Dear friends,

This the second installment of the lecture notes. I have been getting a few queries – but not many. It will be nice if I get more reactions from your side. Or is it that I have to wait till T1 time for your comments.

So, more one installment – RFC!

Regards,
N S Kumar
ask.kumaradhara@gmail.com

We shall end this chapter answering a few queries I have received from my young friends and colleagues. If you are very curious, you may go to the end of this chapter – as you would normally do while reading a mystery story!

Language Design:
There are a few factors which influence the language design.
a) **Von Neumann Architecture:** This architecture implies the presence of a single CPU, a few registers and linear memory connected to the CPU by a bus. Both the program and data are stored in the same memory. There is no way to distinguish once they are in memory. Many languages tend to match the variables to the locations of the memory. The classical example is of 'C' array which uses the base and the index registers to index to an element in the array. So variables are modeled based on memory locations and control structures based on condition flags and jump instructions.
These machines provide a very low level abstraction.

Earlier versions of Fortran followed allocation of all variables in the beginning of the program itself. Is this model very efficient in time? Can this model support recursion?

b) **Stack based machines:** A few Burroughs computers supported stack based machine with zero address format instructions. Java virtual machine is also stack based.

c) Functional languages : These provide a totally different paradigm of programming. A function is a first class citizen of the language. They can be assigned, can be passed as argument and can be returned from a function as the result. These use recursion as the primary control structure. The whole world is immutable.

Program Design Methodologies:

- Imperative
  - specify the solution to a problem in terms of commands
  - example: fortran, C, ...
- Object Based
  - combine data and operations together
  - support encapsulation, abstraction and composition
  - example: Ada 83, C++ STL
- Object Oriented
  - is also object based
  - support inheritance and polymorphism
  - provide dynamic behaviour based on values of variables
  - example: Java, C#, Python, ruby, ...
- functional
  - follows mathematical concept of functions
  - no assignment, no looping
  - functions are first class citizens
  - world is immutable
  - example: Lisp, ML, scala, haskell
- logical
  - specify facts and rules - no order
  - verify the goals
  - example: Prolog

- generic
  - develop data structures and algorithms which operate on various of types of data
  - make type itself a compile time parameter for the data structures and

    algorithms
    - example: Ada, C++, C#, Java 1.8 ...
- concurrent and parallel
    - execute the program units together
    - use multiple processes, multithreads
    - use multicore support or GPGPU(General Purpose Graphic Processor Unit)
    - example: library support through openMP, MPI, pthreads, …

We may discuss these again at appropriate points.
All these methodologies provide different abstraction of the problem being solved.

Scripting language:
These is no clear definition of scripting language. Normally, these are the criteria.
- no declaration of variable with respect to type
- compiler present at run time
- data type is weak or enforced at runtime
- might support polymorphism at runtime based on type
- time for development is short

Language Design Trade-off:
There are always conflicting requirements. The designer has to make the selection based on the philosophy he wants his language to follow.
reliability(safety) vs efficiency
- Enforcing safety at runtime would require more time and space at runtime. You may compare index out of bounds of Java and C.

  The C programmer would say that Java penalized the right programmer.
- **Support dynamic arrays**

  overheads at runtime vs flexibility at runtime
- **runtime type and reflection mechanisms and flexibility**

  Python provides polymorphic behaviour based on type at runtime. This would require that the Python runtime enforce the operation based on checking the type at runtime.

Python also provides adding a member function to a class at runtime – called monkey patching.

Java provides mechanisms to check the content of a class at runtime -find a method satisfying the parameter list – get a handle to it and invoke it.
C++ does not provide any of these,

- **Complexity and readability**
  - Is it good to have simple statements which are readable or allow for combining expressions to form a very complicated statement? There is no single answer for it. In C++, this is a valid statement and not in 'C'.

    (a += b) *= c;

    Some language designers prefer to allow for powerful constructs which combine many operations whereas others prefer simplicity.

  - Should we support pointers? How about pointer arithmetic? How to handle garbage? Should we provide a mechanism for the user to free memory? Should we support a garbage collector? Should this be based on mark and sweep algorithm or reference counting? Can the user decide which garbage collector he may want to use? What about dereferencing a dangling pointer?

- **What should be the translation mechanism?**
  - Should we compile to machine code directly? Should we compile to a program in an intermediate language? Should we compile to a data structure in memory? Should we convert the code line by line to machine language? What to do in case of loops? Should we at least check for syntax? Do we look ahead in the code or re-arrange the statements? Do we support optimization? What about error messages? Do we provide the position in the source code with a proper error message? Do we correct common errors? What about warnings? Should we accept a string as input from the keyboard or a file and translate and execute that code?

Should we support interoperability with other languages? If converted to an intermediate language, should we convert to machine code before loading? Or should we convert only part of the code frequently used to machine code? Or make the decision as the program executes? Should we support pre-processor? Should we support import mechanisms?

So many questions (Yaksha Prashne or Vikram - Bethal), the designer has to answer!!

- You may want to check what JIT, HOTSPOT, inlining mean.
  - You may want to check the difference between a pro-preprocessor and inlining of code.

- **Programming Environment:**

Popularity of a language and its usefulness depends on the tools available for the programmers. Here is a small list of such tools.

- IDE (Integrated Development Environment)
  - intelligent editor
  - integrated compile and link tools(build tools)
  - debug tools
- Profiling tools
- Testing tools
- Static analysis tools

FAQ:

a) what are the steps in executing a program in 'C' or C++?

The steps in executing a program are: (C C++)

- preprocess:

    consider all macros, #defines ...

    input: source file

    output : translation unit

- compile:
  - input: translation unit
  - Output: object file
  - This will have machine code equivalent of your program
  - location for initialized and uninitialized global and static variables and normally string literals.
  - It will also indicate to the linker,
  - which names are defined here and therefore can be used by other translations(public symbols)
  - and which are used here and not defined(unresolved externals)

- linker:
  - Input: object files and libraries
  - output : loadable image
  - Linker resolves unresolved externals by mapping object files and libraries.
  - normal errors:
  - 1. unresolved externals
  - 2. multiple definition

- loader:
  - input: loadable image
  - output: process
  - Transfers machine code to text segment
  - allocates memory global and static variables and string literals
  - places the first instruction(entry point) to start the execution

- runtime
  - CPU executes instructions - fetch - decode - execute - cycle.

- Why Cobol is anti-thesis of a language?

Let us state both good and bad things about the language design. It has been one of the most successful languages ever in spite of its short coming.

good things about Cobol :

- clear separation of entities: identification, environment, data and procedure
- good support for file system: sequential. relative and indexed sequential files
- supposedly English like
- hierarchical data structure

bad things about Cobol:

- verb perform overloaded:
  - function call like mechanism, looping
- no concept of parameter passing
- no variables with blocks
- no recursion
- no type concept
- should specify storage for each variable(dataname)
- too verbose
  - add a to b=> is ok
  - add a to b giving c=> is not ok

- What is the difference between 'compile to an intermediate form' and 'compile to an intermediate language file'?

perl : compiles to an in-memory data structure - some variation of a parse tree – is an intermediate form

java: compiles to one or more class file

python : in some cases, creates a compiled file