Notes on Languages

Philippe Pittoli

ABSTRACT

Computer languages are a mess. Tons of languages exist and are intended for general purpose. This document is my take on computer languages in general: their use, what to expect, what to avoid, some advice. **You're welcome.**

Check out for newer versions: `https://t.karchnu.fr/doc/notes-on-languages.pdf`
And if you have questions: `karchnu@karchnu.fr`

Lastly compiled the **9/10/2022** (day/month/year, you know, like in any sane civilization).
Status: WIP

1. Principles

Basics of the Unix Philosophy in a few points (Raymond, 2003).

- Modularity: write simple parts connected by clean interfaces.
- Clarity: clarity is better than cleverness.
- Composition: design programs to be connected with other programs.
- Separation: separate policy from mechanism; separate interfaces from engines.
- Simplicity: design for simplicity; add complexity only where you must.
- Parsimony: write a big program only when it is clear by demonstration that nothing else will do.
- Transparency: design for visibility to make inspection and debugging easier.
- Robustness: robustness is the child of transparency and simplicity.
- Representation: fold knowledge into data, so program logic can be stupid and robust.
- Least Surprise: In interface design, always do the least surprising thing.
- Silence: when a program has nothing surprising to say, it should say nothing.
- Repair: repair what you can — but when you must fail, fail noisily and as soon as possible.
- Economy: programmer time is expensive; conserve it in preference to machine time.
- Generation: avoid hand-hacking; write programs to write programs when you can.
- Optimization: prototype before polishing. Get it working before you optimize it.
- Diversity: distrust all claims for one true way.
- Extensibility: design for the future, because it will be here sooner than you think.

These principles are still relevant and will continue to be: they represent good programming practices in general. There is nothing related to a specific language or environment. Follow this philosophy as much as you can, it will pay.

A few citations I like about *doing the right thing*.
(Raymond, 2003)

> *You have to believe that software design is a craft worth all the intelligence, creativity, and passion you can muster.*

> *Software design and implementation should be a joyous art, a kind of high-level play.*

2. Assembly

Assembly language is the human form of what is (almost[1]) exactly performed by the hardware. The language is driven by the hardware and all its particularities in term of memory management, available instructions in the CPU, etc. What isn't coded in assembly is (most likely) hardwired,

1. Sometimes it gets a bit more complicated than that. On some CPUs, the assembly code is rewritten by an internal operating system. But that's mostly details about specific hardware, this is not important to understand what assembly is. Moving on.

mechanical, electronic. This means the code really is near the hardware: you can manage the memory up to very narrow details (such as handling CPU registers explicitly) or use very specific instructions only your CPU has.

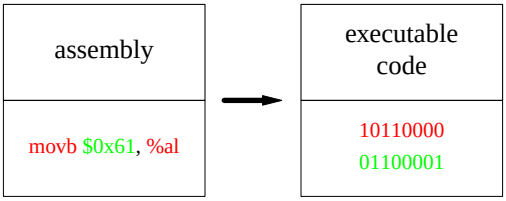
In a sense, assembly code is simple. Each line of code is an instruction, each instruction is a single operation performed by the CPU. There is no high-level concept to grasp, only very basic CPU instructions. This includes loading a value from the memory, performing a simple arithmetic operation, storing a temporary value in a register, etc.

```
; Put the value '0x61' (97 in decimal) in the AL register.
movb $0x61,%al
```

Keep in mind that assembly is the "human readable" version of the actual code executed by the CPU. This code is converted into series of 1s and 0s.

2.1 From assembly to executable code

Assembly is very simple: what you write is what will be executed. The application transforming your code into the final executable binary (named "the assembler") only "translates" your instructions into binary[2]. Instructions such as "mov" will be translated into their actual instruction number, register names will be translated into their actual address (a simple number), and that's about it.



In this example, the assembly code means "write value '97' (in hexadecimal) in the AL register". The code is then translated into the actual CPU instruction. This shows that the assembly code and the executable binary are two representations of the same thing.

2.2 Assembly is inconvenient

Assembly is a simple language but it has many flaws.

Non portable. Different CPU architectures require different assembly code. For example, amd64 and arm64 CPUs have different instructions, an application written in assembly for a CPU won't work for the other.

Difficulty. Assembly requires deep understanding of computer architectures. Writing a simple *hello world* already requires some knowledge about memory sections of executable binaries, a few CPU registers and what a kernel interruption is. And this basic application already requires more than a dozen of lines of code. In any serious codebase, this kind of verbosity can easily introduce bug-ridden code.

Verbosity. There is not much abstraction from the actual inner workings of the computer, which makes any operation quite verbose. Very basic and well-known programming concepts are missing, including conditionnals and loops.

Debugging is a nightmare. Memory-related bugs are both frequent in computing and mostly invisible, specially in assembly code[3].

2.3 Assembly in modern age

Assembly is still used nowadays for compiler development and in a few specific areas. Learning an assembly language today may be not worth the effort for most people. Usefulness of writing assembly by hand already was questioned in the 1950s!

My take. Do not try too hard to learn it. Understanding how everything works under-the-hood is beneficial mostly for developers writing operating systems (and related applications) or compilers. Writing efficient applications often requires to understand some details of the programming language used, not to know how to write an entire application in assembly. Though, understanding some of it already is valuable, even just for your own general culture.

However, understanding how the computer works is beneficial for any serious developer. There is a lot to learn: how your OS handles memory, processes and threads, the basics of computer networking and file systems, basic data structures, basic system administration (which is sadly way overlooked), etc. But it doesn't have to go as far as learning assembly.

2.4 Some noob questions

Is assembly the ultimate language? Any language beside assembly is an abstraction that ultimately will be compiled into an executable code. Since assembly is a human representation of what the CPU will execute, one could see any other language as assembly with an extra step. However, while being great to understand exactly what the CPU does, this representation actually is the worst way to understand the business logic of a program. Verbosity hides logic, bugs and errors. Code should represent the intent, not focus on details that can be automated.

Is assembly the fastest language? Assembly still can be more optimized than the best optimizers in C, even if C

2. Assembly code is almost exactly what is then executed by the CPU. The assembler does almost nothing and thus it really is fast.

3. Since this is a well-known problem, developers now have some leads to avoid most memory problems. This requires a strong discipline, and a lot of patience.

compilers are efficient[4]. However, the gain is so small that it isn't useful anymore by any stretch of imagination. For instance, one can remove a few instructions from the generated assembly from C code, which **may** make your code run 1/100 000 time quicker: **who cares?** This is a complete waste of developers time.

Should I learn assembly? Some compiler developers must understand it to verify their compiler is working correctly and to implement optimizations. Kernel developers may use it for drivers[5]. Security experts should understand at least some of it to understand security breaches. Beside some very specific cases like these, assembly just isn't needed anymore.

3. From assembly to C

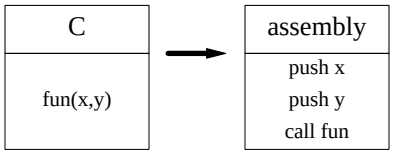
Compared to assembly, C represents a very big improvement on almost every level.

C code runs on several architectures. C is a step toward architecture abstraction[6], this language is "high level" in that regard. Though, it is still very much *bare-metal*, the final binary almost has no overhead compared to assembly. Most "low-level operations" are still provided as-is, memory is still explicitely managed, syscalls aren't sugar-coated, etc.

Readability improved a lot. Memory management is largely improved thanks to structures, memory offsets are now automatically handled in most cases[7]. Code representation (functions, operators) are now more intuitive and way less verbose (no more manual stack operations)[8]. Furthermore, control flow (conditions and loops) is now easier to read. Thus, business logic is more explicit.

Type checking is now a thing. Types allow the developer to reason about the code. Various verifications can be performed once the types have been correctly managed. C types are very limited, see the section on Haskell, but they still represent a big improvement from assembly.

Compiler handles various tasks automatically. For instance, function calls imply some memory stack management. For the developer, a function call is a single line of code, or even embed into a bigger expression, no need to think about stack pointers.



In this example, a function "fun" is being called by both C and ASM code. In ASM, this takes a few lines, and the code actually is simplified! In real code, this may take a dozen of lines, just for this. In ASM, business code is always hidden by details, code that could be abstracted away (it is repetitive, always the same, there is no reason to write it by hand).

Compile-time verifications. The compiler has to understand (to some extent) the code in order to produce the binary. Thus, some verifications are performed before running the application, which makes debugging way easier.

3.1 Some noob questions on C

Is C the ultimate language? C allows to write very efficient applications in a more friendly way than assembly. However, once again, as assembly, the language isn't a silver bullet: writing business logic in C may not always be wise. The rest of the document presents quite a few examples of such cases.

4. Zig and other "modern C" languages

Some design choices are crippling the C language. Thus, while having a C-like syntax, some modern languages provide a few notable improvements. Zig is an example of such language.

Readability. Despite being a massive improvement coming from assembly, readability of C code is still very much perfectible.

Simplistic type system. Generic structures and functions are hard to achieve. One must either use the macro system[9], inlined functions or void pointers. None of these solutions is great: they all require some code verbosity, readability issues or type verification issues at some point.

A simplistic type system leads to simplistic type verifications.

Standard library. C has a very simple library, with few functions. Each operating system has its own (or several) C library, and they **mostly** are the same but may differ slightly.

Portability. C allows to have the same code working on different architectures, but still has a few interoperability issues between operating systems. The standard library is (almost) never enough for an application, and external libraries often are specific to an OS (specially with user interfaces).

4. C compilers produce almost no superflous instructions, they may reorder some operations to optimize the code in known cases, etc.
5. Processors nowadays are so fast that kernel developers may use LUA for writing drivers instead of assembly. See NetBSD.
6. C and any other language beside assembly, really.
7. Taking a value from a structure "character.name" is easier to understand than an arbitrary number and an offset, such as "1000 + 30".
8. In assembly, a simple function call may take a dozen of lines, sometimes more. A function call involves several operations, such as changing some registers and putting parameters on the stack for example. In C, as in any other language beside assembly, these operations are automatically generated during compilation.
9. C macros are very limited, but that's a topic for another day.

5. Language paradigms and classifications

Before continuing and reviewing a few other languages, let's talk about language classifications.

Language classifications are flawed and may even seem arbitrary. Boundaries between two language families are often blur. Thus, this document presents an overview of main categories of languages, without too much details.

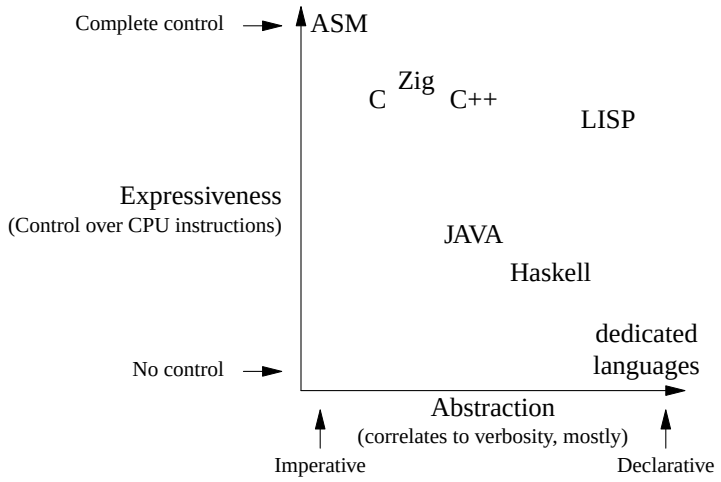
5.1 Imperative vs declarative

A first approach to classify the languages is to start with these two categories: imperative and declarative.

In an imperative language the developer instructs the machine how to change its state.

In a declarative language the developer merely declares properties of the desired result, but not how to compute it.

No boundaries truely separate imperative and declarative languages, a spectrum is a more appropriate. The following scheme places the languages mentionned in this document on a spectrum.



Dedicated languages include awk, sed, etc.

This scheme represents languages based on their expressiveness and abstraction. Expressiveness is, in this context, the ability for a language to have control over the instructions executed by the CPU. Abstraction is, in this context, the ability for a language to let the developer focus on the problem at hand instead of trivialities, programming details that can be handled automatically.

ASM is both very expressive and has no abstraction. The language allows an extreme control over instructions, the developer actually has to write them all down. And that's the main problem: verbosity. An enormous amount of trivial lines of code is required.

Dedicated languages (such as **awk**, **sed**, etc.) are the exact opposite of ASM. They focus on a specific set of frequently encountered problems, such as editing a stream of text. Thus, no control over CPU instructions is provided, only a very limited set of functionalities. However, the code is concise since the desired operations are written in a language tailored for the problem at hand. Because of the dedication of a language to a problem, one can write "sed 's/foo/bar/'" and not several hundred lines of ASM to do the same thing.

Any other language can be found between these two extremes. Most languages try to be concise, generic (to be able to solve any problem, not to be dedicated to a task) and to be efficient by allowing a *good enough* control over CPU instructions.

C and Zig are just an abstraction over ASM, allowing code to be compiled for different architectures and operating systems. Zig is slightly more abstract than C, it is less verbose and more generic. Zig includes a bit of meta-programming, which C also does (but not as good[10]) with its macro system, and a few constructs allowing to lower verbosity (see the "defer" keyword). Zig basically is C without most of its historical debt: no more shitty macros, better cross-compilation (by including different libcs), more regular syntax, more efficient compiler, code hot-swapping, standard library with many well-known and widespread structures, etc.
Beside these differences, C and Zig are in the same category.

C++ is a more abstract language, with abstract concepts, some of them are inherited from Object Oriented Programming (OOP) such as objects, classes, inheritance, etc. These concepts allow to reason about the code in a new way, away from the final ASM code that will be produced. The trend is toward some code that's easier to conceptualize for the developer, even if it means a slightly less efficient code, a longer compilation time[11] or even (in some cases) a more verbose code. Though, the language still is as expressive as C.

Java is similar to C++ in many cases (and it is OOP), but doesn't compile to a final executable. The code is transformed into "bytecode", a code representation that's neither code nor executable, but allows an interpreter to efficiently execute the code. Thus, Java isn't as expressive as C++, but verbosity is pretty much similar.

Haskell is the first functional programming language of this list. The language enforces the functional programming style, providing a language that's both strict **To be defined or to**

10. The C macro-system is flawed, it requires another language (the pre-processor) making any syntax and type verifications difficult. Also, the pre-processor and C have different syntaxes and rules, all quite arbitrary, rendering the learning of C more difficult than it should.

11. And it shows: C++ compilation time is a fucking joke.

finish.

5.2 Procedural vs Object-oriented Programming vs Functional Programming

5.3 Other categories of languages

6. Lisp (and scheme, etc.)

7. Haskell (and idriss, etc.)

8. Dedicated languages

Most languages are general purpose languages: they let you solve most problems that can be solved using computers. Such problems are for example doing equations, printing documents, displaying a graphical interface, etc. Having languages able to solve multiple problems is nice: a single language to learn and problems are solved. However, hard (or repetitive) problems sometimes lead to create a jargon. In natural languages, this happens in about every profession, allowing experts to talk with a precise language while keeping conversations concise. Thus, general purpose languages can be cumbersome to use for some problems, where a dedicated, specialized language may be preferable.

This section provides some insight on a few languages solving specific problems.

8.1 Algebraic computations: dc and bc

Both dc (desk calculator) and bc (arithmetic language and calculator) provide a way to perform calculus.

8.1.1 dc

The first one is the most basic. This language is a stacking (reverse Polish) calculator: numbers are stored on a stack[12].

```
2 2 + p    # computes then prints 4
c          # flushes all values on the stack, which
           # corresponds to the 4 previously computed
1 2 f      # prints values on the stack: 2 then 1
```

8.1.2 bc

The bc language is a preprocessor for dc It provides a more classical language paradigm since its syntax is C-like. Also, bc is an interactive tool, easy to pick-up when we need a calculator.

12. The choice of stacking values and the reverse Polish notation can be surprising as of today, but is memory efficient. I guess this language had its place on computers in a time where memory was very scarce.

```
1 + 2      # prints 3

scale=2    # precision (2 after decimal)
5/4        # prints 1.25

scale=3    # precision (3 after decimal)
5/4        # prints 1.250

for (x=0; x < 10; x++) {
    x      # prints consecutively values from 0 to 9
}

quit       # stops the program (bc is interactive)
```

The language allows to create functions, and with the [-l] parameter provides a few mathematical functions, such as log, sine and cosine, exponential, etc.

bc is a good calculator, simple to use and with arbitrary precision.

```
echo "scale=3; 5/3" | bc
# 1.666
```

8.2 String manipulations: sed and awk

Strings are one of the simplest types. Available in any general purpose language (one way or another), printable, easy to understand; strings are a good candidate to be the common ground data serialization between applications. That was the Unix way, and still is massively used nowadays.

To manipulate strings, several approaches.

8.2.1 sed

sed is a language to edit and filter *stream* of data.

8.2.2 awk

awk is a language based on patterns, and which can easily work on columns instead of whole lines. **T o be defined or to finish.**

Awk can easily work on columns.

```
# Print file sizes (5th column of the 'ls' output).
ls -lhA | awk '{print $5}'
```

Awk allows to work on patterns.

```
awk '/hello/ {print}'
```

Print a whole line when the pattern "hello" is encountered.

As **sed**, **awk** is a language and can be used in a script. The following

is a simple awk script example.

```
# Print each line containing 'hello'.
/hello/ {
    print # no parameters = prints the whole line
}

# Print second column when the first one is 'blah'.
$1 ~ /blah/ {
    print $2
}
```

Awk allows to easily work on specific portion of text, between patterns. For instance, let's consider the following text:

```
<container A>
Blah
<endcontainer>
<container B>
Blah
<endcontainer>
<container C>
Blah
<endcontainer>
```

With this text, rules can be written to apply only for a specific 'container'.

For more complex operations, people are encouraged to use more general purpose languages.

8.3 makefile

8.4 Macros Macros Macros: m4

The `m4` language helps creating macros, meaning it changes text within text.

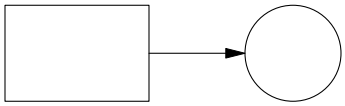
In The Art of Unix Programming (Raymond, 2003) we can find the following citation about m4.

> *[...] actually trying to use m4 as a general-purpose language would be deeply perverse.*
>
> — Eric S. Raymond

8.5 shell

8.6 pic

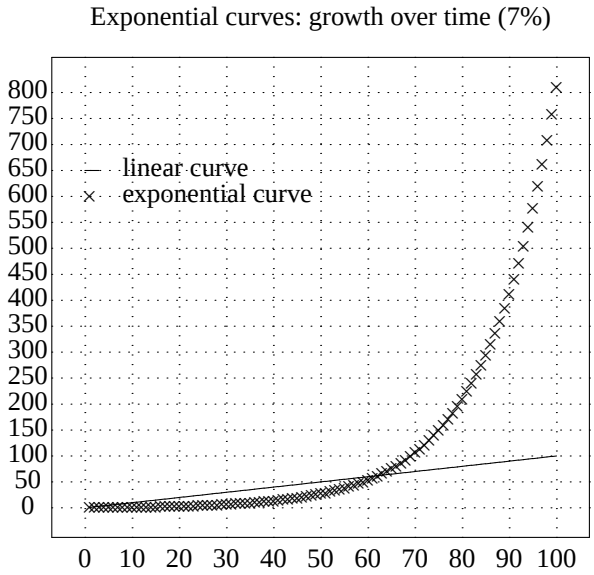
pic is a language to create schemes. Here is an example of such scheme and the code producing it.



Code: `box; arrow; circle`

8.7 grap

grap is a language to create graphs.




```
Produced by
 GROWTHFACTOR=0.07
 grid bot  dotted from 0 to 100 by 10
 grid left dotted from 0 to 800 by 50
 frame ht 2.5 wid 2.8
 define expo { $1+$1*GROWTHFACTOR }
 value = 1
 draw LINEAR solid
 for i from 1 to 100 by 1 do {
   next LINEAR at i, i
   times at i, value
   value = expo(value)
 }
 l1legend=650
 l2legend=600
 line from 0,l1legend to 3,l1legend
 times at 1,l2legend
 "linear curve" ljust at 8,l1legend
 "exponential curve" ljust at 8,l2legend
 label top "Exponential curves: growth over time
(7%)" up -.2
```

References

Eric Steven Raymond, *The Art of Unix Programming*, Pearson Education, Inc (2003).