Introduction

All About Metaprogramming

In this book, we will explore using Smojo for **meta programming** - programs which write other programs.

When and why is this useful?

To understand, we need to step back a little and see how modern programming languages get their advantages.

In the 1970's, when programming was at its infancy, you could say that programming took a fork in the road. There were no "personal" computers and computers were usually huge, expensive machines sitting in the offices of large corporations or government.

Programmers coded their programs either directly in machine code or in assembly. This was tedious and error-prone process.

The 1950's - 1970's were the golden age for programming languages. FORTRAN (1950's), ALGOL and COBOL (late 1950's), Lisp (1960's), C (1970's) and Forth (1970's) , were among the pioneer languages that shaped the style and form of modern languages we are now familiar with.

During the late 1950's to the early 1970's, several **higher level programming languages** were developed, with the purpose of making it easier for programmers to write programs.

Most of the languages we use now - Smojo, Java, Python, Ruby, etc. can trace their ideas from these foundational languages.

Higher Level Languages

The purpose of these higher level languages was to help programmers tame the complexity when constructing programs. They did this in a few ways:

First, they offered a standard set of **keywords** corresponding to frequently required operations: looping, conditional branching, arithmetic and the manipulation of computer memory.

These keywords were either built-in operations (eg, `+` for addition in C) or programming constructs (eg, the **`for`** loop in C). Keywords almost without exception could not be changed by programmers using the language, because this immutability was a stable base on which programmers could agree on program semantics. In fact, in later years, standardisation of these languages became important, using the ANSI standardisation process.

With the exception of Lisp and Forth, programmers using these languages could not add new programming constructs. You were left with whatever the designers had already built into the language.

Second, in order to extend the functionality beyond the small core set of keywords, language designers offered a set of **libraries** and a method to link these to your own code. Libraries offer a vast array of packaged functionality easily accessible to the programmer. All he needed to know was the functions exposed by the library and how to call them from his program.

Programmers could create their own libraries and either distribute the source-code for others to use or distribute the compiled binaries.

The C programming language

The C programming language is unique because it alone was used to build an operating system called UNIX in the 1970's. Linux, MacOSX, the popular Android operating systems and the various flavours of BSD are all descended from UNIX and written in C for the most part.

NOTE: In this book, I will assume you are familiar with the C programming language.

If you are not, I would encourage you to peruse the excellent tutorial from www.tutorialspoint.com

C remains an important and widely useful language to this day. It is used to create the fast math and computation libraries that make Python a popular language for data science. Pure Python code is very slow by comparison, and although there are niche methods to compile python, to my knowledge, none is widely used in data science or AI. What makes Python usable is its ability to use existing libraries written in C.

C retains its appeal because it has:

- **Standardised semantics**, so programmers can reliably depend on its operation.
- **Small set of keywords**, which make learning C relatively easy. Its offshoot, C++ while also popular, is much harder to learn and use.
- **Very large set of libraries**, for almost anything you need to do. Many of these libraries are open-sourced, which means you had the option of customising or forking your own version.
- **Low-level access to computer memory**, using pointers. Which allows for faster operations or operations not possible any other way (eg, access into i/o devices). Other languages disallow this (eg Java) to increase "safety" (ie, your program is less likely to crash) but at the expense of speed.
- **A compiler for every hardware platform**. Code written in C can be compiled and run for almost any hardware platform (x86, ARM, RISC V, MIPS, microcontroller) or operating system. It can even create code that runs "standalone" without an OS. Many of these compilers are mature for the major platforms and produce optimised code. GCC and Clang are two popular and free compilers for C.

Drawbacks of Libraries

While they are very useful, there are systemic problems with using libraries, because of:

- **Unanticipated Situations:** Libraries by definition cannot anticipate every situation. Prudent designers often optimise for the "80%" use case. This leaves the remaining 20% unoptimised or performing poorly.
- **Layered Libraries:** Many modern frameworks are often layered in a hierarchy. Each layer is built on the other. This promotes better testing and simplifies development. However, each layer also contributes to a

decrease in speed and resources to be consumed needlessly. I have seen this happen with complex frameworks (eg Tensorflow).

- **Unnecessary Code:** Not all functionality of the library might be needed, however (depending on the OS) **all** of the library is loaded on an invocation. This can be a problem in constrained environments.
- **Breaking Changes in Libraries**: Most libraries are out of the programmer's direct control, being developed by others. Unfortunately, as libraries change (which is a good thing - it brings bug fixes, improvements etc.) library developers have a habit of needlessly introducing breaking changes to **interfaces** (ie, APIs) previously established. This is mostly a problem for open-sourced projects (You can see such breaking changes in many popular frameworks from everything from deep learning to web servers) but less so for commercially-backed products like the JDK.
- **Libraries can be hard to use**: A design solution for unanticipated situations is to create either (i) very simple generic APIs that meet the 80% need (the **`ioctl`** function in Linux is a good example) or (ii) create a set of APIs that must be invoked in a particular way to anticipate every situation (eg, Linux's **`v4l2`** video library). Simple generic interfaces are problematic because they attempt to hide complexity or abstract away hardware changes, possibly at the expense of being optimal in terms of speed or resource usage. Complex interfaces are hard to use correctly, or to understand. And they are also more likely to suffer from Breaking Changes (since the steps are more in number).

In most situations — I would say 80% of the time — these concerns are minor. But in some specific instances, they are serious - especially in the areas of:

- **High-speed processing:** training and inferencing of neural networks, high-volume data processing and realtime applications.

- **Resource-constrained environments:** microcontrollers, training of NNs.

In these cases the first three drawbacks of libraries are become especially apparent.

Our Approach

Our approach to solving these issues is twofold.

To specify a program for a particular area we propose constructing **Domain Specific Languages (DSLs**) that can be used seamlessly with the rest of the language. DSLs go a long way in addressing the problem of complexity in library use and partly the problem of premature abstractions.

Secondly, we will also rely heavily on **automatic program generation** to solve optimisation issues caused by premature abstractions and layered libraries. In this case, our programs will create as their output optimised code (I have selected C and Assembly as our target languages).

Taken together, I will call these two techniques "meta-programming" since it is about programs creating programs.

Of course, we can and still will use libraries! But these tools are useful because they help us build optimised programs.

Why Smojo?

NOTE: In this book, I will assume you are familiar with the Smojo programming language.

If you are not, I would encourage you to complete Smojo's "Basic" tutorial online.

Every programming language shines in a particular area. Smojo's strength is in meta programming, specifically in the areas I've just outlined. In this book, we will address two questions:

Q1: How can we use Smojo to develop DSLs that enable beginners to create complex programs for high-speed data processing or on constrained hardware like microcontrollers?

Q2: Can we develop a framework for Smojo to C generators that output clean, readable and optimised code from such DSLs?

Exercises

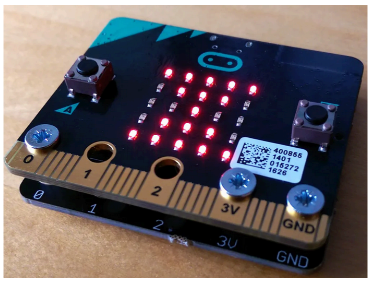
Figure 1: BBC's micro:bit is a microcontroller designed for children.

Figure 2: Arduino is a popular microcontroller platform (hardware and semi-standardised software) for hobbyists and industrial use.

1. Research the `ioctl` function in Linux. What might be some advantages and problems using it?
2. Research the **Video4Linux** library (`v412`) mentioned in the text. Is it easy to use? Why? (or Why not?)
3. Research the video libraries that you might use to connect a camera for video capture in the Raspberry Pi. (a) How many libraries are there? (b) Which are still in use? (c) Which have been abandoned? What language are these drivers or libraries written in?
4. Research the driver suite used for BBC's Micro:bit microcontroller. What language is it written in? What "official" languages can you use to program the Micro:bit?
5. How much RAM and storage does the Micro:bot microcontroller have? Compare these specs against that of the the official Arduino Uno.