Imperative Languages

Imperative languages are abstractions of the underlying Von Neumann computer architecture.

One of the main component of the computer is its **memory**, storing instructions and data. In Imperative languages **variables** represent memory cells.

To understand the behaviour of programs we need to know *properties of variables* and the *flow of control among instructions.*

Variables

Main properties of variables:

- **Name:** usually a string of characters with some constraints (e.g. maximal length, etc…)
- **Type:** indicates the range of values tat are allowed as well as the operations that are available. Some languages are untyped, others are statically typed (Java, Haskell) or dynamically typed (e.g. Python, Ruby, Java Script)
    - **Dynamically typed:** where programmer is not obligated to provide the types at the start and the type Is dynamically determined.
    - **Statically typed:** where programmer is obligated to provide the types at the initialization.
- **Address:** The memory address to which the variable is associated (also called L-value).
    - This association is not simple: the same name may be associated with different addresses in different parts of the program; in some languages I is possible to associate more than one name with the same address (e.g. variant records in Pascal, EQUIVALENCE statement in Fortran)
- **Value:** the content of the memory associated with the variable (also called R-value)

values map to memory locations when they are allocated using assignment

Lifetime of Variables

_PROPERTY OF VARIABLE

Lifetime: time during which variable is allocated a specific memory location between allocation and deallocation.

We distinguish 4 classes of variables according to their lifetime:

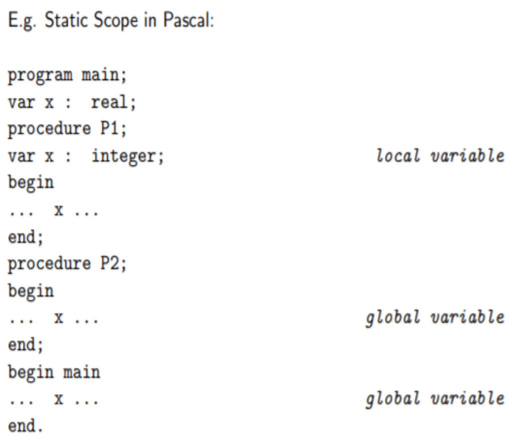
- ***Static variables*** – *When it is available during the whole executing of the program. Bound to a memory location before program execution begins and until it ends.*

  ***Advantages**: efficiency – all addressing is done at compile time, there is no allocation/deallocation at runtime.*
  ***Disadvantages**: storage cannot be reused; recursive programs need dynamic variables.*

- ***Static dynamic variables*** – *variables come into existence when function is called. They exist on the runtime stack, and are temporary, when they fell out of scope the memory is reclaimed by the runtime. They are either in the parameter list or declared inside the function (except the statics which are not instantiated on the stack)* ***ALLOCATED WHEN THE DECLARATION OF THE VARIABLE IS PROCESSED AT RUN TIME. E.G. local variables of recursive procedures.***

- ***Explicit heap-dynamic variables****– variables are nameless memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These variables which are allocated and deallocated to the heap, can only be referenced through pointer or reference variables.*
    - ***Advantage:*** *It gives a control to a programmer to allocate/deallocate memory cells manually.*
    - ***Disadvantage****: Programmer must remember to deallocate memory otherwise it will be a big waste*

- ***Implicit heap-dynamic variables*** *– bound to storage only when they are assigned values. E*.g. strings and arrays in Perl and JavaScript
    - *Advantage*: Flexibility
    - *Disadvantage*: Difficult error detection.



E.g. Static Scope in Pascal:

```
program main;
var x :  real;
procedure P1;
var x :  integer;                    local variable
begin
...  x ...
end;
procedure P2;
begin
...  x ...                          global variable
end;
begin main
...  x ...                          global variable
end.
```

<u>Scope</u>

PROPERTY OF VARIABLES

SCOPE: the range of instructions in which the variable is visible (can be used) variable is out of range if it falls out of scope.

LOCAL VARIABLES: are those declared in a block of program that is currently executing. E.g. creating a temp variable inside function to copy x into y and y into x.

Some languages have static scoping rules whereas others have dynamic scoping rules.

STATIC SCOPE: can be determined before the execution of the program (**AT *COMPILE TIME)***

- ➢ When a variable is referenced, the compiler looks for its declaration in the same block and if there is no declaration then looks in the parent block, etc…
- ➢ If no declaration is found, then the compiler detects an error (**AT COMPILE TIME)**
- ➢ You can infer the structure of constructs by just looking at the code.

DYNAMIC SCOPE: depends on the calling sequence of subprograms instead of their declaration. In the example on picture with dynamic scope the reference to **x** in **P2** may refer to the variable in **P1** or in main depending on who called **P2.**

TO KNOW THE REFERENCE, WE NEED TO KNOW WHO CALLED P2

- **Advantage:**
    - It is faster to a procedure to access the variables of the caller. You don't need to pass parameter. Because those variables are implicitly visible in the called subprogram.
    - Easier to implement in interpreters
- Disadvantage:
    - It is not possible to do static type checking
    - Programs are difficult to understand
- Some simple languages use dynamic scoping (e.g. shell scripting languages like Perl allow the programmer to choose dynamic scoping when defining a variable).

Assignment

COMMAND

VARIABLE-NAME = EXPRESSION

The ‘ = ‘ sign should not be confused with equivalence in the mathematical sense. Sometimes ‘ := ‘ is used.

The value of the expression is stored in the variable, until a new value is given (with another assignment or by a side-effect)

- IN TYPED LANGUGES THE EXPRESSIONS AND THE VARIABLE MUS HAVE COMPATIBLE TYPES, AND THIS IS OBVIOUS BECAUSE YOU ARE NOT ALLOWED TO ASSIGN A TRUE TO INT TYPE.

Control Statements

COMMANDS TO CONTROL FLOW OF EXECUTION OF INSTRUCTIONS ON THE MACHINE

Programs in imperative languages are sequences of instructions. The next instruction that will be executed is the one that follows in the text of the program unless a control statement is used. CODE IS READ LINE BY LINE, IF CONDITIONS ARE NOT EMBEDDED.

The main control statements in imperative languages are:

- ***Selection constructs – if statements***
- ***Looping constructs – repeating***
- ***Branching instructions – allow to jump in the code***

ANY SEQUENTIAL ALGORITHM CAN BE CODED WITH JUST IF-THEN-ELSE STATEMENT AND LOGICALLY CONTROLLED ITERATIONS.

All languages contain more constructs to facilitate the writing of programs. Moreover, serval statements can be grouped together to form compound statement.

Selection Statements

Used to choose between two or more execution paths in a program. We distinguish *two-way selectors* and *multiple selectors*.

Two-way selector (if-then-else):

- if (1 == 1) return true else false
- The semantics of java specifies that the else s matched wit the most recent unpaired then clause(the inner one in this example)

Multiple selectors

Multiple selectors are a generalisation of two-way selectors.

- **E.g.** in Java, C or C++ we can write a **SWITCH statement**
- *In some languages at the end of a branch the control goes to the instruction after the selection, in others we ned an explicit break or return.*

Iterative Statements (Loops)

Iterative constructs cause a statement or sequence of statements (the body of the loop) to be repeated.

- If the test for loop completion is done before the execution of the body we say that it is a **pre-test** loop, and if it occurs after the body is executed, it is a **post-test** loop.
- **There are two kinds of loops:**
  - *Counter-controller: for example, a for loop with a ' I ' counter which is managed and changed before execution of body.*
  - *Logically-controlled* loops: for example, a while loop with some Boolean(logically controlled) condition
- The **while** is a **pre-test** because it checks condition first and then execute the body
- The **do-while** is a **post-test** because it checks condition after execution of the body

Structuring the Program

SUBPROGRAM = METHOD

- **BLOCKS:** a block is a group of statements, delimited by keywords such as **begin** and **end** or by separators like **{ }.**

  A block may contain not only commands but also declarations (of constants, variables or types). If the language has static scope, these will only be visible inside the block (they are local)
- **SUBPROGRAM:** a subprogram is a generic name for a named block that can be executed invoked explicitly. In this case the block of statements is executed upon invocation, and the control returns to the calling point after the execution of the subprogram.

  Advantage: can be accessed from anywhere within the code.

  Examples: procedures in Pascal, functions in C, methods in Java.

  When a subprogram is declared together with the name we can associate ***parameters****. These are called formal parameters, whereas the ones provided when the subprogram is invoked are the actual parameters.*

  SO, IN MY WORDS: WHEN A METHODS IS DECLARED OR A SIGNATURE OF A METHOD THE PARAMETERS TAKEN ARE CALLED THE ***FORMAL PARAMETERS****, HOWEVER IF THOSE PARAMATERES ARE ACTUALLY USED THEN WE CALL THEM* ***ACTUAL PARAMETERS****.*
  THE VALUES OF THE ***ACTUAL PARAMETERS*** *will be replaced for the* ***FORMAL PARAMETERS*** *when the subprogram is invoked.*

  Different languages use different parameter passing mechanisms.

Advantages and Disadvantages of typed / untyped languages

- Advantages:
    - Types help detecting programming errors at an early stage, making the debugging process faster and easier.
    - Types help in the design of software since they are a simple form of specification
    - A program that passes the type control is not guaranteed to be correct but in a strongly typed language, it is free of runtime type errors.
- Disadvantages:
    - Types impose restrictions on the form of programs.
    - If the language does not have a type inference system, then writing a program is harder.