Preface

Dear friends,

I am back again(punaha Ayan ...). You may remember we were together 18 months back studying programming, Object Oriented Programming and Java (in that order) together. This time, I am trying to bring to you another cup of tea(should I say coffee) - principles of programming languages.

You may want to read this (for whatever it is worth) along with the standard books on principles of programming languages. I will try to make a sincere attempt to share my experience of learning and programming in an umpteen number of languages using different paradigms. It is not a replacement for a standard text book.

All the best. Enjoy.

N S Kumar
ask.kumaradhara@gmail.com

Introduction:

The title of the course is “Principles of Programming Languages”.

So, we are talking about languages. Language is a means of communication. You may use a natural language to communicate with your friends. You may use p-language or some language to a few which is unknown to others so that others do not get what you are saying. Deaf use a sign language. Musicians have their own language. You may write something in HTML so that your browsers can render them nicely. It seems my friend was asked to keep his young twins separate – lest they develop a language to communicate amongst themselves and ignore the parents!

You may remember(hope you do!) that a language is defined formally either by grammar or by an automaton. A language defined by a grammar G(N, T, P, S) is a set of strings having zero or more terminal symbols and derivable from the start symbol S.

In this course, our interest is only those languages which can make the computers do something – programming languages.

We shall examine in the course the underlying principles across a wide spectrum of programming languages. We try to understand the constructs that are selected by language designers while designing the language.

You may want give a treat to your friends(Hope I am also included!). You may want to give some soup, some sweet and some tandoori item and some rice dish and of course dessert. You have lots of choices. You may choose tomato soup, tomato halva(?), tomato stuffed parota and tomato rice! Or you may provide some variety. It is your choice. It depends on whether you want your friends back again for another feast.

The designer of a language chooses the different kinds of constructs and brings them together in his language. The implementor writes a compiler(some sort of translator) for the language. The programmer writes the program based on his understanding of the language. If all of them are not in the same page, the hell breaks loose. This course should help you put yourself into shoes of these people and understand the underlying principles.

These are some of the titles of books of great writers on this subject. I have not provided the author names – you may find them yourself.

- Concepts of Programming Languages
- Programming Language Pragmatics
- Programming Languages Design and Implementation
- Fundamentals Of Programming Languages
- Programming languages: a grand tour
- programming languages principles and paradigms
- Principles of Programming Langaues
- The World of Programming Languages
- Programming Language Landscape
- programming languages concepts and constructs

I am sure you will have lots of questions to ask. I will ask them myself and answer them myself.

- What are the prerequisites for this course?
  - Knowledge of programming
  - Knowledge of a few programming languages
- Is this a core course in Computer Science Curricula across universities?
  - Yes. You may check
  - https://www.acm.org/education/CS2013-final-report.pdf
- Is there research happening in this field?

    - Yes. There is a special interest group in ACM. SIGPLAN : Special Interest Group in Programming Languages. *www.**sigplan**.org/*
- Is this a theoretical course?
    - Absolutely NO
- Do I have to memorize?
    - NO
- Do I have to think?
    - YES
- Do I have to learn the syntax of a number of languages?
    - This course is not about learning languages. It is about understanding the semantics of the various constructs. We may discuss the syntax very occasionally when it is very unusual. You do not have to memorize the syntax.
- Is this a programming course?
    - NO. we do write some small programs to understand the concepts and variations across the languages.
- Is this a course in learning various programming paradigms?
    - NO. We do introduce a few with respect to the language design.
- Is this a course in problem solving?
    - NO. Our interest is about understanding language constructs.
- What is the course all about?
    - It is about understanding the various constructs in programming languages

  from different perspectives
  - Programmer’s perspective
  - Language designer’s perspective
  - Compiler perspective

  Also some extent
  - OS perspective
  - Architecture perspective

- Can you given an example of how we proceed in this course?
  OK.
  Let us take an example of a for loop to walk through a collection.
  It is called enhanced for in Java, range for in C++, foreach in perl and so on.
  The following indicates the structure and not necessarily syntax in any particular language.

Syntax: for <variable> in <collection>
<body>

What could be the semantics of this looping construct?
What is the life and the scope of the variable?
Does the variable get a copy or a reference to an element in the collection?
What is the value of the variable once the loop is exited?
Should the collection physically exist? can that be lazy?
Can we have user defined collection?
What if the loop is exited abnormally?
Can we traverse the same collection more than once simultaneously?
Does the modification to the collection affect the loop?
If my architecture supports multiple cores, can we traverse in parallel?
Can we change the elements of the collection by using the loop variable?
Do we have a default loop variable?

By answering these questions, we would understand the semantics and given a new language(new to me!), I can find out how the for loop works in that language.

Evolution of languages :

Digital computers were programmed in machine language in the early years. This was replaced by Assembly language. This had advantage of mnemonics for operations, names for data. This had almost one to one mapping with the machine instructions and registers. Portability across computers was an issue. Maintenance was a nightmare. (Even now it is – that is what you will do rest of your life!).

Fortran(FORmula TRANslation) was the first successful high level language ever designed. It was designed by a team lead by John Backus. He showed to the world that a program in a high level language can be almost as efficient as a program in a machine language. Fortran was designed for Scientific and Engineering Applications. It is active even today. The latest standard is 2008 and expecting 2015 in a couple of years. It is the best language for SIMD architecture – Single Instruction Multiple Data. It supports addition of two arrays in a single clock cycle using multiple processors.

COBOL(COmmon Business Oriented Language) was designed by a group called CODASYL for commercial applications. It has been the most successful language – in terms of number of years of active usage – ever. It is supposed to be English like – Add A, B giving C.

It has a nice organization of the program structure. It has a very powerful file system. It is an anti-theses of a programming language when compared to the languages of today. Latest standard is COBOL 2014. These languages are immortal – they do not easily die! They are like cats.

ALGOL : Algorithmic language was developed by a team of computer scientists – Dijksta, McCarthy, Wirth. It was the first language which allowed writing good and maintainable programs.

PL/I : This was designed by IBM in early 60s. This had all the features of Fortran, COBOL and Algol. It had additional features which were well ahead of time. The language became extremely big and had many defaults. No body could master it. It failed because of its own weight. This was like our great king Muhammad bin Tughluq – a bit too early in the history.

Pascal : This was designed by Niklaus Wirth for teaching. The strength of Pascal is its simplicity.

'C' : This was designed by Dennis Ritchie as a system langauge. 'C' provides low level features as well as features which can be implemented efficiently. The latest standard of C is C 11.

C++ : This was designed by Stroustrup in early eighties. This allows multiple paradigms as well as multi level abstractions. The latest standard is C++ 14.

Java : This was developed in mid 90s by Gosling and his team. It is an object oriented language. It is the most popular language as on today according to some surveys. The standard Java 1.9 has been released recently.

C# : This is part of the dotnet framework of Microsoft. This provides java like features with inter-operability of binaries.
Apart from these, we have had a sequence of scripting languages.
Shell, awk, perl, tcl, python, ruby, lua, scala and so on. The list seems to be endless. We have tower of babel of programming languages.

There are a couple of special languages. One is Lisp(LISt Processing language) designed by McCarthy at MIT. In this language, both program and data are lists. So, a program can modify itself while executing and learn in the process. This was a preferred languages in the field of artificial intelligence for a number of years. The other is prolog(Programming in Logic) where we can state the facts and the rules and deduce the results based on them as in propositional logic and predicate calculus.

You may check the popularity of languages using the following web sites:
https://en.wikipedia.org/wiki/Measuring_programming_language_popularity
You may also note the parameters used in deciding the popularity.

Translation Process:

A program in a high level language cannot be run directly by a computer. We require to translate these programs into machine language. There are number of variations of translation process amongst languages.

- pure compiler : compiler to machine code and create a loadable image
- pure interpreter : read statement by statement – convert to machine code and execute
- compile to an intermediate form – into a file or a data structure and then interpret – Java is an example for the former and perl is an example for the latter
- check for syntax errors and then interpret – in some contexts, create a compiled file
- compile to an intermediate language file - convert to machine code before executing it

Experiment:

a) Try what happens in the following python and perl programs? Why?

```
# python3.5 ex1.py
def foo() :
        print("one")
foo()

def foo() :
        print("two")
foo()
```

```
# perl ex2.pl
$\ = "\n";
sub foo
{
        print("one");
}
foo();
sub foo
{
        print("two");
}
foo();
```

Translation Model in 'C':

- preprocessing : input : source file; output : translation unit
    - gcc -E <filename>

This will give the output on stdout. You can check the output of the preprocessor.

- Compile:
    - gcc -c <filaname>

The output is an object file. You can use the nm command to check what symbols are public which can be used by other translations – external linkage and what symbols are used and not defined – unresolved external references.

- Link:
    - gcc <object files > <libraries>

Output is a loadable image if everything is fine – of course Murphy will be always waiting round the corner. The possible errors are multiple definition and unresolved externals.

You may want to use nm and locate the symbols that you have used in the program.

- Load
  - create a process which is basically a virtual address space
- run
  - Execute a thread in a process – If you are lucky, you get errors!

Experiment:
Check the address of global and local variables by running the program again and again. Find the reasons for the result.
Check id of variables in Python? Are they same each time you run? Can we nest calls to id?

Programming Domains:

- **Scientific Applications**
  - Complicated computations on similar data
  - may have to manipulate very small and very big numbers.
  - fortran - suited for SIMD architecture
    can operate on arrays using parallel processing
    openMP support for fortran

- **Business Applications**
  - large amount of data and simple processing
  - cobol - ruled the world for years
    database - rdbms and non-transaction db are popular now

- **Artificial Intelligence**
  - started in late 50s in MIT
    - ideas then
      self learning program - Lisp
      program and data equivalent
      program can learn as it executes
      logic programming - prolog

state the facts and then the goals

- ideas now

more statistic based

fields : ML, NLP ...

- **System Programming**
  - should interact with the hardware
  - require low level features
  - forms a layer - should be efficient
  - main languages : C, C++

- **Web Applications(Internet applications in general)**
  - client - server architectures

client : more often than not is a browser

common language : javascript

server : could vary

php, python, perl, java ...

Language Evaluation criteria:

How do we compare two languages? How do we select a language for a particular domain ? How to check whether a particular feature of a language compares with that of another? What about the various programming paradigms that a language support? Can we combine these paradigms?

These are questions for which answers are never definite. You may like Vidyarthi Bhavan Dosa. Others may not. They may not like the crowd there. The may not like the ambience. They may not like the timings. They may not like the oily stuff. It is equally true of languages.

There is an article titled 'C++ critique by Joyner' which criticizes C++ left,right

and center. Stroustrup terms the Java white paper as a pack of white lies when he visited out college last year. Language wars are always on!

In spite of these, we have some criteria for comparison. We shall discuss them briefly.

- **Simplicity:**

Simple a feature, it is easy to master and therefore easy to use. If the features are very simple, then expressing a complicated idea may require lots of code.
Binary number system has two digits, expressing a decimal number may require far more digits in binary number system than in decimal number system.

  - Simple of set of constructs:
    - python has just two looping structures where as perl has 5
  - feature multiplicity:
    - if we can express an idea in more than one way, would all of them mean the same thing?
      - In Python, if x were a list, is x = x + x same as x += x?
    - What could be the meaning ++ and –- on scalar variable holding a string (say zz) in perl?
  - Operator overloading:
    - good, bad or ugly?
      - Could be misused. Operator + may do subtraction!
      - May not make sense in that domain – division of matrices
  - default parameters:
    - what if python open made 'w' as the default mode? You may keep losing your files and the cool.
  - Readability:
    - perl allows us to write 'write only' code
    - Reading regular expressions is not easy
    - every command is sed is a single letter – In the good old days, the error message was very helpful – 'command garbled'!!

- **orthogonality:**
  - relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language
  - List comprehension allows combining for – map concept and if – filter concept – together in any way
  - structures and array in 'C' are not exactly orthogonal as functions can return structures and not array.
  - Is the operator + commutative? What in Python? Why friends in C++ to support this feature? What happens in Java in we cascade operator +? Is the operator == associative?

  Orthogonality allows composition of features each of which does not affect the other.

- **data types:**

  type : encapsulation of set of values and set of operators
  (could be a sequence of values if there is an ordinal position for the values)

  Can we express our data items using the primitives?
  Can we make our own types?

  Does the language support boolean type? complex type?
  Type for currency? what about precision?
  what about the operators on these types?
  Are types compile time mechanism or runtime mechanism?

  - **type checking**:

    Can we assign a value one type to another?
    is there a default conversion?
    Is narrowing allowed?
    Could that involve a function call without telling the user?
    Can we combine values of two different types in operation?
    Can checking happen at compile time or runtime?
    Does the language support exception handling?

What about overflow and/or underflow?

- accessing a component?
- using an uninitialized variable?
- what is the overhead of using type checking?

- **Syntax:**
  - is the syntax intuitive?
    - Special words or delimiters
      - example begin end in Pascal, { } in C like languages
      - single end for all blocks
      - end closing nested inner blocks
      - used as variables
        - fortran : if = 111
        - PL/I IF IF = THEN THEN ELSE = 222;
      - lexicial ambiguity: use of ()
    - keywords
      - overloaded keywords example : static, final
      - adding new keyword as the language evolved?
- **Support for logical and physical modularity**
- **Availability of libraries**
- **Naturalness for application**
- **Ease of program verification**
- **Portability**
- **Flexibility (generality)**
- **Tools for development, build, debug and deployment**
- **cost of usage**
  - learning cost
  - development cost(program writing)

- build cost(compile related)
- runtime cost
- language implementation cost(time for compiler development)
- reliability cost(runtime checks - medical equipments)
- maintenance cost

These are some of the features which affect the selection of language as well as comparing languages. You may want to think about these.