Introduction To Programming

Fabio Cannizzo

Objectives

- introduce fundamental concepts of programming (some overlap with C++ course here)
- introduce Matlab

Introduction

- Difficult to talk about programming not in the context of a specific language
- Every language has its own peculiarities, and there are many languages
- There are some abstract concepts which find applicability in most languages

What is a Computer?

- A device which performs computations of some kind, with the purpose of generating the solution of a **well defined** problem in correspondence of some **given inputs**

Algorithm

- The detailed set of instruction which performs a task
- Recipe for cooking pasta:
  - Put the water in the pot
  - Add salt
  - Wait until the water boils
  - Put the pasta in
  - Wait 5 minutes
  - Pasta is ready
- How detailed was that? Did we remember to light up the fire? A computer is not smart!

Algorithm

- Algorithm
  - A well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time
- Unambiguous operation
  - An operation that can be understood and carried out directly by the computing agent without needing to be further simplified or explained
- Computing agent
  - The machine, robot, person, or thing carrying out the steps of the algorithm
  - Does not need to understand the concepts or ideas underlying the solution

Languages

- Natural languages are the languages that people speak. Examples of natural languages are English, Italian and German.
  - Natural languages evolved naturally, with usage by people. They are not designed and fixed by someone, even if often people try to impose some order on them.
- Formal languages are languages that are designed by people for specific goals. Examples are:
  - Math language: a formal language used for denoting relationships among numbers and symbols $y = \sin(x)/2$
  - Programming language: a formal language designed to express computations

Languages

- The main difference between formal and natural languages is about ambiguity.
- Natural languages are full of ambiguity.
- Formal languages are designed to be unambiguous: any statement has exactly one meaning.
- Formal languages tend to be much more concise.

Programming Languages

- There are hundreds programming languages, which can be categorized in many different ways based on their characteristics, for example:
  - General Purpose vs Domain Specific
  - Interpreted / Compiled / Partially Compiled
  - Low Level vs High Level
  - Paradigm: Imperative, Functional Programming, Object Oriented
  - Strongly Typed vs Loosely Typed
  - …

General Purpose vs Domain Specific

- Languages can be designed for general programming or for a specific task
- The intended use defines the primitives and the syntax available
- If I were to design a language specifically designed to control vocally a phone, probably this would only include the primitives
  - dial *name*, hang up
- We will use a general purpose language

High Level vs Low Level

- Think about taking a taxi: what’s your style?

1. Bring me to the airport
2. Bring me to the airport via East Coast Road
3. Turn left here, then right, then left, ….then we arrive at the airport

High Level vs Low Level

- Low level: the closer a language it is to machine language
  - Very verbose to describe even the simplest of the operations.
  - Programmer <u>can</u> and <u>must</u> exercise a very detailed control on the sequence of operations to execute
  - Long source code, difficult to read/maintain
  - High degree of optimization possible
  - Very slow development
  - Require explicit control of resource management (e.g. memory allocation)
- High level: closer to natural language
  - Remove control on details of how macro operations will be executed
  - Hide burden of resource management
  - Allow to express complex operations in easy and quick way
  - High productivity
  - Compact source code, easy to maintain
  - Less control on optimization

Making Less Painful to Work with a Low Level language

- So an high level language take away the ability to optimize at low level, in exchange for higher productivity, maintainability, harder to make mistakes, less need to debug, focus on task at hand rather than on low level details
- Use of libraries can add to a low-level language the advantages of a high-level one (e.g. STL)

High Level vs Low Level

```
function c=vecSum(a,b)
   c = a+b;
endfunction
```

← so simple that it is not worth creating a procedure!

```
void vecSum( double dest*, const double *src1, const double *src2,
             size_t n )
{
   while( n > 0 ) {
      n=n-1;
      dest[n] = src1[n] + src2[n];
   }
}
```

What does this procedure rely on?

- After we build a library, the effort would be almost comparable in the two languages
- How would this look like in assembler?

Interpreted vs Compiled

- Compiled: program is translated into binary code by another program, called compiler. Translation often happens in stages.
- Interpreted: programs are not translated into binary code. The source code is interpreted on the fly (at runtime) by another program, called “interpreter”
- Pseudo-interpreted: it is something in between the two. The program is compiled into pseudo-code, which is not directly executable on a machine, but requires another a virtual machine to be interpreted and executed. This is used typically for portability.

What is the Best Language?

- No language is better than another.
- Some languages are better suited to certain task than others.
- Once the concepts are understood, it is not difficult to learn a new language

Why Matlab?

- Matlab can be used as an interactive calculator and also as a programming environment.
- Powerful mathematical library.
- Graphing tools are easily accessible.
- The language syntax is easy and not very fussy (but this is dangerous because it leads to sloppy programming). Great for prototyping
- It is an interpreted language meaning that no compiling is needed. But this makes it a bit slow.

Traits

- Imperative
- Allow some elementary function manipulation
- Interpreted
- Very high level
- Loosely typed
- Dynamic types
- Suitable for numerical prototyping

Matlab as a Calculator

- using Matlab as a calculator
- basic operators and precedences (+,-,/,*,^)
- number vs string
- comparison operator (==,>,<,~=,>=, <=)
- boolean types (matlab uses 0 and 1)
- boolean (logical) operators (&, &&, |, ||) (what do they mean?)
- help: let’s try “help &&”
- short boolean evaluation
- (ans)

Command Line

- You can cycle through previous commands using up and down arrow keys
- Take the time to read the “getting started with matlab” tutorials available online

Elements of a Language

- Literals
- Variables
- Operators
- Control flow
- Input/Output
- Subroutines
- Delimiters

Literals

- Literals are fixed values in source code (e.g. in x=3, 3 is a literal)
- They have a type
- Some main value types are
  - strings
  - numbers
  - boolean (often a sub-type of *number*)
- Numbers have subtypes (e.g.):
  - Integer
  - Real
- What distinguish one type from another is the way in which it is stored in memory: 3 is not the same as “3”
- Operations are characterized by the value on which they operate

Variables

- A variable is a mnemonic label associated with a particular piece of data inside a program and is characterized by the following properties:
  - **Name**
    - Necessary in order to identify the variable.
    - Different languages have different rules on variables names (e.g. minimum or maximum length, characters allowed, case sensitive)
    - Within allowed rules, It is totally free and up to you to choose a certain name for a variable.
  - **Type**
    - specifies the type of the data that the variable can store; for example: a variable of type String can store a reference to a string.
  - **Value**
    - the data denoted by the variable at a certain point during the execution of the program

Variables in Matlab

- ans
- Variable assignments
- Type-system (double, single, boolean, string, matrix)
- who, whos, clear
- No need for declaration
- Matrices
  - create (square bracket operator)
  - indexation (:, array as an index)
  - extraction
  - assignments to sub-matrices
  - built-ins (zeros, ones, eye)
  - ":" for vector generation
  - size, rows, columns
  - transpose
  - vectorial operators (.*, .^)

Strings in Matlab

- Strings can be declared either with quotes or double quotes

```
>> 'this is a string'
```

- There are 2 ways to concatenate strings

```
>> strcat('this is', ' a string')
>> ['a string', ' is ', 'a matrix of characters']
```

- Strings and number are different types. To mix string and numbers, numbers must be converted to string using num2str(number)

```
>> ['convert number 3 to string: ', num2str(3)]
```

Structures

- A struct is a record, i.e. a set of distinct labels associated with values potentially of different types

```
>> s=struct('a',1,'b','sss');
>> s.a
>> s.b
```

-

Cell

- A cell is a matrix of variables, potentially of different types, without labels, indexed via position in the matrix

```
>> x=cell(2);
>> x{1}='hello';
>> x{2}=[1 2; 3 4];
```

Variable Types

- Types are associated with the content of a variable
- They can be static or dynamic
- They can be primitive or derived
- When they are static, there may be need to declare them explicitly, or the compiler may be able to infer them to a certain extent
- In strongly typed languages the compiler has more chances to be able to help you catch programming errors

Scope

- Scope is the context within a computer program in which a variable name or other identifier is visible and can be used, or within which a declaration has effect.
- Outside of the scope of a variable name, the variable's value may still be stored, and may even be accessible in some way, but the name does not refer to it; that is, the name is not bound to the variable's storage.

Common Scope Rules

- The following scope rules are the most common available an appear in most languages:
- Global scope: the identifier is visible from every part of the program
- Function scope: the identifier is visible within the function were it is declared
- Block scope: the identifier is visible within the block of code (e.g. a loop)

Scope in Matlab

- Any variable declared on the command line has local scope
- Any variable declared in a function has local scope for the function where it is used
- Parameters are passed to function by value, i.e. they are copied (not so inefficient though)

Keywords

- Special words which are reserved for internal use in the language
  - type names (e.g. ‘int’)
  - control flow instructions (e.g. ‘while’)
- Keywords cannot be use for variable names or function names

Delimiters

- Define begin and end of a *something*
- They are typically parenthesis of special keyword
  - The arguments of a function

    ```
    exp( 3 )
    ```
  - the indices of an array

    ```
    x(3)
    ```
  - Creation of a matrix

    [1 2 3]
  - Change operator precedence: (2+3)*4

Sub-Routines (Functions)

- A subroutine, also termed procedure, function, routine, method, or subprogram, is a part of source code within a larger computer program that performs a specific task and is relatively independent of the remaining code.
- As the name subprogram suggests, a subroutine behaves in much the same way as a computer program that is used as one step in a larger program or another subprogram.
- A subroutine is often coded so that it can be started (called) several times and/or from several places during one execution of the program, including from other subroutines, and then branch back (return) to the next instruction after the call once the subroutine's task is done.

Sub-Routines (Functions)

- A subroutine may be written so that it expects to obtain one or more data values from the calling program (its input arguments). It may also return a computed value to its caller (its return value), or provide various result values or output arguments.
- Subroutines are a powerful programming tool, and the syntax of many programming languages includes support for writing and using them. Judicious use of subroutines will often substantially reduce the cost of developing and maintaining a large program, while increasing its quality and reliability.
- Subroutines, often collected into libraries, are an important mechanism for sharing and trading software.

Functions

- Matlab offers lot of mathematical functions
- Some are implemented in m files, some are built in (e.g. ‘exp’)
- Let’s try and build a function. In a file .m
- function z=mySum(x,y)
-     z=x+y;

Function Handle

Functions can be passed as arguments to other functions

```
function y=myfun( f, x)
    y=f(x)



>> myfun(@exp,3)
```

Anonymous functions

A quick way to create functions on the fly

```
>> myfun=@( x) x * x;
>> y=myfun(5)
```

Operators

- Operators are functions which can be called via a special syntax, e.g. *a `op` b*, generally different from other functions (as we will see later). They can simply be called via infix notation: 2+2
- E.g. '+', the addition operator, is a function which operates on two arguments and returns a value
- They have precedence rules (e.g. in 2+3*4 the multiplicative operator <u>typically</u> has precedence on the addition operator)
- All languages have a number of built-in operators)

Octave: Powerful Syntax for Maths

- To solve a linear system *Ax*=*b*, we can use the inv(A)*b, or A^(-1)*b, or we can use the more efficient built-in solver, which we can call using the “\” operator: A\b

```
>> A = [1 2;3 4]; b=[1;1]; A\b
ans =
  -1
   1
```

Matlab: Plot

- We can use the plot command

Matlab: Sequence of Instructions

- In Matlab we can chain a sequence of instructions in a script file
- But is is better to use a function with no arguments

Control Flow Structures

- The instructions in a program are executed sequentially. This is not usually very useful by itself, because it does not allow:
  - execution to adapt to inputs
  - to avoid code repetition
- There are however special instructions which allow to change the flow of a program
- At low level they are all implemented via the primitive
  - *if some condition is true then, instead of going to the next instruction, jump to instruction "i"*

Conditional Operations

- Ask a question and then select the next operation to be executed on the basis of the answer to that question
- In most languages this is represented by the instruction if-then-else

if – then - else

- This is the simplest possible structure

```
if (condition)
    then  A
    else  B
```

- Sometime the ‘else’ part may be missing

```
if (patientTemperature>37)
    then  print “I recommend paracetamol”
    else  print “You are fine, go home”
```

if – then - else

- In terms of instruction flow, this is equivalent to:

```
1. input patientTemperature
2. if patientTemperature<=37 then goto 5
3. print “I recommend paracetamol”
4. goto 6
5. print “You are fine, go home”
6. end program
```

Nested If Statements

```
if (patientTemperature>37)
    then
        if patientTemperature>40
            then print “Go to the hospital”
            else print “I recommend paracetamol”
    else  print “You are fine, go home”
```

- Allow to describe control flow graphs as complex as we want

Matlab: If then else

```
if condition
    instruction A
else
   instruction B
end
```

The ‘else’ section is optional

Indentation

- In the previous slide we used indentation
- In most languages indentation is just optional
- It is good practice to make extensive use of it, as it makes source code easier to read

Loops

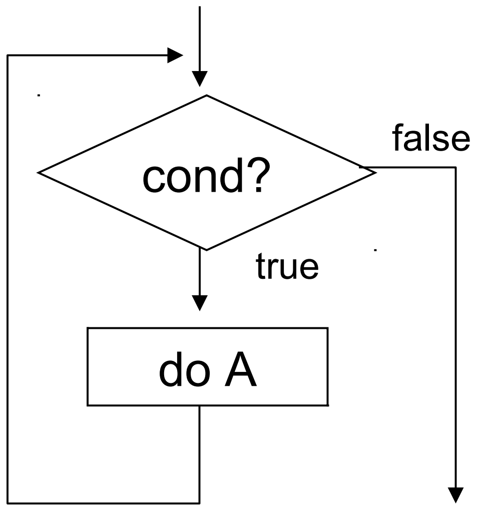
- Tell us to go back and repeat the execution of a previous block of instructions

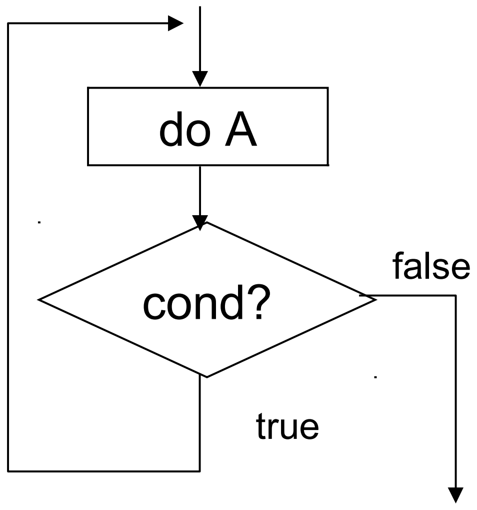
```
check tyre pressure
if tyre pressure is too low
    connect the pump
    repeat until tyre pressure is ok
            pump 10 times
            check tyre pressure
    disconnect the pump
ride your bycicle
```

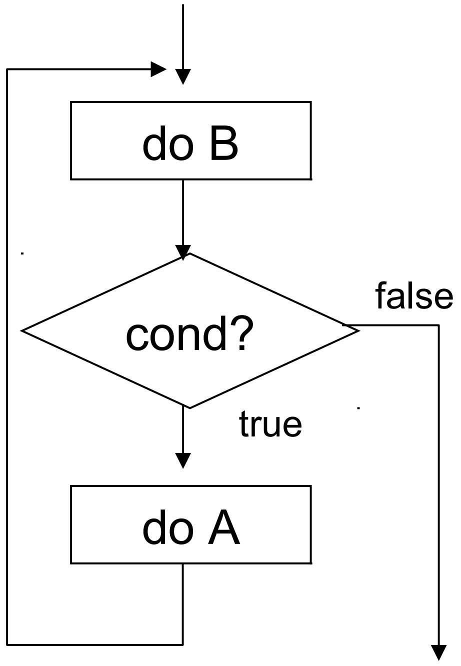
note I used indentation
to define blocks of code

Loops

- Three types, depending if the condition is checked at the beginning, at the end or in the middle

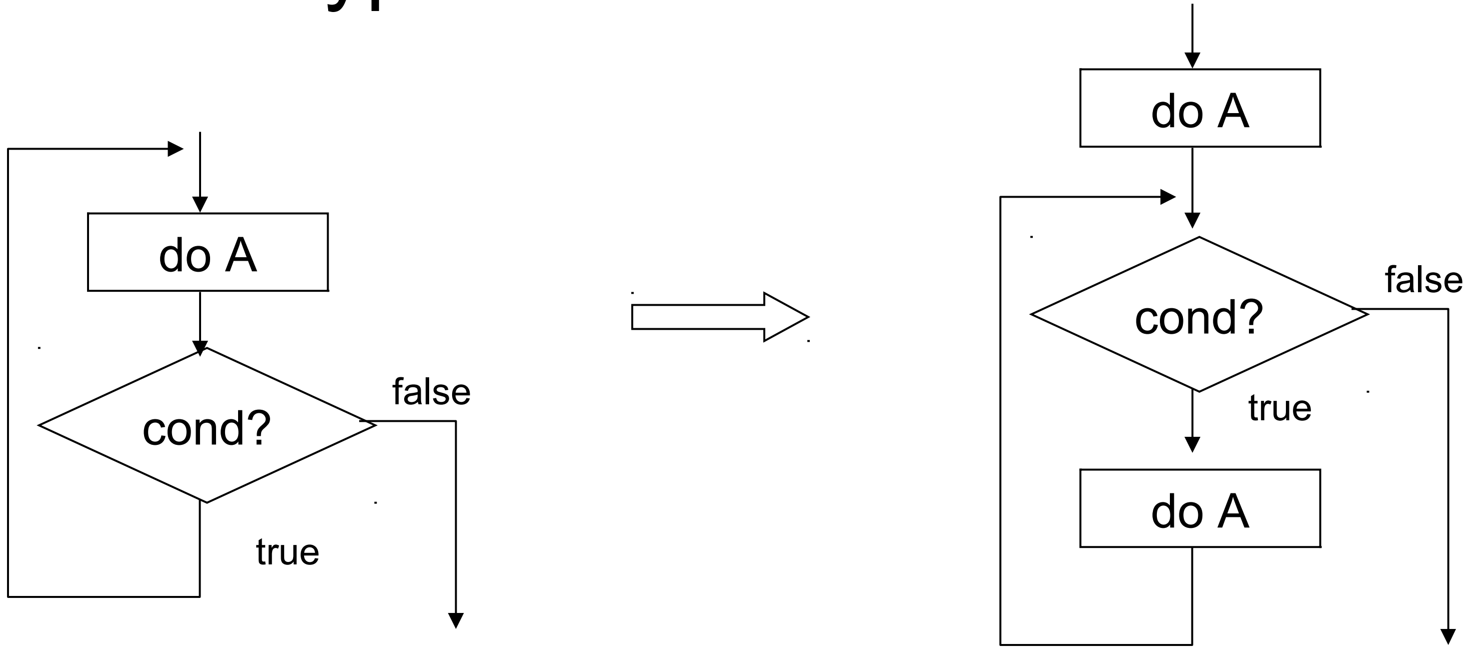






Loop Equivalence

- All type of loops can be transformed into another type



Loops In Matlab

```
for i=1:10
endfor


while (cond)
   …
endwhile

do
   …
until (cond)
```

- A for iterates on the elements of the array
- In While the condition is checked at the beginning
- In Do the condition is checked at the end, meaning there is always at least one iteration
- Choice of while, for or do is determined only by convenience, as we can always converted one type of loop in the other

Recursion

- When a subroutine directly or indirectly calls itself
- Recursive subroutines are defined in terms of tow parts: recursion and exit condition
- Recursion is equivalent to iteration (anything can be done either in recursive or iterative form)
- Some languages favor one of the two programming style
- Certain algorithms are more naturally described in one way rather than the other

n Factorial

Recursive

```
f = factorial( n )
    if (n==0)
         then f = 1
         else f =
       n*factorial(n-1)
```

$$n! = \begin{cases} n(n-1)! & n > 0 \\ 1 & n = 0 \end{cases}$$

Iterative

```
f = factorial( n )
    f = 1
    while (n>0)
         f=f*n
         n=n-1
```

$$n! = \prod i$$

- In this case it is completely equivalent to formulate the problem in recursive or iterative form

Factorial Execution

Recursive

```
factorial( 2 )
  return 2 * factorial( 1 )
      return 1 * factorial( 0 )
         return 1
```

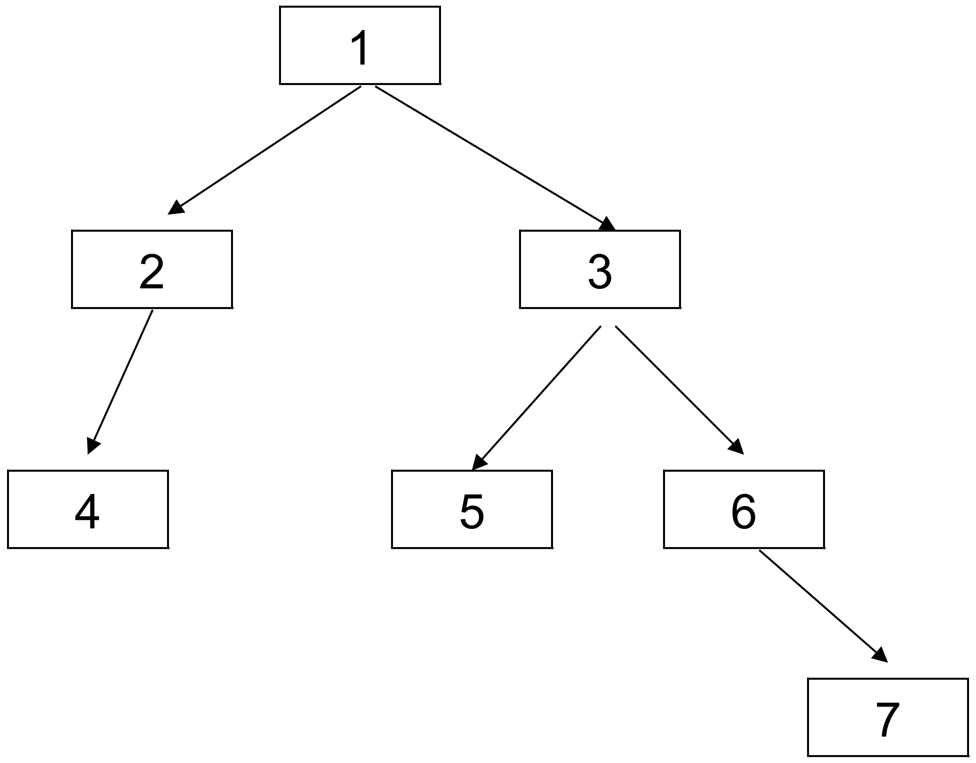
Iterative

```
factorial( 2 )
    f = 1
    f = f*n = 2
    n = n-1 = 1
    f = f*n = 2
    n = n-1 = 0
    return f
```

Visit a Binary Tree in Pre-Order

- Consider an algorithm which print the content of a binary tree of arbitrary depth in pre-order (always visiting the left node first)
- Every node contains one value, and possible connections to a left node and a right node

Visit a Binary Tree in Pre-Order



We want the tree to be visited in order: 1,2,4,3,5,6,7

Visit a Binary Tree in Pre-Order

Recursive

```
visit( node )
    if (!null(node))
        print node.value
        visit node.left
        visit node.right
```

- In this case the iterative equivalent is pretty hard to write, as we would need to simulate explicitly the behavior of the function stack

Finding Errors (Debug)

- Debugging is an art: some bugs can be really time consuming to find and require really skillful strategy to pin them down
- To help the task you need to follow good practices, i.e. you need to write code well organized, documented, indented and consistent, i.e. you need “style”

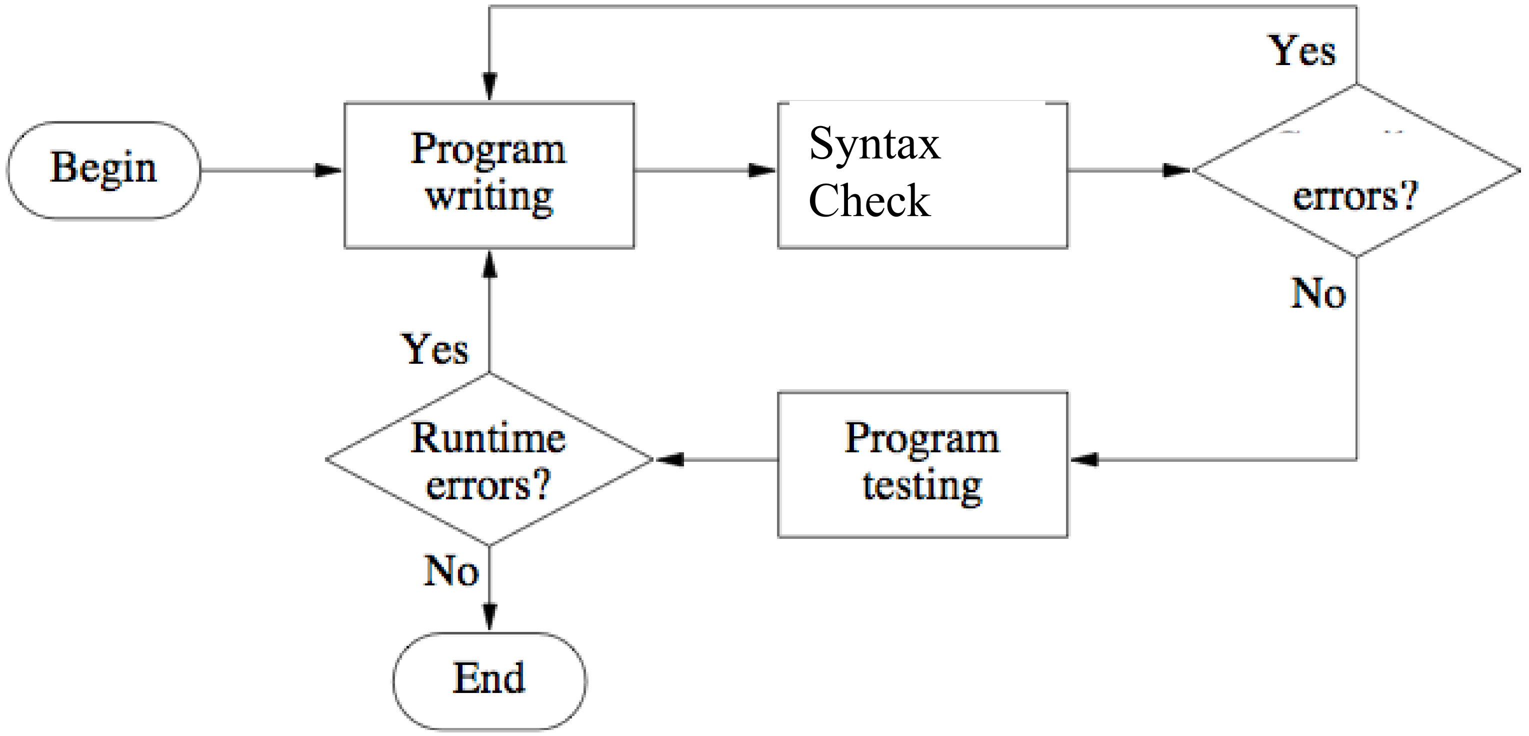
Error Types

- Design Error
  - The solution is conceptually incorrect, i.e. do not address the problem (e.g. Problem: sort ascending an array of numbers, where all numbers except the first two are already sorted. Solution: if x1>x2 the swap x1 and x2. Does this work?)
- Implementation Errors
  - Mistakes in the implementation of the solution (e.g. want sqrt(b^2-4ac), but type sqrt(b*2-4*a*c))
- Runtime Errors
  - At runtime some inadmissible behavior occurs (e.g. forget to check if b2-4ac>0 before taking sqrt())
- Syntax errors
  - Syntactical constraints of the language are violated (e.g. forget to close a parenthesis)

Finding Errors (Debug)

- Syntax error are usually found by the compiler, which will complain.
- All other error types requires brainstorming (how can I possibly get that figure?) or a proper debug session (execute the program step by step and inspect results at every step)
- Some languages offer a debugger, which simplify the task. If not we can always print out the info we need at every step, to find the place where things go wrong.

Program-Writing Lifecycle



Debug in Matlab

- Print data as you go
- Use the debugger

Examples

- a simple function
- input output
- loops (matrix multiplication)
- function arguments (nargin, nargout)
- error handling
- local function
- usage of ‘find’
- function handle
- recursion
- charts (2-axis + semilogy)
- Timing a function
- Plotting convergence

Testing Methods

- Unit Test
  - verify the function returns the expected result manually computed against a fixed set of inputs
- Regression Test
  - Verify that today the function returns the same results as yesterday
- Property Random Test
  - Feed random inputs and verify if some properties of the output are respected
- Randomized Duplicate Method Test
  - If we have two way to do the same thing, feed random input and verify both methods always return the same result

Good Practices

- Give meaningful names to variables
- Add plenty of comments
- Use a naming convention
- Use correctly indentation
- Avoid goto statement (not available in Octave)
- Avoid convoluted nested ‘if’ structures
- Do not use scripts
- Do not use global variables
- Use functions for modularity and maintainability
- Keep procedures short and with well scoped behavior