Prakash Hegade

Design of a Programmer

~*~*~*~*~*~*~*~

Design of a Programmer

By Prakash Hegade

Smashwords Edition

Copyright 2016 Prakash Hegade

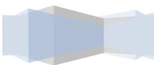
~*~*~*~*~*~*~*~

This ebook is licensed for your personal enjoyment only. This ebook may not be re-sold or given away to other people. If you would like to share this book with another person, please ask them to download an additional copy for each recipient. Please do not use the contents of this book without the author's permission and reference.

~*~*~*~*~*~*~*~

For any query or questions kindly mail: **prakash.hegade@gmail.com**

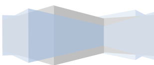
~*~*~*~*~*~*~*~



Design of a Programmer.

First Edition

2016



Preface

There are rules.

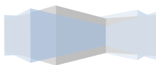
There sure are rules.

But no one is going to tell you those! They aren't told because everyone makes their own and its one of its kind. The question is how do I make my own rules to be a better programmer? Is it possible?

The answer is yes and hence is this book. This book will aid you in being a better programmer. If you have plans to start learning programming or have just began, then this is the right time to read this book. In any other case as well, the time is still good.

"**Controlling complexity is the essence of computer programming**"

- Brian Kernighan



Story One

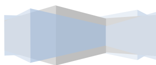
Introduction

It's time to discover the several facets of being a better programmer. One has to know many of principles coming from many of kinds and outline a few for oneself. There are stories which will tell you 'how to' and not 'what to'. The 'what to' is an exercise to discover. It's more like showing the destination and saying there are no roads yet. One has to construct the road the way most comfortable.

When you do something of your favorite you won't stop until you are contented with your work. You see some missing things which no one else does. You can easily recognize something which is missing out or not fitting in. There is this intuition of what the right thing is and how it should be carried out, which you might not be able to explain! Now, that is what you need to develop with respect to programming. It's 'easy' as well as 'not easy' being a good programmer. The reason being that there is no definition of a 'good programmer'. It's mostly the intuitions. It's mostly the gut feeling that this one is going to work and this is the best of it!

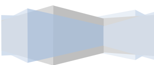
There are certain parameters which can be used to evaluate a program. There are certain principles why the various programming languages came into picture. There are certain notions which the community follows. There are certain goals and objectives for a programming language design. There are certain motivations and know how's of every programming language. **It's good to know them. It's better to realize them. It's best to adopt them.**

The stories ahead talk most generic to most specific. It's left to the reader to dive to the required deep or surf at required height. Once you get a bigger picture of



programming languages you will be all confused. Don't search for a flow to start. There is none. The way is you pick up things the way you go and form a mental picture in your head. Then you connect them together with your intuitions.

One fine day you will have a mental cloud in your head and the thoughts will pour in like rain whenever required. The clouds generally do not have a shape. They are random. This is what exactly happens in your head as well. The cloud takes the shape you give. I know. This is all foggy!



Story Two

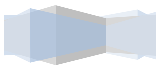
What's Up With These Machines!?

Are the machines very intelligent? Nah! They are dumb. That's why they don't understand what we speak and hence we need to build and learn all that they will understand. Now, that's why we are supposed to learn programming.

The plan is we just don't want to command the computer and see what happens when we click an icon or a button. Rather, we want to interact. We ask to do something and when the machine does it, we want to verify if the job was done right. If not, the machine is wrong (which is hardly and rarely the case) or we went wrong in instructing the machine in the right way (very likely case).

Machines need a lot of learning. Probably that is why the domain 'Machine Learning' has come into existence. Being a programmer is being a teacher. A trainer who is going to train machines on how to behave. Machine is your student; a student who is not that intelligent and needs to be told with every single detail on what to be done. Eww! You understand right? Patience! A lot's of it. You will have to put a little extra effort so that you can be a better trainer. This effort is going to be an initial investment. Once when you are on the track, things will flow naturally. You will exactly know how to deal with your student.

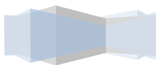
Look at the definition given by webopedia on computer: A computer is generally defined as a programmable machine. The two principal characteristics of a computer are: it responds to a specific set of instructions in a well-defined manner and it can execute a prerecorded list of instructions (a program)*. See! All that the



definition says is that it is not intelligent. You need to provide instructions in a well defined manner.

We Human beings at other hand are so random. We are creative. We don't want to do monotonous job every day. We get bored easily doing same kind of things again and again. Now okay, **if that is what machine wants, let's do it. Let's learn to program!**

*Reference Link as of May 2016:
http://www.webopedia.com/TERM/C/computer.html



Story Three
The Clear Up

Take a walk outside.

Clear all your thoughts about the doubts of 'if you will be a good programmer or not'. Flush out any opinions you have which you had gathered from somewhere else and if you had decided that programming is not your cup of tea. Yeah! If not green tea, we shall have a regular tea which tastes much better.

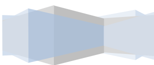
Research has shown that running and coding go hand in hand. One who runs and exercises can code better. Running is a good way to release the stress and freshen up for the good thoughts. If you can make a habit of running atleast for 10 minutes a day and few simple exercises, it is going to be very effective. No compulsion though.

Have you heard of the book series "The Art of Computer Programming" by Donald E. Knuth? You should check them out. Browse the web for more. It's good to know! Then there are many other resources as well. You will find out many in the journey. Keep some as your personal favorites along the way.

You know how in babyhood we all picked up things at our own pace and learnt the way which was very different from the others? We always had guidance from parents and loved ones and things were also dependent on the environment we grew up. No one told us 'you cannot walk', 'you will never be able to talk' and other things. We picked up things slowly as we grew. Every one of us had some things quick and some late. Today, it all does not matter. This is exactly the case with respect to programming. **We are going to do it our way which is most comfortable to us.** I hope

you get the context. To explain, clear your thoughts. You are now a good programmer in making. That making is going to take your decided amount of time. A good enthusiasm and confidence will make the process easier and faster.

Clear up. Take a walk outside. Yet again!



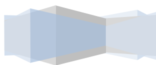
Story Four
The Evolution of Programming Languages

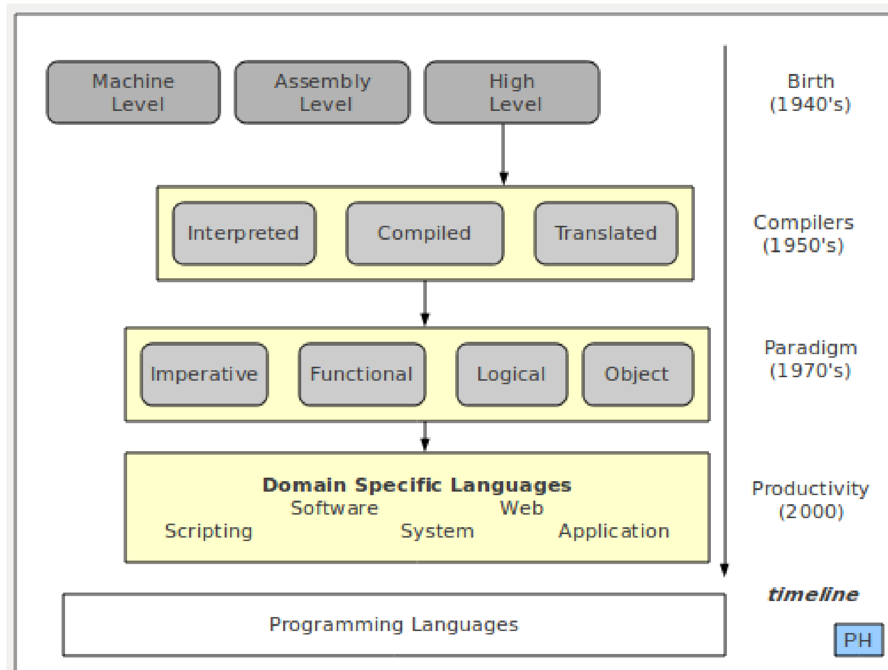
Let us first get the bigger picture. Let us understand a little bit of evolution of programming languages and get a bird's eye view. You start browsing over the classification and evolution of programming languages and, wait! There is so much to take in! How do we perceive them all?

Here is what I did. Read them all; looked upon the time line and put them in buckets and arranged them in order. Though the order is not perfectly in flow, matches the most. However, the programming languages have greatly changed over time and classification has found versatile differentiation.

The progress is so rapid that this piece of information might need several updates in near 05 years. However this article gives the bird's eye view and explains how we can put them all together. Look at the picture for the overall summary.

It all begins with Von Neumann architecture consisting of ALU, memory, input, output and the initiation of programming. Programming began in 1940's. The classification began with machine, assembly and higher level languages.





The Birth Era

Essentially we are talking about how to classify the programming languages at various periods of time. At the birth era they can be put into one of the three categories namely machine, assembly and high level.

Machine level language is a binary program. It is a program with all 0's and 1's. By nature it is very difficult for us to remember and code in 0's and 1's. Though easily understood by machine, it is not human friendly. That is how we started and we have done better.

Assembly level language is a low level programming language. An assembly instruction consists of an operation code mnemonic like ADD, MUL, JUMP etc along with a list of data arguments. It is largely dependent on type of architecture and its

machine code instructions. A little better than machine level, though still not convenient to use.

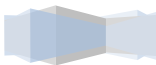
Higher level languages make the programming more users friendly. They take the features of a computer and abstract them providing a simpler view to the user. They make the program simpler and understandable as compared to lower level languages. This set is more like machine trying to understand the way we speak and communicate. Constraints and protocols do exist restricting the fluency, however still a better progress than the machine and assembly levels.

The Compiler Era

Next improvement happened over models of execution in around 1950's for the higher level languages. This is like; we now have many forms of languages and let us write tools to convert one form into another. It's like juggling up with things around to make it more compatible with what we have and what we have invented out. We hence have classification as interpreted, compiled and translated languages.

Compiled languages are transformed into an executable form before running. Either they get converted to machine code or intermediate representation which may or may not be further optimized.

Interpreted languages are read and then executed directly, with no compilation stage. It means there is no conversion into machine language instructions. It directly presents the required results.



Translated or Trans-compiled languages are those where a language may be translated into a lower-level programming language for which native code compilers are already widely available.

There have been a lot of improvements over the above explained models. Essentially they all speak about conversion between the languages.

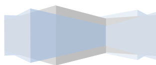
The Paradigm Era

With advancements in programming came paradigm era in 1970's. There are four major classifications: Imperative, Functional, Logical and Object oriented. It is because we wanted to structure our program and pick the best suited for environment. That is how the next revolution started. The question was 'what were the principles based on which the programming language was developed?'

Imperative is a procedural programming. The change of states of a program is observed incrementally with respect to function of time. Various control structures govern the execution of the program statements. Few examples of languages which fall under this category are C, FORTRAN, ALGOL, Pascal etc.

Functional originates from mathematics. Here time plays a minor role and the languages are about expressing the statements of a program in a formal way. The languages stress on the evaluation and using the resulting value. Examples of languages that fall under this category are Haskell, Scheme, Erlang etc.

Logical refers to languages built on axioms, inference rules and queries. To



name it more aptly in today's context, its artificial intelligence. Prolog, Datalog etc are examples of languages that fall under this category.

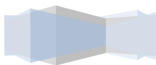
Yet another paradigm is **Object** oriented. It models the human interaction with real world phenomenon using the concept of objects. Objects communicate with each other by passing the messages. C++, C#, Java etc are examples of languages that fall under this category.

The Productivity Era

From late 90's and 2000's, the era of productivity began. Programming languages started turning out to be specific to every domain. Each domain demanded a constraint and to fulfill came a new language with specific set of constraints. Application size and complexity increased over time and started with various classifications like web, system, software, application, scripting etc.

It started with optimizing one problem with specific language and today we are confused on which one to pick to solve the problem. A web language is used for creating and editing web pages (HTML, XML, JavaScript) and software languages for creating executable programs (C, C++, Java). System languages are more concerned with managing system resources and application languages for building an application to solve a problem. Scripting languages are used for wiring together systems and applications at a very high level. This is followed by many named and unnamed categories and classifications.

There haven't remained today any major classifying criteria for the existing languages to put under a category. **Well, so why we call it as HYBRID.**



Story Five

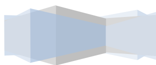
Abstractions

Okay, so far we have had some motivation and known a bit of evolution of programming languages. So, what next? We are going to broaden our thinking by understanding the abstractions. What it means by abstractions is that we want to get the idea and design but not the implementation.

Why we need to program? I am sure you will have answers of your own. By and large mostly it is to automate the things around us. We want machines to do a few things for us. We already know we are not going to deal with machine level languages. We will have to deal with higher level languages. We also know machine don't speak or understand the way we do. So, the languages we are going to study are going to stick to some protocols. How are they designed? What abstractions are lifted up from machine?

We are up to ourselves and at peace in real world. We have found out some best of what machines can understand. **What we need to do is pull up our thoughts so that we match with how machines will be able to do with it.** Yeah, they are the abstractions. I know things are not clear yet. Let's have a better understanding.

Let me put the summary so far once again. In colloquial terms programs are the stories told to the machines. But somehow because a machine cannot understand exactly what we speak, we tend to stick to something which is understandable. In the process of telling a machine what needs to be carried out, we have also constrained ourselves with our expectations. **As we cannot hangout with a machine and tell what exactly to do, we just hang a few things out of context and say, doing this would still do!**



Story Six

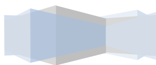
Look Around

Look around. Look around for what you see. Look for the same in what you will do. Let's dissect the parts for what we look and what we should look for. What we look is for the surrounding and what we should look for is for the programming languages.

We see quite a variety of things around us and they all have varying representations. We can count few things. We can only describe a few things. We can only approximate a few things. Some things are explained in long description and some can be done in one or two words. All such kind of things needs to be represented on a machine. We know all this is data and they all have types. Hence they are going to be called as 'data types'.

When you start studying a programming language, ask the question what kinds of data type's representations are available? C language provides the data type 'int' to represent the integers and 'char' to represent the character and strings. Java provides a 'boolean' data type to represent true/false values. Python provides a dictionary data type 'dict' for maintaining key-values pairs. In some languages you just use the data. The type is automatically associated by the machine. Almost every language also provides a mechanism for the user to create their own 'user defined data type' to suit the problem needs. They all vary according to language. We need to understand them because the right representation makes the operations simpler and faster.

Every data type is going to have its associated operation. For example, for integers, we can add, multiply and so on. Look at the world around us. We were never thought in school to add 'I will eat + I will go home', Right? Instead we did something



like ‘8 + 9’. Every data type will have its associated valid operations. So why we need to know the representation options available to us.

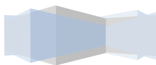
Every language will have a set of words which will be predefined. They will be called as keywords. We cannot use the keywords with our own meaning. We can only use them with the context as defined by the language. Look out for keywords. C programming language has 32 keywords.

Do you see how conditional we are? We always look out for options. We are in the middle of yes and no’s and a few maybes (or may be more). Should I go to market today or tomorrow? Should I visit doctor this weekend or next weekend? Will I qualify my driving test? Should I start exercising from today or tomorrow early morning? Should I propose her or not? Shoul I continue reading this book or stop? We all decide. We all are conditionals. Now, well it has to be a critical part of programming as well.

We have this ‘if - else’ in most and every programming language. This is how the structure mostly looks like:

```
if condition holds
        do this action
else
        do that action
```

If the condition holds true, we perform some action. Otherwise we have a backup plan. Doing nothing might be a backup plan as well. Inside ‘else’ we might also have one more ‘if-else’ and so on. Then we might have options to switch and opt



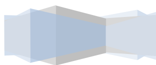
for what we like amongst all the available, break when in need, continue while in need and so on.

Our life is monotonous. We do similar kind of things in our everyday life. The tasks we do have repetitions in them. Let us say we ask a kid to count from 1 to 10. What he does is he starts from 1 and every time adds 1 to it and does the count. Like 1, then 1+1 = 2, then 2+1 = 3 and so on. The preparation of batter for Dosa has the process of grinding, which we carry on until the batter is grinded properly. Grinding is the repetitive process. While we stand in bus stop and wait for bus, the wait is the repetitive process. Here repetitive means doing it for some defined amount of time. Programming languages handle these kinds of statements using looping statement. Looping tasks can be achieved in several ways.

Consider an example where we are supposed to print all the number from 1 to 10, just like the counting example of kid. Let us see how to print numbers 1 to 10 using python. Needless to mention, this is not the only way.

```
for count in range(1,11):
    print (count)
```

Here 'range' is a function which is taking two inputs. It starts from 1 and goes till one less than 11. The 'count' is a counter which is keeping track of iteration. 'print' function is printing out the count. The count is first initialized to 1. Every time it gets incremented to 1 and prints it out. When count becomes 11 the loop stops which is the stopping condition and does not print it out.



Let us see how this is achieved in C programming language. Do not look into the details of the syntactic rules. They can be ignored and still understood.

```
for ( index = 1; index < 11; index++)
    printf(“ %d ”, index);
```

Here we have the variable ‘index’ which is doing the counting job. The printing is done by the ‘printf’ function. Index counter starts from 1 and prints till the condition ‘index < 11’ holds. ‘for’ is the keyword again doing the task like in python.

Let us see how the same task is achieved in scheme, a functional programming language.

```
(let loop ((n 1))
  (if (> n 10)
      '()
      ( cons n (loop (+ n 1)))))
```

Here loop is the name of the function which is doing the job. ‘n’ is initialized to 1 and it goes till 10. ‘cons’ is used to create a list. ‘loop’ creates a list containing the numbers 1 to 10 and displays it to the user.

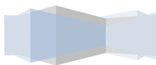
You might have noticed that the way task is achieved is same in all the three languages mentioned above. There is an initialization, there is an increment and then there is a condition checking. The counter moves from 1 to 10 and prints the numbers. What varies is the syntax in all the three languages. Rules are pretty much the same but the way in which it is achieved is different. Rules are derived according to how things happen in real world. Syntax is established based on the programming

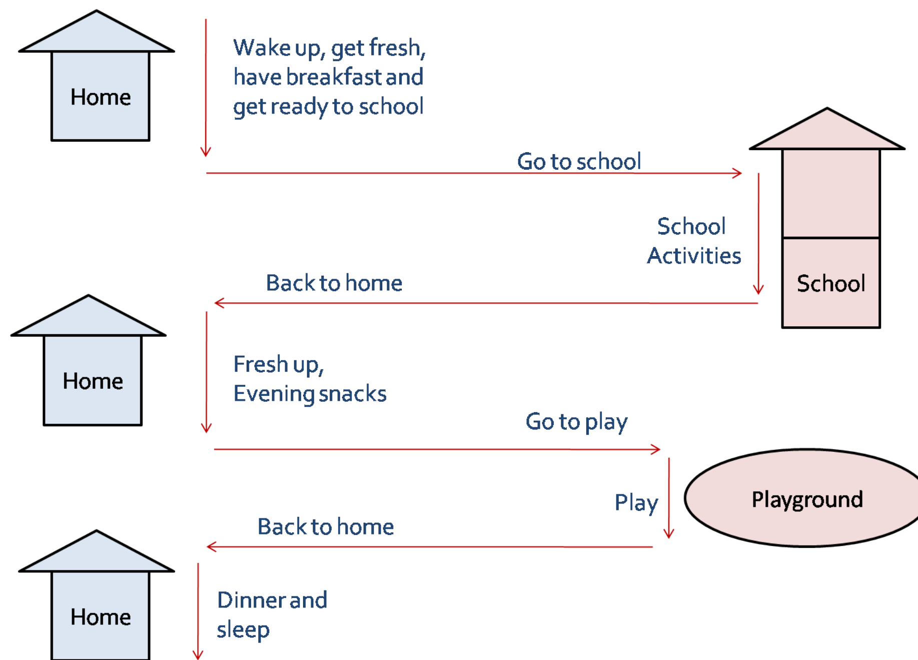
language protocols. The programming language designers have put their thoughts into developing the language according to the rules and protocols.

Pick up the syntax of any programming language and code the task, you can still correlate. Why? Let us take this discussion soon. What we have learnt here is that programming languages are understandable.

They are doable. They make sense. They can be learnt. They do are our cup of tea. They are the machine level correlations of what we see around. They have a little tweak for a machine to understand, which we can definitely understand.

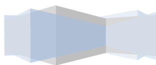
Let us consider one more scenario. Look at a school kids daily routines. The picture below put down all the activities. The activities of the kid begin from home. He moves to school and returns back home, then to the playground and back home. Everything begins from home. The idea is everything has to begin from somewhere, some initial point. Home is our nest, where we all return back. We do, birds do, animals do, in totality; everything has a start and a return point and so does programming.





Think if this was to be programmed, we would want something like home. Home is where everything begins. We give smaller sub tasks. Once they are achieved we return back to home again. This very much happens in the programming languages. The general convention used is the 'main' from where everything begins. The smaller tasks are generally called as procedures, methods or functions etc similar names.

Look around you. As we have taken the task of automating everything around, we need to discover the mapping. You get it right? **What programming does is it essentially maps the real world scenarios to Syntax and Protocols.** Understanding around us is part of understanding programming. Everything is related. Have we addressed everything in the programming? Sure NO! We just looked at some examples. They are just initiatives on how to look at programming.



Story Seven

All That Programming Has!

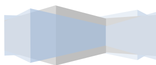
We now have a sufficient idea about what and what not's of programming languages. Let us look at those supports provided by most of the programming languages. We are getting a little technical here, appropriate enough to understand with known context. This part is like a summary of the previous story. Below is the list which most of the programming languages have support to.

Programming languages have data types basic and user defined to represent data. They might be identified with an assortment of names but they all boil down to the same purpose.

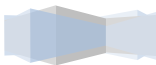
Programming languages support numerous kinds of operations like access operations, assignment operations, arithmetic operations, relational operations, logical operations etc. They are as much essential as we need them in real world.

Programming languages talk about control structures. Like the name says, its control structures AKA they give us the control. They include conditionals, to switch, to break, to continue, to iterate with looping and a little variants as on demand.

Programming languages define how the execution flows and how the sub tasks are carried out. They talk about context switching and scope of each module like how long the module is active and the resources needed to carry it out.



And then the list continues. Some domain specific features, some additional packages, some extra features, some specific to language etc. Like you now understand, yes, you can totally understand and code!



Story Eight

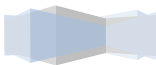
Getting Ready

Here are certain things you should do before you start learning a new programming language. It's almost same as learning a new communication language. What you do is you start asking questions; right? The more you speak the quick you learn! Things with programming are almost exactly the same.

Before you start learning, read the history of the language. Who invented it, why was it invented, what household applications are built with it, what is the principle behind it, does it have predecessors and successors, what paradigm does it follow and a related few.

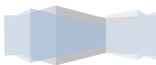
Once you know enough about, ask yourself few questions and if possible make a list of it. Write them down. The questions can be like, How does the execution begin? How do we test conditionals? How do we pass the resources from one function to another? How can we put comments inside the code? Which IDE is the best to code for the language? Etc. Reading the history of the language can motivate you to form a few questions.

Keep the manual or book with you so that you always have assistance while you code. **Remember, you cannot read and understand a language. You need to code and utilize the language**. The more you code, the better you get. Start from small and slowly make it to large. Don't think to build an application or a game on day one. Your flow should be like: first learn to print a message, to save the results, then to add two numbers, then other arithmetic operations and finally design the calculator. Because



in the mean while you will also be aware of the principles and you will know what would be the best way to achieve it.

Pick up any programming language; you will get a lot of support and codes on the internet. Feel free to use them to learn. You have done your best when you have contributed back something to the language!



Story Nine

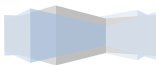
The Logic

The question you might be having in your mind by now would be something on the lines – 'Are all programming languages the same?'I will not answer that nor will we try to discover.

It's like, we speak so many languages and you might know a handful of them as well. Are they all same? They are kind of same and not. Yes, they all are means of communication and one would have picked the mother tongue and / or regional first and learnt a few others as well. Knowing one does not make easily know another. There is still learning needed to speak another. You understand it right? They are same in a way that they are not the same.

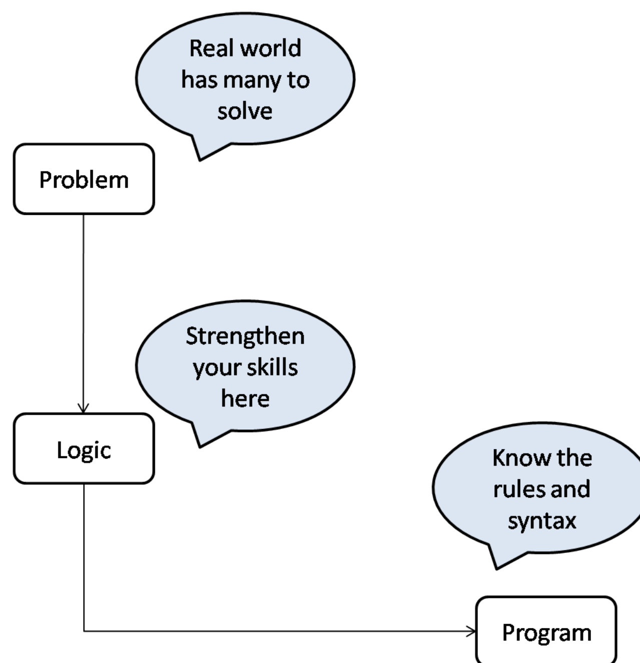
Similar are programming languages. One can easily get into this debate and never come out of it. There are both schools of thoughts that they all are same and they are not. People point out the differences and say that they are not. People point out the principles and say they are. Everything else rests on how you see it. No matter what, one must understand that the logic of the program is not constrained by the rules.

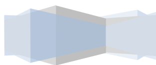
One is always free to think on how a problem could be solved. Consider the logic of the program of printing 1 to 10. We had an initialization statement; there was a condition check and an increment. The logic is the same with varying syntax for different programming languages. Now we cannot keep this as a base and say this is how it is going to be for all the cases. We just took a very basic example.



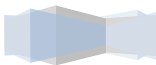
Programming is a tool as expressed by many. The inner strength of it is the logic. More and more practice makes one good in logic. Logic is thinking. **If we can think, we can code. If we can think better, we can code better!** Essentially, we all can think so we all can code. If we can think better, we can code better. When we repeatedly perform a task we tend to think a better means for it. We try to find a shortcut to make it quick. We try to improvise it and make it better. So is the logic. It gets better with practice. That's how you become a good programmer.

The programming language syntax also has an upshot on the logic. There might be some operations directly supported by the language making those easier and faster to code. They need not be remembered. They can be looked into manual while in need.





All in all, you have a problem and you think of logic. You then pick a language and code the task to solve the problem. Picture above puts the summary so far.



Story Ten
The Story of X

Okay. We know that we need to get better in logic. This happens in many ways. One is to understand the complete picture of the problem we are working. Understanding the scenarios is important in the programming context. One needs to understand the type of environment we are now working in. Let us consider an example program that we want to search for a value x. Look at the scenario clearly.

The problem was simple with a too simple solution. I wanted the value of 'x' and I ask for an 'x'. Either I get back the value of it or an answer 'unknown'. The scenario is not the same any more.

Why is it not the same?

Programmer now needs to handle way more than expected number of conditions. We ask for an 'x' and we get back the value or no value, is the routine thinking. Now added to that are few more things, which a programmers needs to exhaust while he obtains the result.

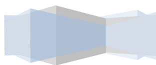
What if you are asked to wait? The value is being computed and there is a delay. Or it's coming over the network and there is a delay. Is your code ready to wait? If yes, for how long?

What if you get blocked? And result is never returned? Neither have you got an error status nor an exception. The request is blocked and there is no permission. Are you ready to throw a suitable message? And realize that you were blocked?

What if you go dangling? And get a 404 error. What if the computation to get ‘x’ was pushed somewhere else? What if you are not asking the right person for the value of x?

What if you have got the result with some disturbance? There could be a positive or negative deviation in the result obtained. Will you ask a few more times and take an average? If yes, how many times? Is there a threshold?

Do you see that all? There were so many questions for that one simple task. Understanding the problem and having the bigger picture makes you think of all the conditions one needs to handle. And yes, that’s good! Well, ‘x’ does not come that easy.



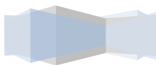
Story Eleven

Algorithms

Mr. Logic has yet another good name. He likes being called as 'Algorithm'. He is part of it. Algorithm is a bigger umbrella and Logic is the heart of it. We now know that computation is a process of unfolding the given problem. It is the process of identifying if the problem is solvable or not. In more concrete terms, as we speak to computer science graduates, it the question of "Can you write an algorithm for the given problem?"

Algorithms play a central role in both the science and practice of computing. They are sequence of unambiguous instructions for solving a problem. Precisely, algorithm is a logical, arithmetical or computational procedure that if correctly applied ensures the solution of a problem. An algorithm has to be lucid, precise and unambiguous and give the correct solution in all cases. More importantly it has to terminate after a finite numbers of steps aka the finiteness property.

An algorithms course is mandatory to be a good programmer. Does not mean a registered course in a university is mandatory. It must be like reading on one's own interest. It teaches you variety of techniques. That's how you gain the expertise knowledge. It teaches you how to be precise. It teaches you the computational procedure. It teaches you the techniques. Algorithms help you build your own interesting queries. Like the one I have. Read up the next story!



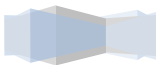
Story Twelve

Case Study: Queries and Questions

After a few years of programming I had several questions in my mind which led to quite an amount of research to check if anything was worked upon it. There were several ideas proposed but most of them have remained as theory. I have picked up two of them and presented in this section.

That's how we become a good programmer. We tend to ask questions and have our own answers. Algorithms will help us with a greater deal in all this. The objective to write this part is that we all should have questions. It is those kinds of questions where it's not easy to get the answers. We don't want to ask those questions to someone else. Instead we want ourselves to research for those answers.

The first one is about why can't we have time associated with programming? Second one is how can we prove correctness of the program?



Case 1: Time Please

We know that 'computation is a process of unfolding the given problem'. I always had this question, why is it not 'computation is a process of unfolding the given problem with respect to time'? Everything we do has the time dimension. We have seen hardware students doing it as well, but why not in software's? That is one serious question I have!

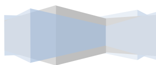
In other understandable term, we say:

"x" made a transition to "y" with input "u"

Instead, why don't we say:

"x" made a transition to "y" with input "u" at time "t"

Time is not in design. So it is not in program as well. But is it essential to have it? I see the answer for it as 'yes'. It could be a means to prove the correctness of a program. Time definitely needs to be associated with every step event in the program.



Case 2: The Invariants

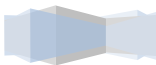
Though we list several desirable properties of an algorithm like simplicity, correctness, unambiguous etc there is no formal method established to verify these properties. Primary concern of every algorithm designer has to be indeed verifying correctness; only then later to establish efficiency parameters. In this regard, I felt programming with invariants is not a new paradigm but definitely the one every programmer needs to adapt. I have put down the procedure with an example explanation. I don't follow it every time strictly, but when I teach I prefer to do it this way.

Procedure:

1. **Understand** the problem.
2. Write few **examples** to verify your understanding.
3. Write the **state space**. State space is basically the domain of the problem. The type of data on which the processing is about to happen.
4. Identify the **transition function** which describes the various states the process can be in.
5. Write down the **traces** to authenticate the transition function
6. Identify the **invariant**. Invariant is property that proves the correctness of the algorithm. It is that one property that does not change in the algorithm.

Example:

1. Problem:



Given a list, find the length of the list. The task is to identify how many elements are present in the list. Let 'list-length' be a function to compute it. A list is represented with '( ) convention.

2. Examples:

```
(list-length '() )          --> 0 (Empty List)
(list-length '(1 2))        --> 2
(list-length '(3 5 6 7))    --> 4
```

3. State Space:

Here we identify the domain. We decide what the input and output domain is.

Input:

List of integers

Output:

Natural number which is the length of the list

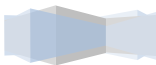
4. Transition Function:

We next write down the transitions. If list is empty the length is 0. If list has one item then the length is 1. If the list has more than one item we can count it one by one. We can pick one item at a time and increase the counter. The transition rules can be seen below:

```
list-length(n) = 0                    if n = 0        rule01
               = 1                    if n = 1        rule02
               = 1 + list-length(n-1) if n > 1        rule03
```

5. Trace:

Let us see examples on how the transition rules work.

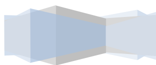


```
Example 01:
--------------------
list-length '()
= 0                             rule01


Example 02:
---------------------
list-length '(1 2 3)
= 1 + list-length '(2 3)           rule03
= 1 + 1 + list-length '(3)         rule03
= 1 + 1 + 1                        rule02
= 2 + 1
= 3
```

6. Invariant:

Observing the transition function and the trace we can conclude that the invariant for the process is: **sum + list-length (n-1)**. Reason being at any point of iteration, we can use the invariant and find the length of the list! By the way, identifying an invariant is not an easy task. It comes by practice. And this is how we prove the correctness of our program.



Story Thirteen

Every Language has a Story

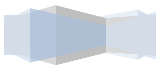
You don't read a romantic novel expecting horror story in it right? So are all the programming languages. They have some objectives and principles behind them. You can't expect anything beyond for what it was designed.

PHP is a hyper text preprocessing language. PERL is for extraction and reporting. R is mainly for data analysis. C attempts to be ideal for systems programming. PYTHON is an ideal scripting language. HASKELL is a pure mathematical language. HTML is a markup language used to create web pages. The story goes on for every language. Don't pick a language and use it for what it is not supposed to be. Every language has a dedicated audience.

'Awk' was supposed to be a command. The power it had was more for just a command. It turned to be a language on its own. So is the story with 'sed'. 'Prolog' is a logic programming language. APL introduced array programming. Well, they all have stories.

Did you know there is a programming language called Ook!?? It has only three syntax elements: OoK. Ook! and Ook?. LOLCODE is also a programming language. 'Whenever' you feel like, wait, 'Whenever' is also a programming language. 'Chicken' and 'Chef' are languages as well. Think of any weird word and you might find a language with that name. If not yet, you could be the inventor!

If you feel this is all a Brainfuck, there is a programming language called 'Brainfuck' as well. 'OMGROFL!' now that's also a language.



Story Fourteen
Literate Programming

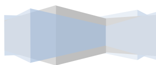
If you really want to get the story of the code you write, follow literate programming for the initial few days. There is a high probability that you might continue once you are used to it which is obviously a very good programming practice. Literate programming was introduced by Donald Knuth. His idea was to write a program as the thoughts flow. However the syntax of the program hinders that thought process.

Knuth suggested writing the story of the code as the thoughts flow and embedding suitable code snippets in it. You get the whole program if you pull out only the source code from the story. Let us see a simple example of how to do literate programming for adding two numbers in C programming language.

#+Title: Program to Add Two Integer Numbers
#+Author: Prakash B H
#+Email: prakash.hegade@gmail.com
#+Date: 31 May 2016

*** Introduction**
This document explains how to write a program to add two numbers using C programming language. The program is written using the literate programming technique. The documentation is carried out using the =*emacs*= editor in =*org*= mode. The code can be extracted from the document using the =*tangle*= commands.

*** Program to Add Two Numbers**



We begin the program by including the necessary headers which will bring together all the required libraries.

#+BEGIN_SRC C

```
#include <stdio.h>
#include <stdlib.h>
```

#+END_SRC

We next write a module to add two numbers. We pass the two numbers as parameters to the method. We perform the addition operation on the supplied two numbers and return the result.

#+BEGIN_SRC C

```
int add_numbers(int num1, int num2)
{
   int result = 0;
   result  = num1 + num2;
   return result;
}
```

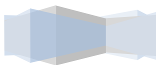
#+END_SRC

The next task is to write the main function which initializes the required variables and calls the function to add the two numbers. Let us put down the initialization part.

#+BEGIN_SRC C

```
int main()
{
  int num1 = 10;
  int num2 = 10;
  int result = 0;
```

#+END_SRC



Next we call the function where the variable =*result*= captures the result.

#+BEGIN_SRC C

```
  result = add_numbers(num1, num2);
```

#+END_SRC

We then finally print the =*result*= to check if the operation was performed correct.

#+BEGIN_SRC C

```
  printf("The result of addition is %d\n" result);
```

#+END_SRC

Closing of the main function

#+BEGIN_SRC C

```
  return 0;
}
```

#+END_SRC

* Conclusion

The story above presented a C program to add two integer numbers.

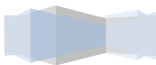
When we tangle this story, what we get is everything and only those which are embedded between

#+BEGIN_SRC C

...

#+END_SRC

The 'C' tells that it is a C program. We then have our C program ready to compile and run. By writing the story we actually capture all the details of the program. Following this method, needless to say the readability and maintainability of the code increases. You can give this method a try!



Story Fifteen

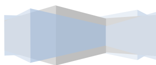
Why Program!!?

The decisive goal of each programming language is to keep it effortless for the user and conceal the complexity in the implementation. The intact notion of theory in computer science verbalizes the abstraction by keeping it easy to the user. Concept is goal driven to optimize it to the computer architecture.

We all speak of domain specific languages today. Why is not my language customizable? It should be as small as possible which can grow as large as possible. User bears in mind very few details and grows the language in his own terms. Why there is a restriction that execution has to begin with "main"? Why is writing an interpreter for my required domain so complex? Is it that the limitations of a language are covered up by adding more features than providing the abstraction?

We never realized that the productivity from technology did come with side effects. Look at this arresting instance of a telephone company customer care. Earlier people used to sit at customer care and handle the customer queries. Technology carved in and the process got automated. Now we snoop to automated voice and elect to choose for the required service. Look at the process in monetary terms. Earlier company used to pay for person who used to sit at customer service. Now we pay for using the service and listening to automated voice. Where is the benefit heading?

Why do most applications fail to reach the audience? Too many features have only made the process slower. Some applications are in use, only because we are mandated to use it or there is no enhanced version out yet.



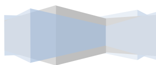
Software's have created inter-dependencies. I need to use some product only because my system does not support the format. The mandatory usage to one makes his connected also to depend and bring into play the same.

At times, they come as a solution to a problem and thereby introduce many other tribulations. Once when we are into the system, we get dependent and drive the tradition. They come with a never ending manual documenting what doesn't work; as the limitation of the product and never say that they were the shortcomings. We would have never purchased a washing machine if the vendor told us he would deliver the components in installments.

Yes, technology choices make a huge difference. I don't intend to run a strike over **software usage**. All the **world changing** stuff seem like they need a polishing. What we need are better interactive systems. Oh yeah! We need better engineers!

What I mean by better engineers is the better designer, better inventor, better programmer and many other betters. That's why you should learn programming. That's why you should hop in. There is a lot to be discovered yet. The computing society needs your talent and contribution.

That's why you should program!



Story Sixteen

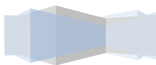
Experience Bucket

This section is an experience shared by engineer Vishwanath Telsang, who currently works in SAP, Bangalore. Here he goes:

Programming and Me

Like it happens to the most, the perception of programming has varied to me with time. In the beginning, with even no proper understanding of print and read statements, I used to feel like a machine for myself. It almost took me a month to understand the procedures to read and write. They all seemed like hell and I did not wanted to live in there. And when this happens, commonly, you start to lose interest over things which you do not understand and eventually it ends up in hating them, which is what exactly happened with me. I started to hate programming and it later turned into hating the professor who used to teach programming.

I had this weird way of coding. I used to prefer print statements where ever possible as I was comfortable using them than over the looping, conditional and other kind of statements. Let me explain that with an example. We had to print this following pattern in a C program:



```
*
*    *
*    *    *
*    *    *    *
```

So, I used to write this program like this

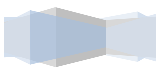
```
Printf("*\n");
Printf("*\t*\n");
Printf("*\t*\t*\n");
Printf("*\t*\t*\t*\n");
```

And when I got nothing from this solution in exam, **I realized programming is not about getting an answer or solution, it's about getting it the right way.** After this I felt that it's time to understand those looping statements. It took time to understand things but it was not difficult once I started looking at them in the right way.

Programming, Then and Now

At first it was all about getting an output for the given problem statement, I would not think much upon logic or optimized code; it was just about getting an answer. Many a times even I used to get confused of how this code worked, because it used to be like blind coded. Even after completing my engineering I was not that comfortable with programming, I had passed programming subjects like java, c++, data structure with passing marks.

But now, I feel like I have good domain knowledge about all these subjects and I feel confident. Given a problem I can think over it, visualize code inside my mind and



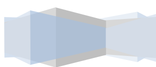
think of different ways to solve the problem. There is now more of confidence and it feels like programming is an art.

When I Started Doing It the Right Way

I was recruited in my final year of engineering and I was happy with the offer with pretty decent salary. When I got an interview call from this company which is a leader in domain of enterprise resource management products, I was curious and confused. I was curious to face it and confused if I will be able to crack it.

The interview process had 4 rounds - 2 technical, 1 HR and 1 managerial. I was more challenged looking at the line up for the company. For few days I did not think much about interview as it was my last days of my engineering and every time when I thought of it, I felt like it's a space mission to Jupiter.

Soon after completion of engineering as I moved home, for the first ten days I did not think about anything. I had two months left for interview and three months left to join the company I was recruited for. Then I felt why should not I check what interview is all about and got to know that I can manage if I prepare for java and data structures. I thought I will read them again from scratch. I started with some good old notes and textbooks, first few days were same I couldn't understand much and I felt this is some kind of alien language. Later I said let's type and see. I started practicing each and every program; I mean I started typing them out. After few days I felt confident, it was just as practicing mathematics for me. Then I discussed things with myself, I started asking 'WHAT IF' questions. These gave me ability to solve problems in different way. That's how I started understanding things and started doing them in right way.

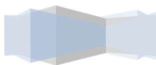


Me and Programming

"**Be a slow learner there is nothing wrong in that, but always try to learn things correctly**".

This is how I was able to understand programming concepts and once I understood it I was able to give solutions to them. Even today I get confused with concepts but it takes hardly couple of minutes to recall those concepts.

"**Everybody has their own way of learning things and no one knows that better than thyself, learn things in that way. You will sure enjoy things then"**.



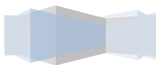
Story Seventeen

Conclusions and the Beginnings

We hereby conclude this discussion and wishing the best for the programmer's journey. Trust me. It's crazy happiness once you start to code. It will drive you maniac until to finish your code and shout it out. It has to be experienced. One needs to enjoy that little secret dance in the room when the code compiles with no errors and warnings.

It is happiness to know what programming can do. Once when you start contributing, your perception to look into the world will change for better good. Look! There is a programming book jumping out to reach your hands! Go, grab it!

"Every Programmer has a design on how he was created. Every Programmer has his own design on how he was created! "



Author

Prakash B Hegade
Academician, Blogger and Researcher.

Thank you for reaching out till the end section. I hope the stories in the book were readable and worth the time.

It feels good to write things and when someone reads it, it's a motivation to do better. Programming is one my passions and when it comes to C, it's always the priority. This book was in my thoughts from a very long time and I was happy when it turned into one. Smiles.

Thank you.

~~*~*~*~*~*~*~*