**AALIM MUHAMMED SALEGH COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER APPLICATION**

**SEMESTER- I**

**MC5102 – PROBLEM SOLVING AND PROGRAMMING**

**UNIT – II NOTES**

**SYLLABUS:**

**UNIT II PROGRAMMING, ALGORITHMS AND FLOWCHARTS**

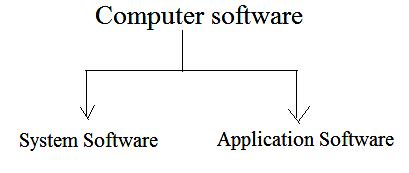
Programs and Programming – building blocks for simple programs -Programming life cycle phases – pseudo code representation – flow charts - Algorithm - Programming Languages - compiler – Interpreter, Loader and Linker - Program execution – Classification of Programming Language - Structured Programming Concept.

**Program and programming**

* A computer can neither think nor judge on its own.
* A computer independently analyzes a given data and find a solution on its own.
* A program is a set of logically related instructions that is arranged in a sequence and guides the computer in solving a problem.
* A program is a set of instructions to perform a task.
* Process of writing a program is called Programming.
* If the system is not correctly programmed, it delivers information results that cannot be used.

Program acquired by

* Purchase an existing program – Packaged software
* Prepare a new program from scratch – Customized software.



**[1] System software**

**Language Translator:**

* Transforms a computer program written by the u ser into a form that can be understood by the machine.

1. ***Examples of Language translators:***
2. COBOL 3
3. Turbo C
4. Borland C etc….

**Operating system:**

* It manages the computer resources effectively, take care of scheduling multiple jobs for execution, and manages the flow of data and instructions between the input/output units and the main memory.
* OS have been developed and gone several revisions and modifications to achieve better utilization of computer resources.
* Advances in computer hardware have helped in the development of more efficient OS.

**Utilities:**

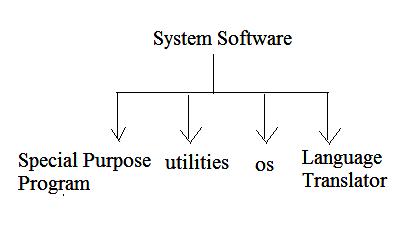
* Utility programs are those that may be requested by application programs many times during the execution phase.

**Example (**Utilities**):**

* Sort/merge for sorting la rge volumes of data and merging them into a sorted list.
* Transfer programs for transforming data content from one medium to another disk to tape, tape to disk etc……

**Special purpose software:**

* Extend the capability of operating systems to provide specialize services to application programs.



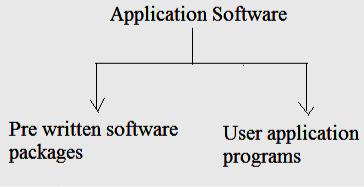
**[2] Application Software:**

* Enable the compute r to solve a specific data processing task.
* Two categories of application software
  + 1. Pre written software packages
    2. User application programs.
* Application programs are written to meet the exact requirement.
* User application program may be written using one of these packages or programming languages.

**Important categories of software packages available are**

1. Database management software
2. Spreadsheet software
3. Word processing, Desktop publishing and Presentation soft ware.
4. Multimedia software
5. Data communication software
6. Statistical an d operational research software

***Categories of a Application software***



**Building Blocks for simple programs**

Building blocks for programs are

* + 1. Sequential control
    2. Selection control
    3. Repetition

These are easy to construct any algorithm.

1. **Sequential Control**
   1. Steps of an algorithm are carried out in a sequential manner where each step is executed exactly once.
   2. Example: Conversion of Fahrenheit into Celsius

**Algorithm**

* + 1. Read the temperature in Fahrenheit
    2. Apply conversion formula
    3. Display result in degree Celsius.

Description of an algorithm done using pseudo code.

**Pseudo code:**

* It is a mixture of English (human language), symbols and selected features commonly used in programming languages.

**Read degree- farenheit**

**Degree-celsius=(5/9)\*(degree=farenheit-32)**

**Display degree-celcius**

* Describing an algorithm can also be done by graphical representation.
* Graphical representation of an algorithm is a flowchart.

**Selection control:**

1. Only one of a number of alternative steps is executed.
2. Eg Student result generation

**Read grade**

**If(grade>=95)**

**Then destination**

**Else**

**Try to get it**

1. With if only used one condition checked.
2. If else 2 condition checked.
3. If else if else then more than 2 conditions are checked.
4. For checking more than 2 conditions we can use switch case.
5. Example: Read result Case(result>=90) Print ‘a’ Case(result>=80) Print ‘b’ Case(result>=70) Print ‘c’ Case(result>=60) Print ‘d’

Case (result<60) Print ‘f’

* 1. This has more alternatives. According to the alternatives the structures are chosen.

1. **Repetition:**
   1. One or more steps are performed repeatedly under a condition.
   2. Example: Compute the average of 10 numbers.

**Ex 1:**

Total=0

Average=0

For 1 to 10

Read number

Total=total+number

End for

Average=total/10

**Ex 2:**

**For 1 to 10**

**Print “hai”**

**Ex 3:**

Read 10 numbers

While (total<s) do

Read number

Total=total+number

End do

**Programming life cycle phase**

* + Programming is a problem solving activity.
  + Good problem solver has potential to become a good performance.
  + Business students solve problems with systems approach.
  + Engineering and science students use engineering and scientific method.
  + Programmers use the software development method.

**Software development method:**

* Specify the problem requirement
* Analyze the problem.
* Design the algorithm to solve the problem.
* Implement the algorithm
* Test and verify the completed program.
* Maintain and update the program.

1. **Specify the problem requirement:**
   * Forces to state the problem clearly and unambiguously.
   * This gain a clear understanding and requirement for the solution.
   * Objective is to eliminate unimportant aspects.
   * More information to be calculated.
2. **Analysis:**
   1. Involves identifying the problem
      1. inputs
      2. Outputs
      3. Constraints on the solution
   2. Determine the required format in which output is displayed.
   3. Develop a list of problem variables and their relationship.
   4. It can be expressed as formulas.
   5. Problem statement-read very carefully
   6. Underline the phrases in the problem statement that identify input and output.

***Compute and display the total cost of apples given the number of pounds of apples purchased and the cost per pound of apples.***

**Problem Input**: Quantity of apples purchased (in pounds)

Cost per pound of apples (in dollars per pound)

**Problem output:** Total cost of apples (in dollars)

**Formula designed:** General formula

Total cost=Unit cost ×Number of units.

1. The processing of modeling a problem by extracting the essential variables and their relationships is called abstraction.

**[3]Design:-**

* To solve a problem, list of steps called algorithm are designed.
* No need for every detail of problem in the beginning.
* Top-down design is used.
* List the major steps or sub-problems that are needed to solve the problem.
* Problem can be broken to sub problems called divide and conquer method.
* Generally any problem has the following sub-problems.

1. Get the data.
2. Perform the computation.
   1. Display the result.

* Attack the each problem individually.
* Algorithm refinement is used to perform breaking of large steps in to more detailed list of steps.
* To desk check algorithm, carefully perform each algorithm step and verify the task of algorithm like a computer.
* To save the time and effort to locate algorithm errors early in the problem solving process.

**[4]Implementation:-**

* Involving in writing the program.
* It converts each algorithm step into one or more statements in a programming language.

**Example:-**

**int x,y;**

**scanf(“%d”,&x);**

**scanf(“%d”,&y);**

**X=x+y;**

**printf(“The Result is %d”,X);**

**[5]Testing:-**

1. To test the completed program to verify that it works as desired.
2. Don’t rely on one test case.
3. Test the program using different set of data for every situation.

**[6]Maintenance:-**

* 1. Maintaining and updating the program involves modifying the program to resolve previously undetected errors and to keep it up to date.
  2. Creating a program that is to read, understand and maintain is a discipline approach.
  3. Follow accepted program style guidelines.

**pseudo code Representation**

* + **Pseudocode** is a simple way of writing programming code in English. Pseudocode is notan actual programming language.
  + It uses short phrases to write code for programs before one actually create it in a specific language.
  + An outline of a program, written in a form that can easily be converted into real programming statements.
  + For example, the pseudocode for finding given number odd or even.

**pseudo code for finding whether a given number is even or odd.**

1. Input number 'X' from user
2. Divide the X by 2 and store its remainder As 'R'
3. if R is 0 then print 'X' is an even number
4. if R is not 0 then print 'X' is an odd number.
5. Exit

**C Program for finding whether a given number is even or odd.**

void main()

{

int x,r;

clrscr();

printf("Enter a number ");

scanf("%d", x);

r = x/2;

if (r = 0)

printf("Given number is Even")

else

printf("Given number is Odd.")

getch();

}

**Flowchart**

* Flowchart is a diagrammatic representation of an algorithm.
* Flowchart is very helpful in writing program and explaining program to others.

**Symbols Used In Flowchart**

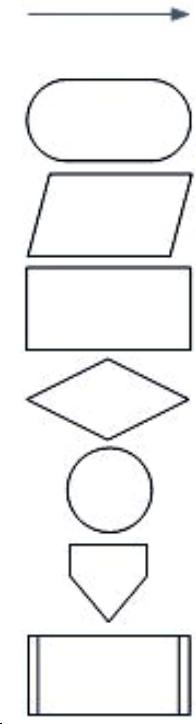
Different symbols are used for di fferent states in flowchart.

For example:

Input/Output and decision making have different symbols.

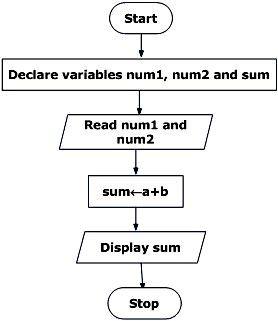
The table below describes all the symbols that are used in making flowchart

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Purpose** | **Description** |
|  | Flow line | Used to indicate the flow of logic by conn ecting |
|  |  | symbols. |
|  | Terminal(Stop/Start) | Used to represent start and end of flowch art. |
|  |  |  |
|  | Input/Output | Used for input and output operation. |
|  |  |  |
|  | Processing | Used for arithmetic operations and data-m anipulations. |
|  |  |  |
|  | Desicion | Used to represent the operation in which t here are two |
|  |  | alternatives, true and false. |
|  |  |  |
|  | On-page Connector | Used to join different flow line |
|  |  |  |
|  | Off-page Connector | Used to connect flowchart portion on different page. |
|  |  |  |
|  | Predefined | Used to represent a group of statements p erforming one |
|  | Process/Function | processing task. |
|  |  |  |

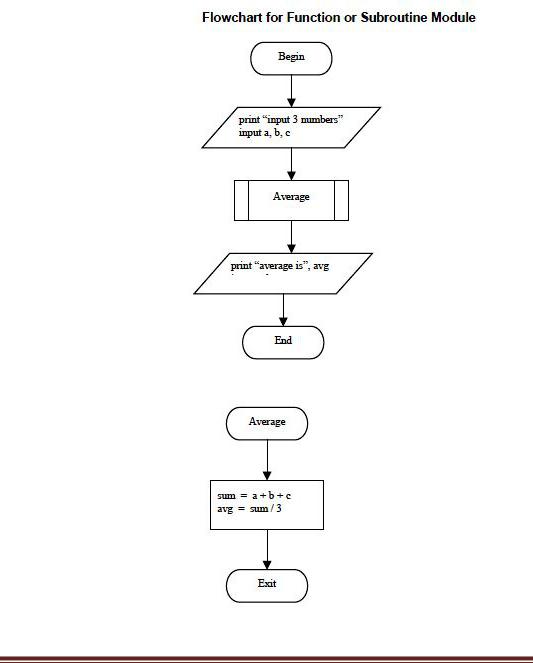


**Examples of flowcharts in programming**

**Flowchart to add two numbers**



Though, flowchart are useful in efficient coding, debugging and analysis of a program, drawing flowchart in very complicated in case of complex programs and often ignored.



**Algorithm**

* In programming, algorithm is the set of well defined instruction in sequence to solve a program.
* An algorithm should always have a clear stopping point.

***Qualities of a good algorithm***

* Inputs and outputs should be defined precisely.
* Each steps in algorithm should be clear and unambiguous.
* Algorithm should be most effective among many different ways to solve a problem.
* An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

***Examples Of Algorithms In Programming***

**Algorithm to add two numbers entered by user.**

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

**Algorithm to find the largest among three different numbers entered by user.**

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a>b

If a>c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b>c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

**Algorithm to find all roots of a quadratic equation ax2+bx+c=0.**

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminate

D←b2-4ac

Step 4: If D≥0

r1←(-b+√D)/2a

r2←(-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp←b/2a

ip←√(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

**An algorithm to find the factorial of a number entered by user.**

Step 1: Start

Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables

factorial←1

i←1

Step 4: Read value of n

Step 5: Repeats the steps until i=n

5.1: factorial←factorial\*i

5.2: i←i+1

Step 6: Display factorial

Step 7: Stop

**An algorithm to check whether a number entered by user is prime or not.**

Step 1: Start

Step 2: Declare variables n,i,flag.

Step 3: Initialize variables

flag←1

i←2

Step 4: Read n from user.

Step 5: Repeats the steps until i<(n/2)

5.1 If remainder of n÷i equals 0

flag←0

Go to step 6

5.2 i←i+1

Step 6: If flag=0

Display n is not prime

else

Display n is prime

Step 7: Stop

**Algorithm to find the Fibonacci series till term≤1000.**

Step 1: Start

Step 2: Declare variables first\_term,second\_term and temp.

Step 3: Initialize variables first\_term←0 second\_term←1

Step 4: Display first\_term and second\_term

Step 5: Repeats the steps until second\_term≤1000

5.1: temp←second\_term

5.2: second\_term←second\_term+first term

5.3: first\_term←temp

5.4: Display second\_term

Step 6: Stop

Algorithm is not the computer code. Algorithm is just the instructions which give clear idea to you idea to write the computer code.

**Programming Languages**

***The different generations of languages***

* There are currently five generations of computer programming languages.
* In each generation, the languages syntax has become easier to understand and more human-readable.

**First generation languages (abbreviated as 1GL)**

Represent the very early, primitive computer languages that consisted entirely of 1's and 0's - the actual language that the computer understands (machine language).

**Second generation languages (2GL)**

* Represent a step up from the first generation languages.
* Allow for the use of symbolic names instead of just numbers.
* Second generation languages are known as assembly languages.
* Code written in an assembly language is converted into machine language (1GL).

**Third generation languages (3GL)**

* With the languages introduced by the third generation of computer programming, words and commands (instead of just symbols and numbers) were being used.
* These languages therefore, had syntax that was much easier to understand.
* Third generation languages are known as "high level languages" and include C, C++, Java, and Javascript, among others.

**Fourth generation languages (4GL)**

* The syntax used in 4GL is very close to human language, an improvement from the previous generation of languages.
* 4GL languages are typically used to access databases and include SQL and ColdFusion, among others.

**Fifth generation languages (5GL)**

* Fifth generation languages are currently being used for neural networks.
* A nueral network is a form of artifical intelligence that attempts **to imitate how the** **human mind works.**
* Human makes the computer to think.

**Compiler**

1. "A compiler translates the high-level source programs into target programs in machine languages for the specific hardware.”
2. Once the target program is generated, the user can execute the program.
3. A compiler reads analyses and translates code into either an object file or a list of error messages.

**A compiler for a language generally has several different stages as it processes the input.**

**Preprocessing**

* During the preprocessing stage, comments, macros, and directives are processed. Comments are removed from the source file. This greatly simplifies the later stages.
* If the language supports macros, the macros are replaced with the equivalent text.
* For example, C and C++ support macros using the #define directive. So if a macro were defined for pi as: #define PI 3.1415927
* Any time the preprocessor encountered the word PI, it would replace PI with 3.1415927 and process the resulting text.
* The preprocessor may also replace special strings with other characters. In C and C++, the preprocessor recognizes the \ character as an escape code, and will replace the escape sequence with a special character. For example \t is the escape code for a tab, so \t would be replaced at this stage with a tab character.

**Lexical analysis**

1. It is the process of breaking down the source files into key words, constants, identifiers, operators and other simple tokens. A token is the smallest piece of text that the language defines.

**C tokens:**

1. C tokens are the basic buildings blocks in C language which are constructed together to write a C program.

1. Each and every smallest individual units in a C program are known as C tokens.
2. C tokens are of six types. They are,

|  |  |  |
| --- | --- | --- |
| § | Keywords | (eg: int, while), |
| § | Identifiers | (eg: main, total), |
| § | Constants | (eg: 10, 20), |
| § | Strings | (eg: “total”, “hello”), |

1. Special symbols (eg: (), {}),

§ Operators (eg: +, /,-,\*)

**Syntactical analysis**

* It is the process of combining the tokens into well-formed expressions, statements, and programs.
* Each language has specific rules about the structure of a program--called the grammar or syntax. Just like English grammar, it specifies how things may be put together. In English, a simple sentence is: subject, verb, predicate.
* In C or C++ an if statement is: if ( expression ) statement
* The syntactical analysis checks that the syntax is correct, but doesn't enforce that it makes sense.
* In English, a subject could be: Pants, the verb: are, the predicate: a kind of car. This would yield: Pants are a kind of car.
* Which is a sentence, but doesn't make much sense.

1. In C or C++, a constant can be used in an expression: so the expression:

**float x = "This is red"++**

1. Is syntactically valid, but doesn't make sense because a float number cannot have string assigned to it, and a string cannot be incremented.

**Semantic analysis**

1. It is the process of examining the types and values of the statements used to make sure they make sense.
2. During the semantic analysis, the types, values, and other required information about statements are recorded, checked, and transformed as appropriate to make sure the program makes sense.
3. For C/C++ in the line: **float x = "This is red"++**
4. The semantic analysis would reveal the types do not match and cannot be made to match, so the statement would be rejected and an error reported.
5. While in the statement: **float y = 5 + 3.0;**

* The semantically analysis would reveal that 5 is an integer, and 3.0 is a double, and also that the rules for the language allow 5 to be converted to a double, so the addition could be done, so the expression would then be transformed to a double and the addition performed.
* Then, the compiler would recognize y as a float, and perform another conversion from the double 8.0 to a float and process the assignment.

**Intermediate code generation**

* Depending on the compiler, this step may be skipped, and instead the program may be translated directly into the target language (usually machine object code).
* If this step is implemented, the compiler designers also design a machine independent language of their own that is close to machine language and easily translated into machine language for any number of different computers.
* The purpose of this step is to allow the compiler writers to support different target computers and different languages with a minimum of effort.
* The part of the compiler which deals with processing the source files, analyzing the language and generating the intermediate code is called the front end, while the process of optimizing and converting the intermediate code into the target language is called the back end.

**Code optimization**

* During this process the code generated is analyzed and improved for efficiency. The compiler analyzes the code to see if improvements can be made to the intermediate code that couldn't be made earlier.
* For example, some languages like Pascal do not allow pointers, while all machine languages do. When accessing arrays, it is more efficient to use pointers, so the code optimizer may detect this case and internally use pointers.

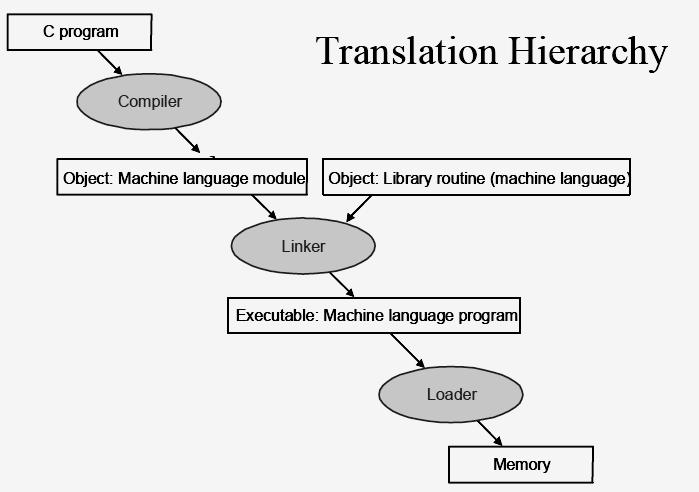
**7. Code generation**

Finally, after the intermediate code has been generated and optimized, the compiler will generate code for the specific target language. Almost always this is machine code for a particular target machine.

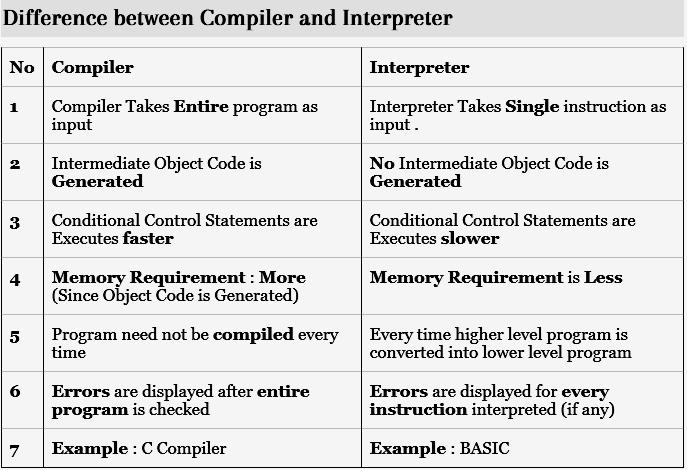
* Also, it us usually not the final machine code, but is instead object code, which contains all the instructions, but not all of the final memory addresses have been determined.
* A subsequent program, called a linker is used to combine several different object code files into the final executable program.

**Interpreter**

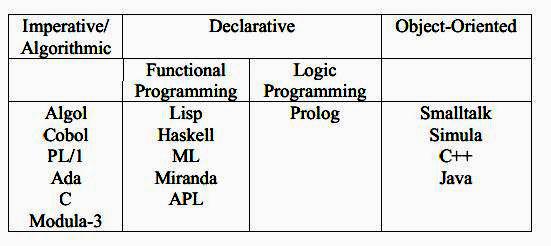
* An **Interprete**r reads, and translates code line by line.
* A **linker** combines one or more object files and possible some library c ode into some executable, some library or a list of error messages.
* A **loader** reads the executable code into memory does some address translation and tries to run the program resulting in a running program or an error message (or both).



* During **program execution**, constructs and statements are executed in a prescribed order.
* Execution in computer and software engineering is the process by which a computer or a virtual machine performs the instructions of a computer program. The instructions in the program trigger sequences of simple actions on the executing machine. Those actions produce effects according to the semantics of the instructions in the program.
* Programs for a computer may execute in a batch process without human interaction, or a user may type commands in an interactive session of an interpreter. I n this case the "commands" are simply programs, whose execution is chained together.



**Classification of Programming languages**



**Imperative Programming**

* Control flow in imperative programming is *explicit*: commands show *how* the compuation takes place, ste p by step.
* Each step affects the global **state** of the computation.

result = []

i = 0

start:

numPeople = length(people)

if i >= numPeople goto end

p = people[i]

nameLength = length(p.name)

if nameLength <= 5 goto next

upperName = toUpper(p.name)

addToList(result, upperName)

next:

i = i + 1

goto start

end:

return sort(result)

**Structured Programming**

1. Structured programming is a kind of imperative programming where the control flow is defined by nested loops, co nditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

result = [];

**for** i = 0; i < length(people); i++ {

p = people[i];

**if** length(p.name)) > 5 {

addToList(result, toUpper(p.name));

}

}

**return** sort(result);

***Early languages emphasizing structured programming:***

Algol 60, PL/I, Algol 68, Pascal, C, Ada 83, Modula, Modula-2. Structured programming as a discipline is sometimes though to have been started by a famous letter by Edsger Dijkstra entitled Go to Statement Considered Harmful.

**Object Oriented Programming**

* OOP is based on the sending of messages to objects. Objects respond to messages by performing operations. Messages can have arguments, so "sending messages" looks a lot like calling subroutines.
* A society of objects, each with their own "local memory" and own set of operations has a different feel than the "monolithic processor and single shared memory" feel of non object oriented languages.

result = []

**for** p **in** people {

**if** p.name.length > 5 {

result.add(p.name.toUpper);

}

}

**return** result.sort;

* The first object oriented language was Simula-67; Smalltalk followed soon after as the first "pure" object-oriented language.
* Many languages designed from the 1980s to the present have been object-oriented, notably C++, CLOS (object system of Common Lisp), Eiffel, Modula-3, Ada 95, Java, C#, Ruby.

**Declarative Programming**

1. Control flow in declarative programming is *implicit*: the programmer states only *what* the result should look like, **not** how to obtain it.

**select** upper(name)

**from** people

**where** length(name) > 5

**order by** name

1. No loops, no assignments, etc. Whatever engine that interprets this code is just supposed go get the desired information, and can use whatever approach it wants. (The logic and relational paradigms are generally declarative as well.)

**Functional Programming**

In functional programming control flow is expressed by combining function calls, rather than by assigning values to variables.

**let**(

f, **fun**(

people,

**if**(equals(people,emptylist),

emptylist,

**if**(greater(length(name(head(people))),5),

append(to\_upper(name(head(people))), f(tail(people))),

f(tail(people))))),

sort(f(people)))

Of course, there's usually syntactic sugar

**let**

**fun** f[] = []

| f (p :: ps) =

**if** p.name.length() >5 **then** p.name.to\_upper()::(f ps)

**else** (f ps)

**in**

sort(f(people))

The real power of this paradigm comes from passing functions to functions (and returning functions from functions).

sort(

filter((λs. s.length() > 5),

map((λp. p.name.to\_upper()), people)

**Logic and Constraint Programming**

* Logic and constraint programming are two paradigms in which programs are built by setting up relations that specify **facts** and inference **rules**, and asking whether or not something is true (i.e. specifying a **goal**.)
* Unification and backtracking to find solutions (i.e. satisfy goals) takes place automatically.
* Languages that emphasize this paradigm: Prolog, GHC, Parlog, Vulcan, Polka, Mercury, Fnil.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*