets the language to simulate the input channel as an infinite list, which must be evaluated only as much as data has been read. An an example, the program below computes the n-th term of the Fibonacci Sequence. Yet, the function fib, that generates this sequence, has no termination condition!

```
fib m n = m : (fib n (m+n))

getIt [] _ = 0
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)

getN n = getIt (fib 0 1) n
```

The getIt function expands the list produced by fib only as many times as it is necessary to read its n-th element. For instance, below we have a sequence of calls that compute the 4-th element of the Fibonacci sequence:

```
getIt (fib 0 1) 4
= getIt (0 : fib 1 1) 4

getIt (fib 1 1) 3
= getIt (1 : fib 1 2) 3

getIt (fib 1 2) 2
= getIt (1 : fib 2 3) 2

getIt (fib 2 3) 1
= getIt (2 : fib 3 5) 1
= 2
```

Type Checking

Type checking is an essential process in programming where a compiler verifies and enforces constraints on data types within a program. It ensures that the code follows the syntactic and semantic rules of the

programming language, including type regulations. Type checking assigns types to values, restricts their usage accordingly, and detects any violations. The compiler's type checker module plays a crucial role in managing and computing data types, correcting errors, and maintaining the integrity of the program.
Types of Type Checking

There are two primary types of type checking:

1. **Static Type Checking:** Performed at compile time, static type checking verifies types of variables based on the program's source code. It detects errors before the program runs, enhancing code reliability.
2. **Dynamic Type Checking:** Dynamic type checking occurs at runtime and verifies types as the program executes. While more flexible, dynamic type checking may lead to runtime errors if types don't match.

Static type checking :-

Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time. It generally examines the program text during the translation of the program. Using the type rules of a system, a compiler can infer from the source text that a function (fun) will be applied to an operand (a) of the right type each time the expression fun(a) is evaluated.

Examples of Static checks include:

- **Type-checks:** A compiler should report an error if an operator is applied to an incompatible operand. For example, if an array variable and function variable are added together.
- **The flow of control checks:** Statements that cause the flow of control to leave a construct must have someplace to which to transfer the flow of control. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement, an error occurs if such an enclosing statement does not exist.
- **Uniqueness checks:** There are situations in which an object must be defined only once. For example, in Pascal an identifier must be declared uniquely, labels in a case statement must be distinct, and else a statement in a scalar type may not be represented.
- **Name-related checks:** Sometimes the same name may appear two or more times. For example in Ada, a loop may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

The Benefits of Static Type Checking:

1. Runtime Error Protection.
2. It catches syntactic errors like spurious words or extra punctuation.
3. It catches wrong names like Math and Predefined Naming.
4. Detects incorrect argument types.
5. It catches the wrong number of arguments.
6. It catches wrong return types, like return "70", from a function that's declared to return an int.

Dynamic Type Checking:

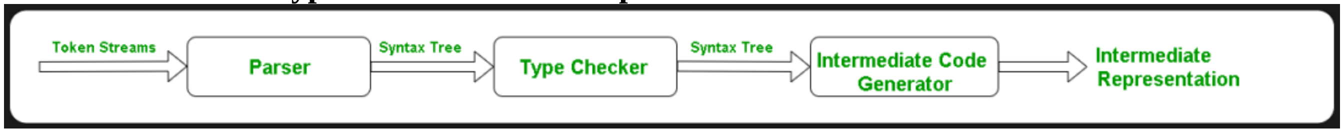
Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed. Dynamic typing results in more compact programs since it is more flexible and does not require types to be spelled out. Programming with a static type system often requires more design and implementation effort.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types, whether it is syntactically correct or not before their execution.
Static Type-Checking is also used to determine the amount of memory needed to store the variable.

The design of the type-checker depends on:

1. Syntactic Structure of language constructs.
2. The Expressions of languages.
3. The rules for assigning types to constructs (semantic rules).

The Position of the Type checker in the Compiler:



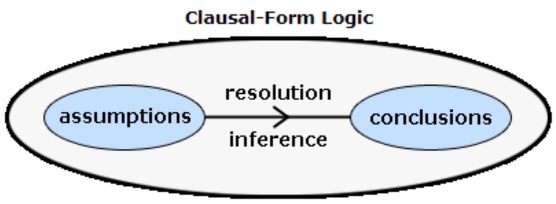
.

Logic Programming

Introduction

Artificial Intelligence (AI) is the ability for an artificial machine to act intelligently. Logic Programming is a method that computer scientists are using to try to allow machines to reason because it is useful for knowledge representation. In logic programming, logic is used to represent knowledge and inference is used to manipulate it.

The logic used to represent knowledge in logic programming is clausal form which is a subset of first-order predicate logic. It is used because first-order logic is well understood and able to represent all computational problems. Knowledge is manipulated using the resolution inference system which is required for proving theorems in clausal-form logic. The diagram below shows the essence of logic programming.



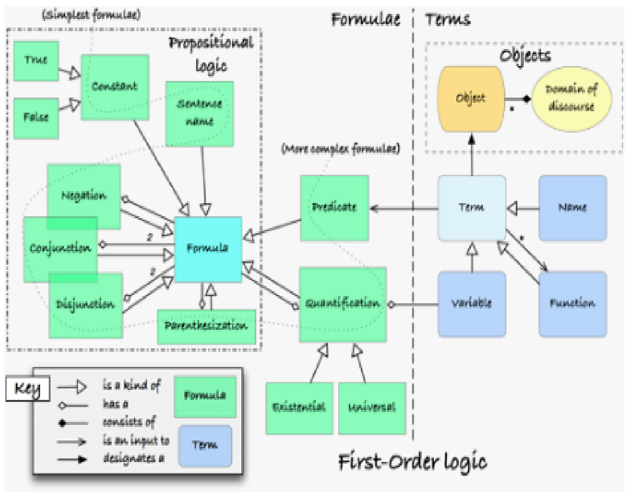
Prolog, PROgramming in LOGic, is a declarative programming language which is based on the ideas of logic programming, such as those discussed above. The idea of Prolog was to make logic look like a programming language and allow it to be controlled by a programmer to advance the research for theorem-proving.

Logic Programming

First-Order Logic (1)

< 1 2 3 >

First order logic is an extension of propositional logic. It considers whether things are true or false in a partial view of the world, called a domain. Logic programming is based upon an extension of first-order logic. As first order logic is well understood and can be mathematically provable it can be used for representing all computational problems.



First-Order Logic (2)

< 1 2 3 >

The binding strength of the quantifiers ∀x and ∃x, is that same as that of ¬. The rest of the connectives follow with the following decreasing binding strength ∧ ∨ → ↔. In Prolog, constant symbols conventionally begin with lower case letters, and variable symbols begin with upper case letters or an underscore.

In some texts, different symbols are used for the connectives:

Name	Connective	Alternative	Example
Negation	¬	~	¬a
Conjunction	∧	&	a∧b
Disjunction	∨	or	a∨b
Conditional	→	if	a→b
Biconditional	↔	iff	a↔b

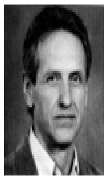
The first order language given by an alphabet consists of the set of all formulas constructable from the symbols of the alphabet. Here are some definitions of various pieces of informal grammar used in conjunction with an alphabet, when constructing first order predicate logic language:

- A **term** is either a constant symbol, a variable symbol or an n-ary function symbol applied to a tuple of n terms.
  - e.g. a, X, f(b, f(b,Y,Z),Y)
- An **atomic formula** is a predicate symbol applied to a tuple of terms.
  - e.g. likes(X,sim), borrows(book, sim)
- A **well-formed formula** is either an atomic formula or else takes one of the following forms:
  - W, ¬W, W1 ∧ W2, W1 ∨ W2, W1 → W2, W1 ↔ W2, ∀xW, ∃xW.
- In a well formed formula, the **scope** of a quantifier is defined as W. We say that the quantifier (ie ∀x or ∃x) **quantifies** every occurrence of x in W.
- An occurrence of a variable symbol x in a well formed formula W is said to be **bound** if and only if it lies within the scope of or in some quantifier (∀x or ∃x) which occurs in W.
- An occurrence of a variable symbol that is not bound is said to be **unbound** (or **free**).
- A **sentence** (or **closed formula**) is a well-formed formula in which every occurrence of every variable symbol is bound.
  - e.g. ∀x (∃y child(x) → father(y,x)) every child has a father.
- A **literal** is either an atomic formula or a negated atomic formula.
  - e.g. likes(sim, books), ¬likes(sim, pain)
- A **ground** term is a term containing no variable symbols.
- A **ground formula** is a formula containing no variable symbols.
- Atomic formulas are commonly called **predicates.**

First-Order Logic (3)

< 1 2 3 >

Model theory is a way of attributing meaning to any given logic sentences. This is done by interpretation, the process of associating the sentence with some truth-valued statement about a chosen domain. This domain can be any set of our choice, for example the set of natural numbers. Here are some definitions associated with model theory:



- A **model** is an interpretation of a sentence that gives a true value (satisfies the sentence).
- An interpretation that does not satisfy a sentence is a **counter-model** for that sentence.
- A sentence with at least one model is **satisfiable.**
  - e.g. likes(sim, chocolate).
- A sentence with no models is **unsatisfiable.**
  - e.g. ∃x∃y(likes(x, y) ∧ ¬likes(x, y)).
- A sentence in which every interpretation is a model, is **valid.**
  - e.g. ∀x∀y(likes(x, y) ∨ ¬likes(x, y)).
- A sentence (A) **logically implies** another sentence (B) if and only if every model for A is also a model for B, this is written as A |= B. This symbol is known as double turnstile.
  - e.g. ∀x(likes(sim, x)) |= likes(sim, chocolate).
- Two sentences A and B are **logically equivalent** when each one logically implies the other, this is written as A ≡ B.
  - e.g. ∀x(¬likes(x, pain)) ≡ ¬∃x(likes(x, pain))

SLD Resolution:-

SLD resolution (*Selective Linear Definite* clause resolution) is the basic inference rule used in logic programming. It is a refinement of resolution, which is both sound and refutation complete for Horn clauses.

The SLD inference rule [edit]

Given a goal clause, represented as the negation of a problem to be solved :

$$\neg L_1 \vee \cdots \vee \neg L_i \vee \cdots \vee \neg L_n$$

with selected literal $\neg L_i$, and an input definite clause:

$$L \vee \neg K_1 \vee \cdots \vee \neg K_m$$

whose positive literal (atom) L unifies with the atom L_i of the selected literal $\neg L_i$, SLD resolution derives another goal clause, in which the selected literal is replaced by the negative literals of the input clause and the unifying substitution θ is applied:

$$(\neg L_1 \vee \cdots \vee \neg K_1 \vee \cdots \vee \neg K_m \ \vee \cdots \vee \neg L_n)\theta$$

In the simplest case, in propositional logic, the atoms L_i and L are identical, and the unifying substitution θ is vacuous. However, in the more general case, the unifying substitution is necessary to make the two literals identical.

The origin of the name "SLD" [edit]

The name "SLD resolution" was given by Maarten van Emden for the unnamed inference rule introduced by Robert Kowalski.[1] Its name is derived from SL resolution,[2] which is both sound and refutation complete for the unrestricted clausal form of logic. "SLD" stands for "SL resolution with Definite clauses".

In both, SL and SLD, "L" stands for the fact that a resolution proof can be restricted to a linear sequence of clauses:

$$C_1, C_2, \cdots, C_l$$

where the "top clause" C_1 is an input clause, and every other clause C_{i+1} is a resolvent one of whose parents is the previous clause C_i. The proof is a refutation if the last clause C_l is the empty clause.

In SLD, all of the clauses in the sequence are goal clauses, and the other parent is an input clause. In SL resolution, the other parent is either an input clause or an ancestor clause earlier in the sequence.

In both SL and SLD, "S" stands for the fact that the only literal resolved upon in any clause C_i is one that is uniquely selected by a selection rule or selection function. In SL resolution, the selected literal is restricted to one which has been most recently introduced into the clause. In the simplest case, such a last-in-first-out selection function can be specified by the order in which literals are written, as in Prolog. However, the selection function in SLD resolution is more general than in SL resolution and in Prolog. There is no restriction on the literal that can be selected.

SLD resolution strategies [edit]

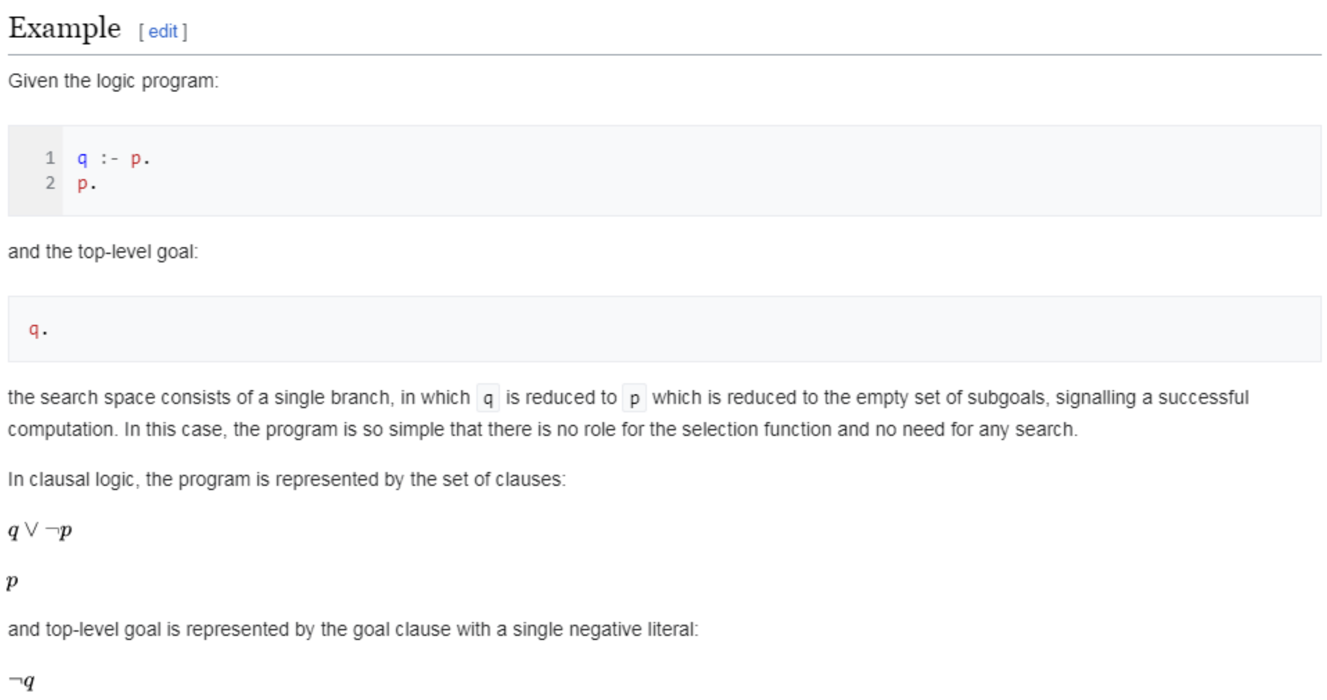
SLD resolution implicitly defines a search tree of alternative computations, in which the initial goal clause is associated with the root of the tree. For every node in the tree and for every definite clause in the program whose positive literal unifies with the selected literal in the goal clause associated with the node, there is a child node associated with the goal clause obtained by SLD resolution.

A leaf node, which has no children, is a success node if its associated goal clause is the empty clause. It is a failure node if its associated goal clause is non-empty but its selected literal unifies with no positive literal of definite clauses in the program.

SLD resolution is non-deterministic in the sense that it does not determine the search strategy for exploring the search tree. Prolog searches the tree depth-first, one branch at a time, using backtracking when it encounters a failure node. Depth-first search is very efficient in its use of computing resources, but is incomplete if the search space contains infinite branches and the search strategy searches these in preference to finite branches: the computation does not terminate. Other search strategies, including breadth-first, best-first, and branch-and-bound search are also possible. Moreover, the search can be carried out sequentially, one node at a time, or in parallel, many nodes simultaneously.

SLD resolution is also non-deterministic in the sense, mentioned earlier, that the selection rule is not determined by the inference rule, but is determined by a separate decision procedure, which can be sensitive to the dynamics of the program execution process.

The SLD resolution search space is an or-tree, in which different branches represent alternative computations. In the case of propositional logic programs, SLD can be generalised so that the search space is an and-or tree, whose nodes are labelled by single literals, representing subgoals, and nodes are joined either by conjunction or by disjunction. In the general case, where conjoint subgoals share variables, the and-or tree representation is more complicated.



Example [edit]

Given the logic program:

```
q :- p.
p.
```

and the top-level goal:

```
q.
```

the search space consists of a single branch, in which `q` is reduced to `p` which is reduced to the empty set of subgoals, signalling a successful computation. In this case, the program is so simple that there is no role for the selection function and no need for any search.

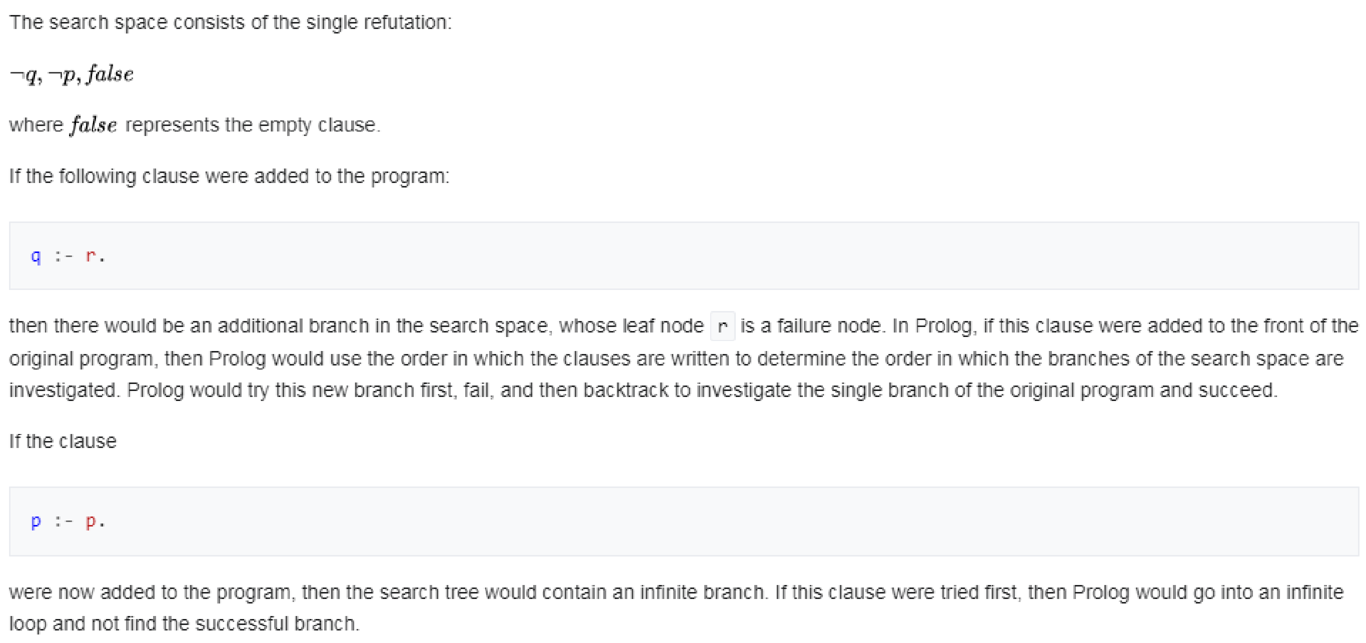
In clausal logic, the program is represented by the set of clauses:

$q \vee \neg p$

p

and top-level goal is represented by the goal clause with a single negative literal:

$\neg q$



The search space consists of the single refutation:

$\neg q, \neg p, false$

where $false$ represents the empty clause.

If the following clause were added to the program:

```
q :- r.
```

then there would be an additional branch in the search space, whose leaf node `r` is a failure node. In Prolog, if this clause were added to the front of the original program, then Prolog would use the order in which the clauses are written to determine the order in which the branches of the search space are investigated. Prolog would try this new branch first, fail, and then backtrack to investigate the single branch of the original program and succeed.

If the clause

```
p :- p.
```

were now added to the program, then the search tree would contain an infinite branch. If this clause were tried first, then Prolog would go into an infinite loop and not find the successful branch.

Unification:-

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\boldsymbol{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.
- **Example: Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)......... (i)
P(a, f(z))......... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

```
Step. 1: If $\Psi_1$ or $\Psi_2$ is a variable or constant, then:
       a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL.
       b) Else if $\Psi_1$is a variable,
              a. then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
              b. Else return { ($\Psi_2$/ $\Psi_1$)}.
       c) Else if $\Psi_2$ is a variable,
              a. If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE,
              b. Else return {( $\Psi_1$/ $\Psi_2$)}.
       d) Else return FAILURE.
Step.2: If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$ are not same, then return FAILURE.
Step. 3: IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.
Step. 4: Set Substitution set(SUBST) to NIL.
Step. 5: For i=1 to the number of elements in Ψ1.
      a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put
the result into S.
      b) If S = failure then returns Failure
      c) If S ≠ NIL then do,
            a. Apply S to the remainder of both L1 and L2.
            b. SUBST= APPEND(S, SUBST).
Step.6: Return SUBST.
```

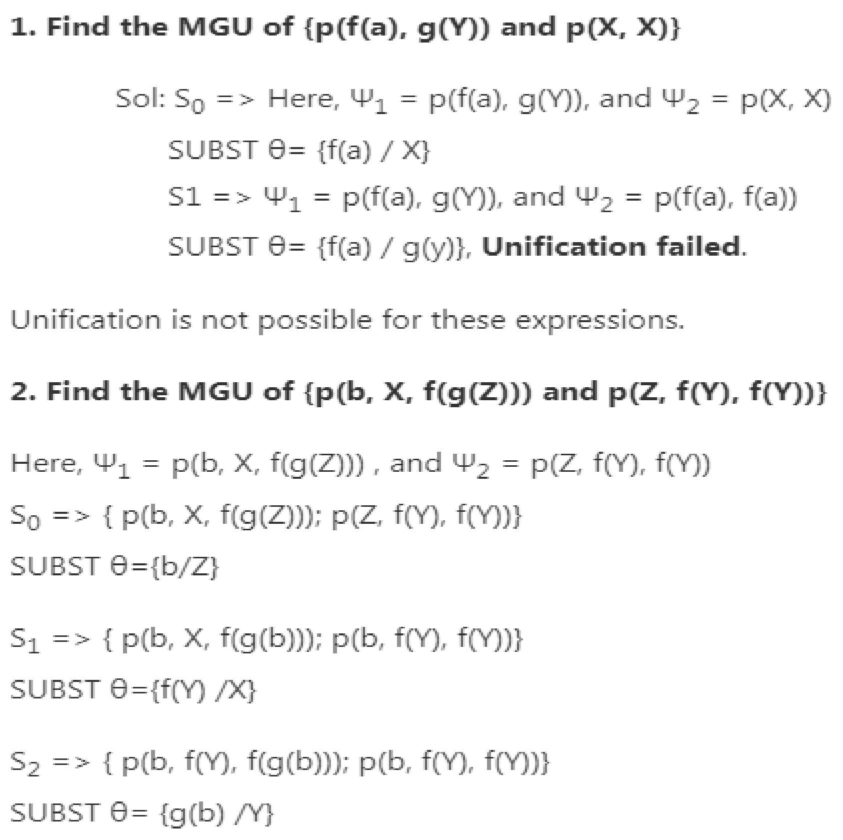
Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

a. Check for Identical expression match.

b. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:
    a. Substitute t_i / v_i in the existing substitutions
    b. Add t_i /v_i to the substitution setlist.
    c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.



1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S_0 => Here, Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(X, X)
SUBST θ= {f(a) / X}
S1 => Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(f(a), f(a))
SUBST θ= {f(a) / g(y)}, **Unification failed**.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ_1 = p(b, X, f(g(Z))) , and Ψ_2 = p(Z, f(Y), f(Y))
S_0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}
SUBST θ={b/Z}

S_1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}
SUBST θ={f(Y) /X}

S_2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}
SUBST θ= {g(b) /Y}

S_2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} **Unified Successfully.**

And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ_1 = {p (X, X), and Ψ_2 = p (Z, f(Z))

S_0 => {p (X, X), p (Z, f(Z))}

SUBST θ= {X/Z}

S1 => {p (Z, Z), p (Z, f(Z))}

SUBST θ= {f(Z) / Z}, **Unification Failed.**

Hence, unification is not possible for these expressions.

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}

S_0 => {prime(11) , prime(y)}

SUBST θ= {11/y}

S_1 => {prime(11) , prime(11)} , **Successfully unified.**

Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ_1 = Q(a, g(x, a), f(y)), and Ψ_2 = Q(a, g(f(b), a), x)

S_0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}

SUBST θ= {f(b)/x}

S_1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}

S_1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, **Successfully Unified.**

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)

S_0 => { knows(Richard, x); knows(Richard, John)}

SUBST θ= {John/x}

S_1 => { knows(Richard, John); knows(Richard, John)}, **Successfully Unified.**

Unifier: {John/x}.

Negations:

In logic, **negation**, also called the **logical not** or **logical complement**, is an operation that takes a proposition P to another proposition "not P", standing for "P is not true", written $\neg P$, $\sim P$ or $\overline{P}$. It is interpreted intuitively as being true when P is false, and false when P is true.[1][2] Negation is thus a unary logical connective. It may be applied as an operation on notions, propositions, truth values, or semantic values more generally. In classical logic, negation is normally identified with the truth function that takes *truth* to *falsity* (and vice versa). In intuitionistic logic, according to the Brouwer–Heyting–Kolmogorov interpretation, the negation of a proposition P is the proposition whose proofs are the refutations of P.

An operand of a negation is a **negand**,[3] or **negatum**.[3]

Definition [edit]

Classical negation is an operation on one logical value, typically the value of a proposition, that produces a value of *true* when its operand is false, and a value of *false* when its operand is true. Thus if statement P is true, then $\neg P$ (pronounced "not P") would then be false; and conversely, if $\neg P$ is true, then P would be false.

The truth table of $\neg P$ is as follows:

P	$\neg P$
True	False
False	True

Negation can be defined in terms of other logical operations. For example, $\neg P$ can be defined as $P \rightarrow \bot$ (where $\rightarrow$ is logical consequence and $\bot$ is absolute falsehood). Conversely, one can define $\bot$ as $Q \wedge \neg Q$ for any proposition Q (where $\wedge$ is logical conjunction). The idea here is that any contradiction is false, and while these ideas work in both classical and intuitionistic logic, they do not work in paraconsistent logic, where contradictions are not necessarily false. In classical logic, we also get a further identity, $P \rightarrow Q$ can be defined as $\neg P \vee Q$, where $\vee$ is logical disjunction.

Algebraically, classical negation corresponds to complementation in a Boolean algebra, and intuitionistic negation to pseudocomplementation in a Heyting algebra. These algebras provide a semantics for classical and intuitionistic logic.

Notation [edit]

The negation of a proposition p is notated in different ways, in various contexts of discussion and fields of application. The following table documents some of these variants:

Notation	Plain text	Vocalization
$\neg p$	¬p , 7p[4]	Not *p*
$\sim p$	~p	Not *p*
$-p$	-p	Not *p*
Np		En *p*
p'	p '	*p* prime, *p* complement
$\bar{p}$	p̅	*p* bar, Bar *p*
$!p$	! p	Bang *p* Not *p*

The notation Np is Polish notation.

In set theory, $\setminus$ is also used to indicate 'not in the set of': $U \setminus A$ is the set of all members of U that are not members of A.

Regardless how it is notated or symbolized, the negation $\neg P$ can be read as "it is not the case that P", "not that P", or usually more simply as "not P".

Precedence [edit]

See also: Logical connective § Order of precedence

As a way of reducing the number of necessary parentheses, one may introduce precedence rules: ¬ has higher precedence than ∧, ∧ higher than ∨, and ∨ higher than →. So for example, $P \vee Q \wedge \neg R \rightarrow S$ is short for $(P \vee (Q \wedge (\neg R))) \rightarrow S$.

Here is a table that shows a commonly used precedence of logical operators.[5]

Operator	Precedence
¬	1
∧	2
∨	3
→	4
↔	5

Module-4

Concurrency

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple processor cores on a single chip)

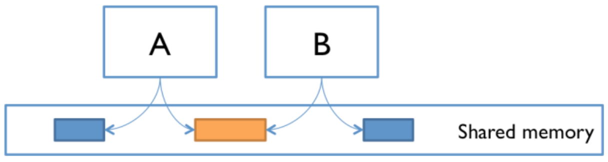
In fact, concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users.
- Mobile apps need to do some of their processing on servers ("in the cloud").
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it.

Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we're getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces.

Two Models for Concurrent Programming

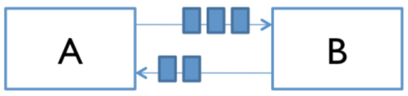
There are two common models for concurrent programming: *shared memory* and *message passing*.



Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.

Other examples of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.
- A and B might be two programs running on the same computer, sharing a common filesystem with files they can read and write.
- A and B might be two threads in the same Java program (we'll explain what a thread is below), sharing the same Java objects.



Message passing. In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling. Examples include:

- A and B might be two computers in a network, communicating by network connections.

- A and B might be a web browser and a web server – A opens a connection to B, asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging client and server.
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like ls | grep typed into a command prompt.

Processes, Threads, Time-slicing

The message-passing and shared-memory models are about how concurrent modules communicate. The concurrent modules themselves come in two different kinds: processes and threads.

Process. A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine's memory.

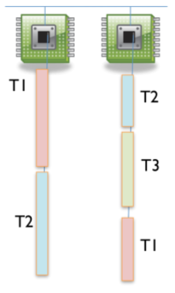
The process abstraction is a *virtual computer*. It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.

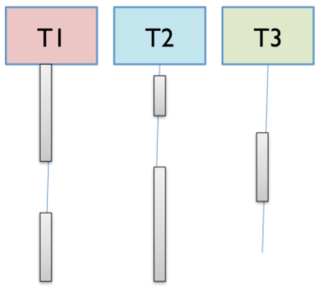
Just like computers connected across a network, processes normally share no memory between them. A process can't access another process's memory or objects at all. Sharing memory between processes is *possible* on most operating system, but it needs special effort. By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the System.out and System.in streams you've used in Java.

Thread. A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place to which it will be necessary to return through.

Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside the virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in process.

Threads are automatically ready for shared memory, because threads share all the memory in the process. It needs special effort to get "thread-local" memory that's private to a single thread. It's also necessary to set up message-passing explicitly, by creating and using queue data structures. We'll talk about how to do that in a future reading.

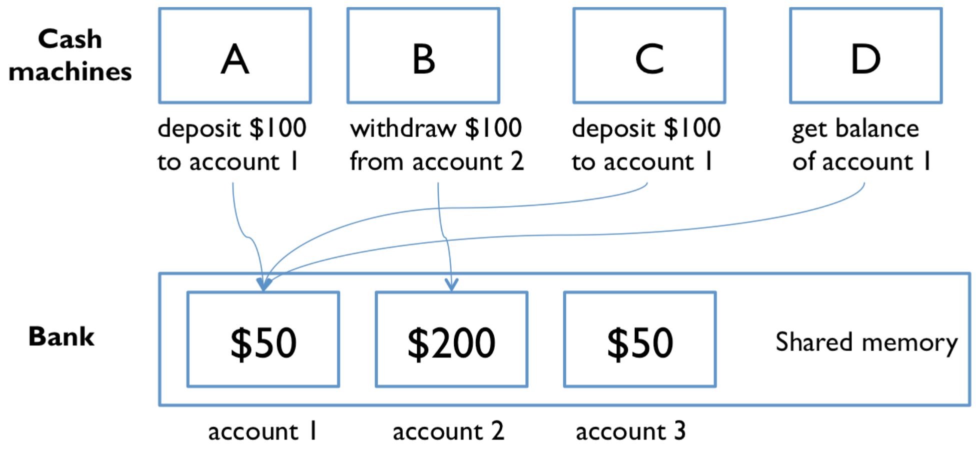




How can I have many concurrent threads with only one or two processors in my computer? When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads. The figure on the right shows how three threads T1, T2, and T3 might be time-sliced on a machine that has only two actual processors. In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3. Thread T2 simply pauses, until its next time slice on the same processor or another processor.

Shared Memory Example

Let’s look at an example of a shared memory system. The point of this example is to show that concurrent programming is hard, because it can have subtle bugs.



Imagine that a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.

To illustrate what can go wrong, let’s simplify the bank down to a single account, with a dollar balance stored in the balance variable, and two operations deposit and withdraw that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;


private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}
```

Inter-thread Communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other. Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

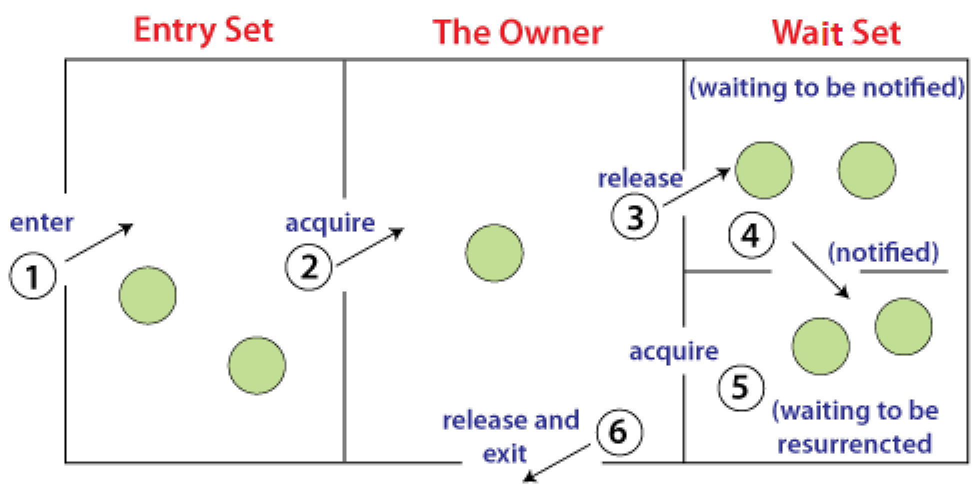
2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
Syntax:
public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.
Syntax:
public final void notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:
Threads enter to acquire lock.
Lock is acquired by on thread.
Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
Now thread is available to acquire lock.After completion of the task, thread releases the lock and exits the monitor state of the object.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
```

```
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

- The synchronization is mainly used to
- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

- Mutual Exclusive
- Synchronized method.
- Synchronized block.
- Static synchronization.

Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

By Using Synchronized Method

By Using Synchronized Block

By Using Static Synchronization

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
class Table{
void printTable(int n){//method not synchronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }
 }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
100
10
200
15
300
```

```
20
400
25
500
```

Java Synchronized Method

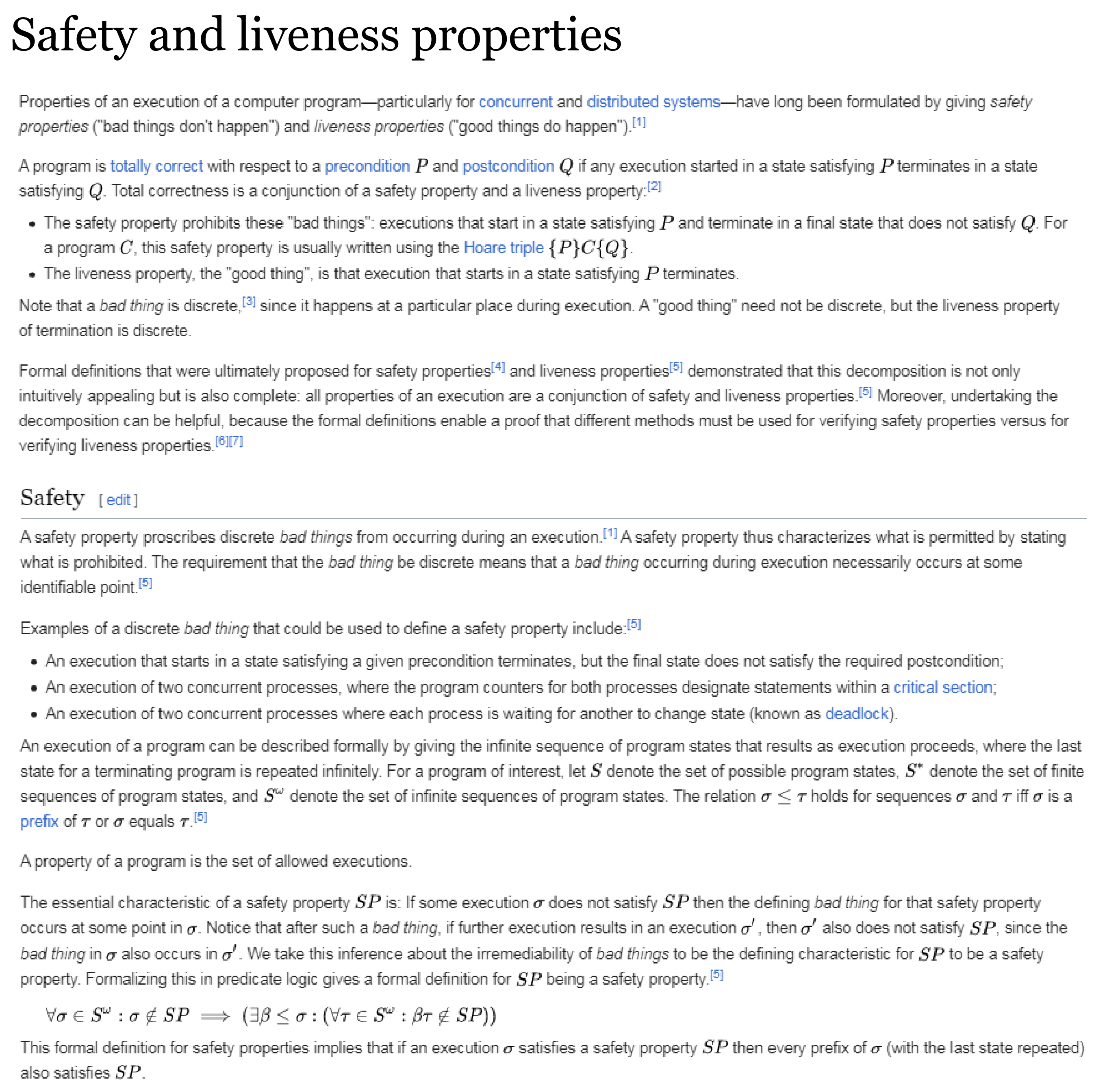
If you declare any method as synchronized, it is known as synchronized method.
Synchronized method is used to lock an object for any shared resource.
When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }
 }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
ADVERTISEMENT
ADVERTISEMENT
          5
            10
            15
            20
            25
            100
            200
            300
            400
            500
```



Safety and liveness properties

Properties of an execution of a computer program—particularly for concurrent and distributed systems—have long been formulated by giving *safety properties* ("bad things don't happen") and *liveness properties* ("good things do happen").[1]

A program is totally correct with respect to a precondition P and postcondition Q if any execution started in a state satisfying P terminates in a state satisfying Q. Total correctness is a conjunction of a safety property and a liveness property:[2]

- The safety property prohibits these "bad things": executions that start in a state satisfying P and terminate in a final state that does not satisfy Q. For a program C, this safety property is usually written using the Hoare triple $\{P\}C\{Q\}$.
- The liveness property, the "good thing", is that execution that starts in a state satisfying P terminates.

Note that a *bad thing* is discrete,[3] since it happens at a particular place during execution. A "good thing" need not be discrete, but the liveness property of termination is discrete.

Formal definitions that were ultimately proposed for safety properties[4] and liveness properties[5] demonstrated that this decomposition is not only intuitively appealing but is also complete: all properties of an execution are a conjunction of safety and liveness properties.[5] Moreover, undertaking the decomposition can be helpful, because the formal definitions enable a proof that different methods must be used for verifying safety properties versus for verifying liveness properties.[6][7]

Safety [edit]

A safety property proscribes discrete *bad things* from occurring during an execution.[1] A safety property thus characterizes what is permitted by stating what is prohibited. The requirement that the *bad thing* be discrete means that a *bad thing* occurring during execution necessarily occurs at some identifiable point.[5]

Examples of a discrete *bad thing* that could be used to define a safety property include:[5]

- An execution that starts in a state satisfying a given precondition terminates, but the final state does not satisfy the required postcondition;
- An execution of two concurrent processes, where the program counters for both processes designate statements within a critical section;
- An execution of two concurrent processes where each process is waiting for another to change state (known as deadlock).

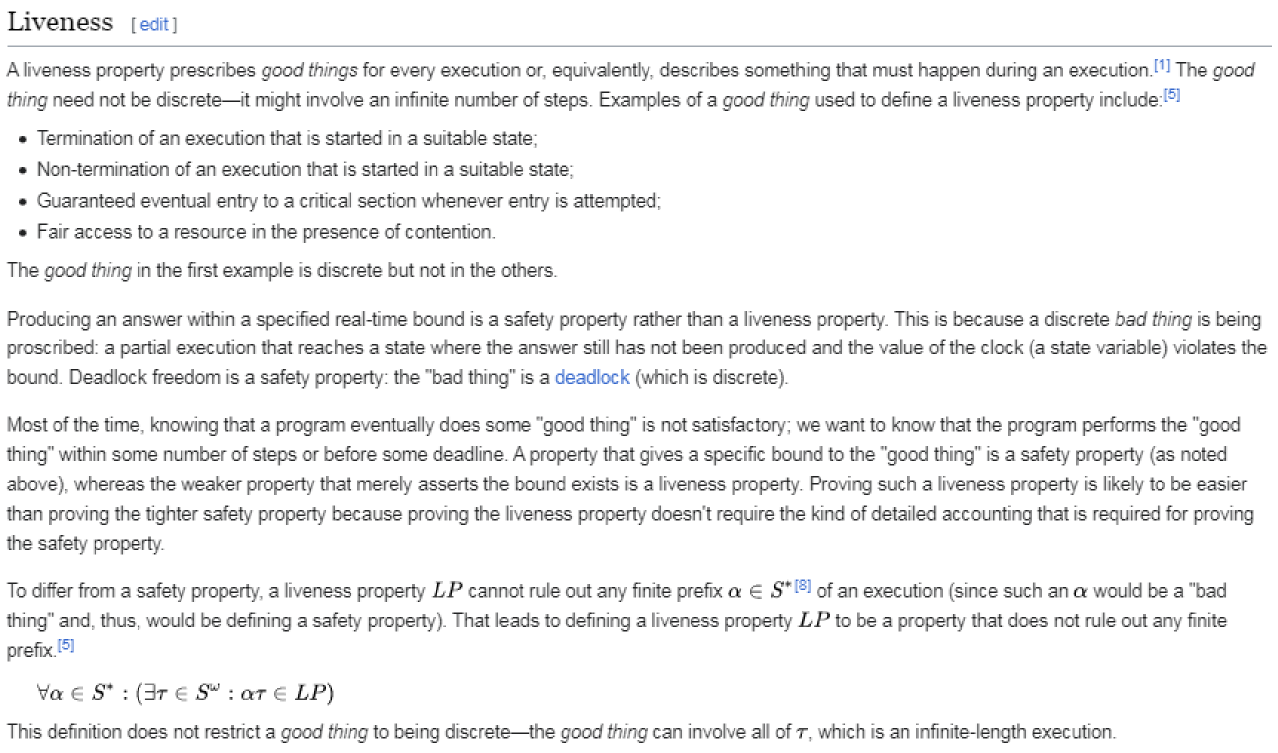
An execution of a program can be described formally by giving the infinite sequence of program states that results as execution proceeds, where the last state for a terminating program is repeated infinitely. For a program of interest, let S denote the set of possible program states, S^* denote the set of finite sequences of program states, and S^ω denote the set of infinite sequences of program states. The relation $\sigma \leq \tau$ holds for sequences σ and τ iff σ is a prefix of τ or σ equals τ.[5]

A property of a program is the set of allowed executions.

The essential characteristic of a safety property SP is: If some execution σ does not satisfy SP then the defining *bad thing* for that safety property occurs at some point in σ. Notice that after such a *bad thing*, if further execution results in an execution σ', then σ' also does not satisfy SP, since the *bad thing* in σ also occurs in σ'. We take this inference about the irremediability of *bad things* to be the defining characteristic for SP to be a safety property. Formalizing this in predicate logic gives a formal definition for SP being a safety property.[5]

$$\forall \sigma \in S^\omega : \sigma \notin SP \implies (\exists \beta \leq \sigma : (\forall \tau \in S^\omega : \beta\tau \notin SP))$$

This formal definition for safety properties implies that if an execution σ satisfies a safety property SP then every prefix of σ (with the last state repeated) also satisfies SP.



Liveness [edit]

A liveness property prescribes *good things* for every execution or, equivalently, describes something that must happen during an execution.[1] The *good thing* need not be discrete—it might involve an infinite number of steps. Examples of a *good thing* used to define a liveness property include:[5]

- Termination of an execution that is started in a suitable state;
- Non-termination of an execution that is started in a suitable state;
- Guaranteed eventual entry to a critical section whenever entry is attempted;
- Fair access to a resource in the presence of contention.

The *good thing* in the first example is discrete but not in the others.

Producing an answer within a specified real-time bound is a safety property rather than a liveness property. This is because a discrete *bad thing* is being proscribed: a partial execution that reaches a state where the answer still has not been produced and the value of the clock (a state variable) violates the bound. Deadlock freedom is a safety property: the "bad thing" is a deadlock (which is discrete).

Most of the time, knowing that a program eventually does some "good thing" is not satisfactory; we want to know that the program performs the "good thing" within some number of steps or before some deadline. A property that gives a specific bound to the "good thing" is a safety property (as noted above), whereas the weaker property that merely asserts the bound exists is a liveness property. Proving such a liveness property is likely to be easier than proving the tighter safety property because proving the liveness property doesn't require the kind of detailed accounting that is required for proving the safety property.

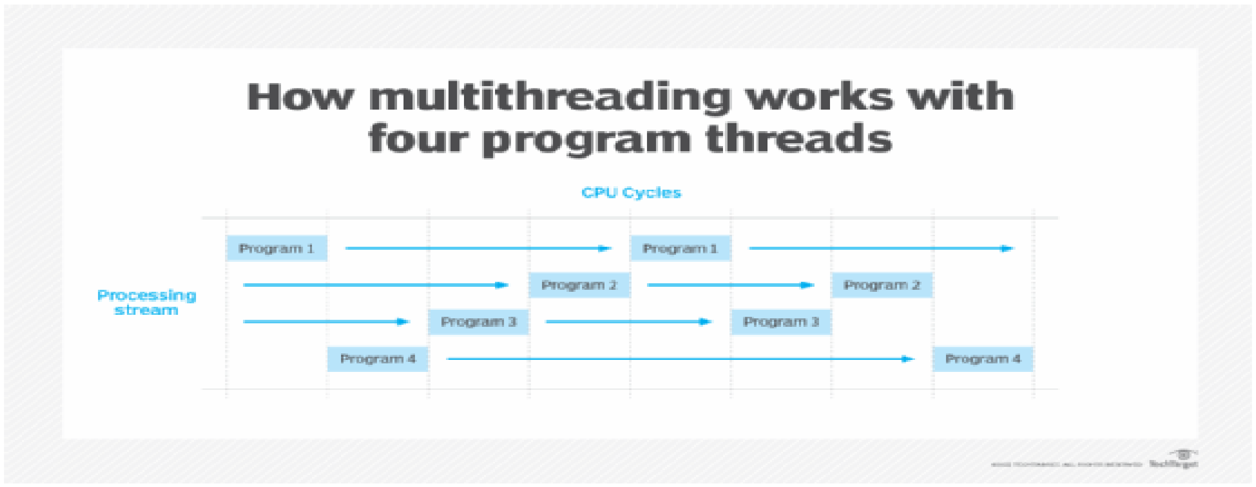
To differ from a safety property, a liveness property LP cannot rule out any finite prefix $\alpha \in S^*$[8] of an execution (since such an α would be a "bad thing" and, thus, would be defining a safety property). That leads to defining a liveness property LP to be a property that does not rule out any finite prefix.[5]

$$\forall \alpha \in S^* : (\exists \tau \in S^\omega : \alpha\tau \in LP)$$

This definition does not restrict a *good thing* to being discrete—the *good thing* can involve all of τ, which is an infinite-length execution.

What is multithreading?

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer. Multithreading can also handle multiple requests from the same user. Each user request for a program or system service is tracked as a thread with a separate identity. As programs work on behalf of the initial thread request and are interrupted by other requests, the work status of the initial request is tracked until the work is completed. In this context, a user can also be another program. Fast central processing unit (CPU) speed and large memory capacities are needed for multithreading. The single processor executes pieces, or threads, of various programs so fast, it appears the computer is handling multiple requests simultaneously.



With multithreading, while the computer system's processor executes one instruction at a time, different threads from multiple programs are executed so fast it appears the programs are executed simultaneously.

How does multithreading work?

The extremely fast processing speeds of today's microprocessors make multithreading possible. Even though the processor executes only one instruction at a time, threads from multiple programs are executed so fast that it appears multiple programs are executing concurrently.

Each CPU cycle executes a single thread that links to all other threads in its stream. This synchronization process occurs so quickly that it appears all the streams are executing at the same time. This can be described as a multithreaded program, as it can execute many threads while processing.

Each thread contains information about how it relates to the overall program. While in the asynchronous processing stream, some threads are executed while others wait for their turn. Multithreading requires programmers to pay careful attention to prevent race conditions and deadlock.

An example of multithreading

Multithreading is used in many different contexts. One example occurs when data is entered into a spreadsheet and used for a real-time application.

When working on a spreadsheet, a user enters data into a cell, and the following may happen:

- column widths may be adjusted;
- repeating cell elements may be replicated; and
- spreadsheets may be saved multiple times as the file is further developed.

Each activity occurs because multiple threads are generated and processed for each activity without slowing down the overall spreadsheet application operation.

Example:-

```
class ABC implements Runnable
{
public void run()
{
```

```
// try-catch block
try
{
// moving thread t2 to the state timed waiting
Thread.sleep(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}


System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -
"+ ThreadState.t1.getState());

// try-catch block
try
{
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}

// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable
{
public static Thread t1;
public static ThreadState obj;

// main method
public static void main(String argvs[])
{
// creating an object of the class ThreadState
obj = new ThreadState();
t1 = new Thread(obj);

// thread t1 is spawned
// The thread t1 is currently in the NEW state.
System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on
// the thread t1
t1.start();

// thread t1 is moved to the Runnable state
System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());
}

public void run()
{
```

```java
ABC myObj = new ABC();
Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
t2.start();

// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());

// try-catch block for the smooth flow of the  program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getState() );

// try-catch block for the smooth flow of the  program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());
}

}
```

Output:

```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING
The state of thread t2 when it has completed it's execution - TERMINATED
```