Programming for the Lazy

Robert W. Harrison
Department of Computer Science
Georgia State University

2014

Preface

This book is a language-agnostic approach to learning how to program. It was inspired, if that is the correct term, by the difficulties exhibited by undergraduates taking the junior-level system programming class at Georgia State University. It also stands on the shoulders of giants in the field [1, 2].

Only about one third of those students who had bothered to take (and pass) the prerequisite class in object-oriented programming could actually program when they took the class. This was awful, but predictable. No one had taught the other two-thirds of the class "how to program", even though they had mastered elements of the Java language. It was as if they had been taught "I am", "able", and "to program", but not how to string the elements together to produce the sentence "I am able to program".

This book is about how to program, and not about any language in particular. It is about how to think, to take a problem apart into its component parts and write a solution to it.

The book is aimed at intermediate-level students, who have had instruction in one, maybe two computer languages and a smattering of theory. Its goal is to help them integrate that information and become effective programmers so that they can continue to be successful in computer science.

How Not to use this Book

Since students tend to not pay attention when told to do something, maybe telling them what not to do will work.

First The book contains practice problems. They've been picked rather arbitrarily because I think they're fun and/or illustrate some point in computer science that is worthwhile. If you don't try them then you will not use the book properly. *DO THE PROBLEMS.*

Second Using Google or some other web searching engine to find some program that more or less is similar to the program you want to write and then fiddling with that code is a guarantied way to fail. It's also really inefficient so you can tell your instructor that you've been working on the problem for days and days without success. *DO YOUR OWN WORK.*

Third Waiting for the last minute to try the problems is a great idea if you don't want to succeed. After all there is nothing like the threat of a deadline to concentrate the mind. *ALLOW YOURSELF TIME TO DO THE WORK.*

Fourth Don't read the book or look at the reference material. This is another excellent strategy for failure. *READ THE BOOK, LOOK UP THE REFERENCES.*

Fifth Don't ask any questions. Asking questions is just an admission of mental inferiority. *ASK QUESTIONS (in class or online).*

Contents

Chapter 1

Introduction

Programming is inherently a process of building a model. It can be a model of a process which leads to imperative programming or a model of data which leads to object-oriented programming or even a model of a mathematical functions or statements which leads to functional programming. In addition to being a model, programming can be modeled and analyzed mathematically. While it is not absolutely critical for learning how to program, having a basic understanding of models of computation is useful. The following discussion is, per force, simplified and a more detailed treatment can be found in [] or [].

Models of programming and programmable machines

It is worth quickly and, somewhat superficially, examining models of programming and programmable machines. It is possible to skip this section for a later reading, but eventually a thorough understanding of this material will be useful.

Imperative, machine-like or Turing Models

Functional, λ-calculus, or Church Models

Church-Turing Equivalence

The Church-Turing thesis [] states that these two models of computing are equivalent. Informally, this is easy to see - the Turing model has an infinite tape or memory that can represent almost anything but is otherwise a finite (though arbitrarily large) state machine. The Church model has a set of functions with a potentially infinite number of arguments, but where the functions themselves do not store state information. The are basically orthogonal copies of each other - the tape and machine state in the Turing machine is the list of arguments in the Church model. The Church-Turing these posits that there is a one to one

equivalence between the two representations and thus what is computable in one model is computable in the other. Putting this into a rigorous setting is a little more difficult.

It must be strongly pointed out that Turing machines, and their λ-functional equivalents are not complete models of all possible calculations nor are they particularly accurate models of modern computers[]. They describe conditions under which a large class of functions can be computed. One important area where they are inadequate is "interactive computing" where the program responds to apparently random - or at least not predetermined - stimuli[]. There are generalizations of the Turing model that can be used for these cases. That said, thinking in terms of the state of the machine, what each section of code[1] does and what the state of the machine is after executing that section of code is an effective way to analyze imperative programming.

Elements of a computing machine

A computer program consists of an ordered set of instructions[2]. The instructions instruct or tell the computer what to do. The computer has the ability to step through the set of instructions in order, and more importantly to move to a new location in the ordered set based on the results of a computation. The computer itself also has memory - it can remember or store values and the recall them later as needed for other computations.

A computing machine has to have the elements necessary to execute such a program.

Memory In order to save state, values from computations or "data" the machine has to have some form of memory. The tape in the Turing model is the memory. In the Church model this is the arguments to the functions. The sequence of instructions is usually held in memory too - though it may be in a different "segment" on some real machines. There has to be a mechanism for putting data into memory and recalling data from memory.

Order Control Since the steps are in an order, there has to be part of the machine that controls or supports the ordering.

Arithmetic Unit Since the computer typically does things to data, it needs to have a mechanism to perform those calculations.

Logical Operations Logical operations, the ability to ask questions about the data, are critical in a computing machine. Even more critically, the results of the logical operations can be used to alter the Order Control and change the order of the instructions that the machine uses.

[1] The idea of a section of code is surprisingly powerful - many code analysis tools use a "basic block" analysis to describe and optimize programs[] where a "basic block" is a section of code with a single entry and exit point.

[2] in parallel or concurrent programming a program consists of sets of ordered sets of instructions with the addition of sets of instructions to enable communication between the individual parts of the program

Logic vs. Syntax

Syntax is the form of the language or the grammar and constructions that both help the programmer express what they mean and the compiler or interpreter understand what they are saying. Syntax describes how the steps in the algorithm are defined to the computer.

Logic , in a programming context, is the order and composition of statements and operations to implement a given algorithm. Logic describes the order of the steps in the algorithm and how the steps depend on each other.

Beginning programmers make both syntax and logic errors. They also tend to not differentiate between the kinds of errors, which doesn't sound too bad until you realize that there is almost no way to correct a logic error with syntax, and almost no way to fix a syntax error with logic. The following Java and Python programs, both intended to write "hello world" to the user are good examples. They are syntactically correct (have correct syntax and can be compiled or interpreted), but just won't work.

The Java program:

```
public class hello{
public static void main( String ac[])
{
System.exit(0);
System.out.println("hello world");
}
}
```

and in Python (where it is especially obvious):

```
exit(0)
print "hello world"
```

There's sort of an obvious error in these codes, isn't there? But they both compile (or interpret) without errors. The *syntax* is correct, but the *logic* is wrong. [3]

Exercise 0

Fix the logic error in the programs so that they correctly print hello world.

[3]This may seem to be an obvious error. It isn't to a beginning programmer. I used the exact same error for an easy question in a mid-term (it was in *awk*, but the students were supposed to know that language), after it had been sent to me by a student with one of those "why won't my code run?" questions. Most of the class could not find it.

Chapter 2

Specifying the Problem and the Solution

Specifying a problem and how you solve it seems to be obvious. It isn't and most beginning programmers make mistakes at this stage that prevent them from solving the problem. It is very difficult to write a correct, functioning program if you don't know what the program is supposed to do. Laziness is often a virtue in computer science, but laziness at the design stage resulting in an unclear or incorrect formulation of the problem leads to many hours[1] of frantic and fevered effort. So this is one case where laziness is definitely not a virtue. Sloppy thinking seldom pays off.

What's an Algorithm?

At its simplest, an algorithm is an ordered sequence of effectively computable steps that terminates and returns an answer. That's a bit trite, though, and a slightly more detailed discussion is worth pursuing.

Ordered Steps

An algorithm is a ordered sequence of steps. Some things happen first, some second and some last. Because it is ordered an algorithm, and the computing machine that implements it can move forwards and backwards in the location. It can repeat itself in a loop.

Imperative languages like C, Java, Fortran, Basic, and many more mirror this order by having the sequence of steps as an explicit construction in the language. (i.e. do this statement, then this one, then ...). For example, in Python a code fragment to count the number of lines in an input file could be written:

[1] if not days, weeks, months or years

```
# open a file
input = open("somefile.name","r")
# set a counter to zero
counter = 0
# read all the lines in the file and increment the counter
# at each line
for line in input:
    counter = counter + 1

print counter
```

Note that the order of the lines of code matters - almost all of the time (only the order of opening the file and setting the counter to zero can be swapped.)[2]. Replacing this code with:

```
# open a file
input = open("somefile.name","r")

print counter
# read all the lines in the file and increment the counter
# at each line
for line in input:
    counter = counter + 1

# set a counter to zero
counter = 0
```

Would result in a very different (and rather silly) result.

Functional languages like Haskell, Ocaml, and Erlang tend to have an implicit ordering which can be a bit more confusing (though they can have explicit ordering in the code for the functions). A function will often have a set of patterns to match, and then respond based on which pattern is matched[3].

The same algorithm for line counting, expressed in Erlang:

```
% these lines are needed to make the program run
% basically they define what the program is called
% and howmany arguments are used when it is called
-module( line_count).
% it expects the file name
-export( [line_count/1] ).
```

[2]Don't worry if you don't understand the details of the syntax. You will understand it by the end of the book. For the Pythonista's - yes there are other, better, ways to do this - they just aren't as clear.

[3]Don't get upset with the logic here. If it doesn't make sense at first, come back after reading the chapter on recursion.

```
line_count(Name) ->
 case file:open(Name,[read]) of
  {ok,Afile} -> counter( Afile, 0);
  {error,Reason} -> {error,Reason}
  end.

% the counter function recursively counts the lines
counter( Afile, Sum) ->
    case io:get_line(Afile, "") of
         eof ->  Sum;
         _Data -> counter( Afile, Sum+1 )
    end.
```

The order which the functions are defined is unimportant,and most of the time the order of the patterns matched in the case ... end is not critical either. (The pattern _Data matches eof, so the pattern eof has to be first)[4].

The code:

```
% these lines are needed to make the program run
% basically they define what the program is called
% and howmany arguments are used when it is called
-module( line_count).
% it expects the file name
-export( [line_count/1] ).

% the counter function recursively counts the lines
counter( Afile, Sum) ->
    case io:get_line(Afile, "") of
         eof ->  Sum;
         _Data -> counter( Afile, Sum+1 )
    end.
% and its caller
line_count(Name) ->
 case file:open(Name,[read]) of
   {error,Reason} -> {error,Reason};
   {ok,Afile} -> counter( Afile, 0)
  end.
```

is exactly equivalent to the first example.

Independent from the choice of language, it is critical to know how the steps in the algorithm are ordered and ensure that the order of steps in the program mirrors that in the algorithm.

[4]It might be asked why Erlang and not Ocaml or Haskell for this example. The short answer is that in Haskell this would be done with a do statement in the IO monad and in Ocaml with a while loop and an input stream. Both the IO monad and the while loop are effectively imperative programs while the Erlang example is a purely functional example

Effectively Computable Steps

Each step of an algorithm has to be effectively computable. That's a fancy way of saying that you have to know how to do each step. Finally, one of the steps has to be "stop and return an answer".[5]

Computable steps fall into several categories. Obviously, arithmetic is computable. Computers know how to add, subtract, multiply and divide. With few exceptions, though, they work in a field of binary numbers. So there are other operations, specific to the representation of numbers that are important. And, Or, Exclusive Or (Xor), Mod, left shift and right shift are just a few examples. These will be represented in different manners by different languages. Just to make things difficult sometimes the same symbol is used in two different languages to mean different things. The symbol '^ ' is a classic example as it can mean exponentiation (2^ 3 is 8) or it can mean bitwise Xor (2^ 3 is 1) - you have to know what the language you are using means by the symbol.

In addition to computations, computable operations include logical operations. Computers know how to ask if two things are equal or if one is greater or less than another. Based on that result they can then move to another part of the program

Finally, there has to be some method of getting data into and out of a computer. So another class of computable steps are input and output operations (I/O for short). These can be conventional things, like letters and numbers, but this category includes things like the computer network or sensors like the temperature or Oxygen sensors in a car engine.

Compositions of effectively computable steps are also effectively computable.

Exercise 1

Which of the following things are effectively computable? Why or why not?

1. $(1 + 2) * 3$

How to Describe an Algorithm or a Program

It should be obvious that clear description of an algorithm[6] is a critical step for writing a correct program. Simply look at the word count example programs a few pages back. The Python example was a lot easier to understand than the Erlang example, wasn't it? You could probably understand it from the code and the comments even if you didn't know the language.

So clearly, we need a way to describe an algorithm that clear and independent from the computer language and, as far as possible, from the programming model.

[5] note that I didn't say correct answer, an algorithm can return "sorry there's an error" and be perfectly correct.

[6] algorithm, data structures, and software module/object structure in the general case

Pseudocode refers to a style of writing the steps of an algorithm in a clear human-readable fashion. There are a number of styles which people use, but the basic aim is to write clearly. Pseudocode is *not* a formal language, like C, Python, Lisp or Java. On the other hand there are some generally recognized conventions and elements of good form.

- Give it a name
  Naming a procedure or algorithm allows you to reuse it later without reproducing it in full and glorious detail. It's a whole lot easier to say "use the merge sort algorithm defined above" than to reproduce the algorithm in the context of your current site.

- Describe the inputs
  It should be clear what the data are, what is required and in what form it needs to be.

- Describe any assumptions
  Hidden assumptions about input, data, or properties of the operations described in the algorithm are an easy way to write invalid pseudocode. Make sure the reader understands the context of the algorithm.

- Keep loops simple
  Use simple imperative loops like for and while. You can say things like "for every line in the input", and generally should use short sentences to describe the flow of the algorithm. Similarly using "if ... then" and "if ... then otherwise" or "if ... then else" are a good way to describe conditional logic. Similarly structured programming concepts like break and continue can be very useful. Goto's are probably not a good idea.

- Compartmentalize
  Just like functions, subroutines, or methods in computer languages, if the algorithm breaks down into groups of steps or stages, use those stages to describe it. Concepts like "call" or "use" are good ideas. You did remember to give the procedure a name, didn't you?

- Entry and Exit points should be clear
  It should be possible to start at the beginning, and follow the algorithm to the end without confusion.

- Avoid extraneous details
  Don't put in details that are specific to a language

  ```
  for each character in the line
  ```

  is much clearer than

  ```
  for i = 0 to the end of the line
   c[i] ...
  ```

The Importance of Good Design

It's been my experience that sometimes programs seem to just write themselves, and sometimes it is a bit of a slog. The difference is the quality of the design. When it's been carefully thought out so that the individual tasks a simple and discrete then the code itself is relatively easy to write. If the tasks are convoluted and poorly coherent, then the code is very difficult. Spending time on design is usually[7] rewarded with faster and better code.

[7]Don't overdo it! *Analysis paralysis* can be a problem when you spend too long trying to come up with the optimal solution in the absence of any data. If you can't think of the answer, write something, it may be sub-optimal, but you'll find that out.

Chapter 3

Comments and Documentation

A program is written for two audiences. One audience is the complier or interpreter that turns the program statements into actions and actually does the work. The other audience is other programmers. Other programmers can include yourself after a short period of time. It is important the code be clear and understandable, that it is well documented, and that it able to be modified. Out of the two audiences, the compiler is the easiest one to satisfy.

Comments

Comments are the first line of defense in the battle against obscure code. Code without comments can be considered "cursed". Therefore every computer language has some way to write comments. Typically some reserved symbol is used for the first non-blank character of the line, or sometimes the first character of the line (C for a comment in FORTRAN). Java, C, C++ and similar languages use \\ for line comments and $\backslash * \ldots * \backslash$ for block comments. The \\ can occur anywhere in a line so lines like:

```
i = magic_function(x) ; \\ magic_function finds the hash of x
i = 1000 ; \\ estimated by trial and error
```

are common. Python and many interpreted languages use # as a comment - used once per line. Block comments in Python are also possible using a triple quoted string (e.g. "' stuff "') which ignored unless they are assigned to a variable - of course this assumes that you're not using a triple quote within another triple quote. Other languages use % or even ; (which is especially common in assembly languages).

Knowing what to say with comments is probably more important than knowing the mechanics of commenting. Comments which simply echo the code are largely useless and hard to maintain. In the word count example:

```
# open a file
input = open("somefile.name","r")
# set a counter to zero
counter = 0
# read all the lines in the file and increment the counter
# at each line
for line in input:
    counter = counter + 1

print counter
```

the comments that say "open a file" and "set a counter to zero" are redundant and should be replaced. (Those comments were deliberately written to echo the code so that the initial example program would be clearer to non-python programmers.)

```
# The initial steps open a file and
# prepare to count the lines by setting the counting variable to zero
# TODO replace this hard-coded file name with one the user can specify
input = open("somefile.name","r")
counter = 0

# read all the lines in the file and increment the counter
# at each line
for line in input:
    counter = counter + 1

print counter
```

The comments now say what the code does at a higher level than the statements. The code, which implements the algorithm, can now be changed without changing the comments. Note the use of a "TODO" comment - it was left to tell the next programmer about a known inefficiency or short cut in the code.

Javadoc, Doxygen and literate Programming

Donald Knuth introduced a concept called "literate programming"[3] which is the idea that the comments, documentation and program should be woven into one document which is then interpreted in different ways. One tool would extract a learned paper from the source, another software manuals and user guides information, and a third actually compile the program. While rather ambitious, and therefor not that commonly used, this idea has spawned a number of more common and limited tools.

Javadoc[] and Doxygen[] are two of the more common semi-literate programming tools.

Intelligent Choice of Variable Names

Syntactical "Sugar"

Syntactical "Sugar" are program statements that are needed to tell the compiler or interpreter things about the code that do not describe what the program does. Instead they tell to compiler or interpreter how to implement the code as a program. They are formulaic "magic incantations" that are needed to make things work. Compare a Java program for hello world:

```
public class hello{
public static void main( String ac[])
{
System.out.println("hello world");
}
}
```

with the same thing in Erlang

```
-module(hello).
-export([speak/0]).

speak() -> io:fwrite("hello world\n").
```

in C

```
#include <stdio.h>
int main()
{
  printf("hello world\n");
}
```

and in Python

```
print "hello world"
```

There is a lot more verbiage in the Java code than in the other examples. Almost all of it doesn't say anything about what the program does. It does say, however, that there is a class hello which has a method main, that takes some arguments but doesn't return anything. Invoking hello would invoke the main function, which in turn looks up the System class method println and prints "hello world". Erlang has a bit less(-module(hello). and -export([speak/0]).), which tells the Erlang run time system that this module is called hello and has a method called speak that takes no arguments and does something. C has some sweetness (#include <stdio.h> and int main()), while Python essentially has none.

There is a reason for this. The designers of Java and Erlang assumed that individual components of the program will be written separately and combined

largely automatically by the programming system. Therefore it was necessary to provide hints in the program for how to do this. Java tends to use verbose hints, as befits its object-oriented design, while Erlang's hints are sparser and more like functions - reflecting its function design. C is somewhat intermediate in design. C programs are expected to be made of multiple parts, but the programmer is expected to include references to libraries (#include <stdio.h>) and provide hints about how to combine this code with other parts (int main()). At run-time the C program starts with a call to main() and so that has to be defined at least and no more than once in the C source code. Python programs can also be made of many parts (packages in the Python lingo), but here the programmer is expected to resolve the packages with little automatic support.

It's important to get the syntactical sugar correct when programming - after all the program won't compile or run without it. Syntactical sugar is a major headache when converting programs between different operating systems or versions of languages. It's also very useful to be able to distinguish between syntactical sugar and substantive or meaningful lines when reading the code.

Chapter 4

First Steps
- variables and operations

Representation of numbers

Pointers, Memory, and Hardware

Common Operations

C operator	meaning
++	increment
- -	decrement (subtract)
%n	remainder modulus n

Logical and Bitwise Operations

It is important to know how logical operations are expressed because the ability to make decisions is one of the things which makes a computer a computer[1] Bitwise operations tend to be used less often, but when they are needed, for example when manipulating bit-masks for operating system or network programming or in various steps in encryption, they are essential. The C language representations are shown below, because they are used by many other languages, including C++, Java and Python.

[1] So important that there is an entire chapter on it

C operator	meaning
&	bitwise and (applies to the bits)
&&	logical and (applies to the whole word)
|	bitwise or
||	logical or
!	logical negation
^	xor (bitwise only)
<<	shift left 1 bit
<<n	shift left n bits
>>	shift right
>>n	shift right n bits

These operations usually map quite closely to the instructions on the hardware of the CPU. With the X86 family of processors, for example, the instructions SAL and SAR are exactly the same as << and >>. Similarly XOR is ^ OR is || and NOT is !.

Operator Precedence

Chapter 5

Logical tests, branching and control of the program flow

Chapter 6

Iteration

Chapter 7

Connecting to the Outside

Chapter 8

Debugging and Testing

Chapter 9

Methods, Functions, and Related Ideas

Chapter 10

Recursion and Other Ways to Iterate

Chapter 11

Interruptions and Errors

Chapter 12

Memory control
- Dynamic Memory
- the Stack
- Garbage Collection

Chapter 13

Managing Complicated Code
- Multiple Source Modules
- Version Control

Chapter 14

Libraries, Mathematics, and System Calls - Leveraging other people's work

Chapter 15

Extending the Language Structures and Objects

Chapter 16

More on Objects

Chapter 17

Serious Testing and Quality Assurance

Chapter 18

Data Structures

Chapter 19

More Fun with the System

Appendix - Binary and other number systems

How high can you count with your fingers? - assuming you can only make discrete moves and don't write down intermediate results.

Index

Bibliography

[1] B.W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley professional computing series. Addison-Wesley, 1999.

[2] B.W. Kernighan and P.J. Plauger. *Software tools*. Addison-Wesley Pub. Co., 1976.

[3] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.