From Basics to Bytecode: A Guide to Computers and Programming

Set Lonnert
& ChatGPT 4.o from OpenAI[a]

[a] https://openai.com/

Contents

Introduction

This book is an experiment. It is an exploratory book, designed to guide readers from already acquired fundamentals of programming to complex real-world applications, with an emphasis on learning by doing. It builds upward from basic concepts, encouraging readers to engage deeply with both small-scale and large-scale programming environments. And it assumes using Large Language Models (LLMs), which assist you with feedback, code suggestions, and deeper understanding.

1: Foundations of Programming. We begin with an introduction to fundamental programming concepts, including variables, control structures, and functions. Along the way, we'll explore these topics in greater depth and cover related foundational concepts. To enhance your understanding, you'll be encouraged to use LLMs alongside your learning process. These tools will offer immediate feedback, code suggestions, and explanations, helping you grasp the material more effectively.

2: Understanding Virtual Machines. Transition from basic programming into the concept of virtual machines to understand computers. We will use simple examples of virtual machines, and gradually introduce more complex virtual machines, linking each one to programming concepts previously covered.

3: Debugging, Optimisation, and Tests. In this part, you'll learn systematic debugging techniques to identify and resolve issues effectively. We will explore optimization strategies to enhance code performance. Lastly, we'll cover the fundamentals of testing, including unit testing, integration testing, and using automated testing tools to ensure code reliability and robustness.

4: Building and Experimenting. Here we encourage you to modify and experiment with the provided examples on real hardware, the Raspberry Pi Pico. This part includes project ideas for readers to explore. Use exercises at the end of each chapter to extend the examples or apply them to solve specific problems.

5: Compilers and Advanced Concepts. We introduce compilers and

parsing with practical examples. We show how the virtual machines introduced earlier can be used to run compiled code. Also there are references to hands-on projects where you build simple compilers or interpreters for your own small languages.

6: Case Studies. Examine real-world virtual machines and discuss their importance in software development. Explore how the concepts you've learned apply to modern programming environments.

7: Methods and Tools. There are many ways to approach programming, through style and philosophy of the programmer. Here we make a choice of the 'craftsperson'.

8: Advanced Topics. Delve into more advanced topics such as abstract data structures, garbage collection, memory management, and graphics. ...

Introduction

In the 1980s, I was optimistic about AI, especially logic programming, but became disillusioned during the AI winter. The internet's rise in the 1990s revolutionized access to information, and I believed AI would eventually transform human-computer interaction. With the advent of Large Language Models (LLMs) in the 2020s, this vision is finally being realized, reshaping how we interact with technology.

Some parts of this book date back 20 years, originating from a manuscript for a Java programming textbook that was never published (intended it to be an update of an earlier textbook). Much of the content on virtual machines was written long before the advent of Large Language Models (LLMs) and was subsequently posted on GitHub.

However, with the announcement of ChatGPT 4.o, I reconsidered the prospect of writing a new textbook—one that I would study myself. This time, the book will shift its focus from learning a single programming language, to understanding how computers work starting with virtual machines. With the help of LLMs, this approach is now feasible.

In the early 1970s, there was significant debate about the use of *calculators* in schools, with concerns that students might lose the ability to perform basic arithmetic. Over time, this perspective evolved, and calculators became widely accepted in classrooms. While mental arithmetic is undoubtedly a valuable skill, one might question whether the time and effort required to master it are justified, given the efficiency and availability of calculators.

Similarly, when learning programming, the use of an *interactive language* or one with a fast compiler provides immediate feedback, allowing for quick iteration and testing. This approach is an incredibly powerful tool for accelerating the learning process. My own introduction to programming in the very early 1980s was through the interactive language BASIC. While BASIC may not have

been the most sophisticated language, it served as a practical tool for gaining familiarity with and understanding the coding process.

Reflecting on these experiences highlights *the value of embracing new tools in education*. Learning what a tool replaces, how and when to use it, and understanding its underlying mechanisms becomes an integral part of what coding and programming will evolve into in the future.

Purpose

The book is designed for *beginners in the programming field*, utilizing *Large Language Models* (LLMs) to facilitate interactive and personalized learning. At the time of writing, notable examples of such AI models include Google's Gemini, Meta's LLaMA, and OpenAI's ChatGPT. These models can enhance the learning experience by providing real-time support and guidance, making the educational process more dynamic and responsive.

The proposed structure of the book follows a project-based learning approach. Each chapter will focus on building small, practical projects that reflect real-world applications. LLMs will be integrated as virtual assistants throughout these projects, offering real-time support to answer questions, clarify concepts, and provide guidance.

- *AI as a learning partner.* AI can play an active role in education by serving as a mentor, providing instant feedback, and generating code snippets.
- *Interactive learning with LLMs.* A programming book for beginners can use LLMs to offer personalized and interactive education.
- *Project-based approach.* The book will focus on hands-on projects, with LLMs assisting throughout the learning process.
- *Interactive languages.* The use of interactive languages, such as Python, highlights the benefits of immediate feedback in learning programming.
- *Value of new tools.* Understanding and embracing new tools in education is crucial for advancing teaching methods and learning experiences.

Prerequisites

You are expected to have some prior experience with programming. In this book, we will focus on three widely recognized programming languages: Python, C, and JavaScript. They are used as starting points here, but naturally not exclusive to learn programming. Each represents a certain approach to programming. JavaScript is particularly accessible, as it can be run directly in web browsers. There's no need to install a dedicated development environment—simply write your code in a text editor, save it, and open it in a browser. Python is widely used, user-friendly, and supported by a vast amount of readily available help. C,

though considered an older language, remains highly valuable for its efficiency and simplicity, making it an excellent language for understanding fundamental programming concepts.

Installing Python and C can vary depending on the computer and operating system you are using. Fortunately, there are plenty of resources available online to guide you through the process. You can search for installation guides specific to your operating system, or even use an LLM to assist you with step-by-step instructions and troubleshooting tips.

You are also expected to engage in some hands-on work with hardware, particularly in connection with the Raspberry Pi Pico. While the choice of using the Pico over other platforms, such as Arduino, STM32, or similar microcontrollers, is entirely arbitrary, it is important to note that the Pico offers considerable processing power at a relatively low cost. Additionally, it boasts high availability, making it an excellent option for both beginners and experienced users alike. This versatility allows for a wide range of projects, enhancing your understanding of programming in conjunction with hardware.

The book presents a very steep learning curve, requiring readers to actively engage with LLMs to acquire new and complementary knowledge in all relevant fields.

A friendly warning on LLMs

At the time of writing, LLMs (Large Language Models) have already had a significant influence on society, where they are *expected to disrupt conventional learning approaches and redefine the skills we prioritize*. This emerging technology is anticipated to change not only how we acquire knowledge but also how we apply it in practice.

While LLMs can rapidly generate useful code snippets or explanations, they often lack full context or deeper understanding of specific problems. This is where human intuition and expertise become essential. Programmers must critically evaluate the suggestions, spot potential logical inconsistencies, and decide when to trust or override the LLM's output. Experienced developers are more likely to recognize subtle issues that might not trigger immediate errors but could lead to inefficient or incorrect behavior down the line. Therefore, while LLMs can be powerful tools for learning and problem-solving, they should complement, not replace, traditional problem-solving skills and thoughtful analysis.

Here are some problems identified by an LLM (ChatGPT 4.o) itself:

- LLMs can provide incorrect or outdated information as they rely on patterns from training data without fact-checking or cross-referencing.
- They may struggle with context, leading to irrelevant responses or misinterpreting student questions.

- LLMs can reflect biases from their training data, raising concerns about fairness and equity in education.
- Over-reliance on LLMs might hinder students' critical thinking and problem-solving skills.
- Privacy and data security concerns arise when using LLMs, as they may require sharing personal information.
- LLMs offer generalized responses, which may not suit individual learning styles or adapt to unique student needs.
- Ethical concerns exist around academic honesty, with LLMs potentially enabling cheating.
- Not all institutions have the resources to implement LLM-based tools, limiting access in under-resourced settings.

While LLMs have the potential to greatly enhance education by providing instant feedback, explanations, and resources, it is crucial to address these challenges to ensure they are used effectively and ethically.

Balancing the use of LLMs with traditional educational methods and human oversight can help mitigate these issues.

Note on exercises

This material doesn't include traditional exercises. Usually, exercises help guide learning and let readers or teachers check understanding and progress. But with large language models (LLMs) now available, the focus may shift towards project-based learning. LLMs can assist with projects and support learning, but more importantly, they highlight a new way of learning—one that values active participation, creativity, and deeper understanding instead of repetitive drills. You can find relevant code, a few exercises, starting points, project ideas, and other related info on GitHub.

That said, while LLMs are powerful tools, relying on them too much can hinder the development of critical thinking and problem-solving skills. It's important to use them in balance with independent learning and reasoning.

From Basics to Bytecode and **Workbook**
https://github.com/Feyerabend/bb/tree/main
Scan the QR code to access more information, errata, exercises, code, resources and updates.

Chapter 1

Foundations

We begin with the fundamental concepts that form the foundation of elementary programming: *variables*, *control structures*, and *functions*. A variable is a named storage location in memory that holds data. The value stored in a variable can change during program execution, a "variable." Control structures are constructs that determine the flow of execution in a program. They allow a program to make decisions, repeat actions, and branch into different paths based on conditions. Functions are sequences of instructions stored in memory that the CPU can execute. Typically, when a function is called, the computer temporarily pauses the current sequence of instructions, switches to the function's instructions, and then returns to the original sequence once the function has completed. Functions in computers can perform tasks ranging from basic arithmetic to complex algorithms, and they can invoke other functions, enabling intricate chains of computation.

1.1 Simple data types

You've probably heard that digital computers use binary numbers, which are composed of ones and zeros, to represent all types of data and instructions. This binary system, based on just two states (on/off, true/false, or 1/0), is fundamental to how computers operate.

Let's explore how different types of data, like integers, floating-point numbers, characters, and strings, are represented in this binary system, particularly within the context of an 8-bit word length, which is a common unit of data in computing.

When we talk about an 8-bit word length in computing, we're referring to the amount of data that a computer's processor can handle or process in a single operation. In this context, a "word" is a fixed-sized group of bits that are processed together as a unit by the CPU. The length of the word determines how much data the CPU can process at one time. The 8-bit word length wasn't initially established as a standard but rather evolved over time due to practical

considerations and technological advancements. Data with the 8-bit length is called a "byte." Today an ordinary computer you would use such as a laptop, probably have a 64-bit processor, thus operating with 64 bit words.

1.1.1 Integers in binary

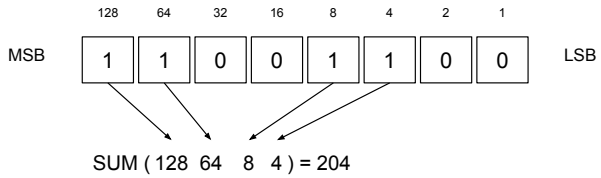
In an 8-bit system, each integer is represented by a sequence of eight binary digits (bits). Each bit can be either 0 or 1, and the combination of these bits determines the integer value. The leftmost bit in the sequence is the most significant bit, and the rightmost bit is the least significant bit. Here's how it works:

0 in binary is 00000000.

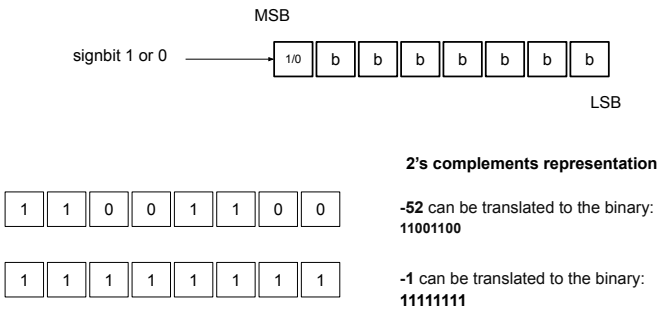
1 in binary is 00000001.

2 in binary is 00000010.

45 in binary is 00101101.



In this system, you can represent any integer from 0 to 255 (for unsigned integers) using 8 bits. If signed integers are used, one bit is reserved for indicating the sign (positive or negative), allowing representation of values from -128 to 127.



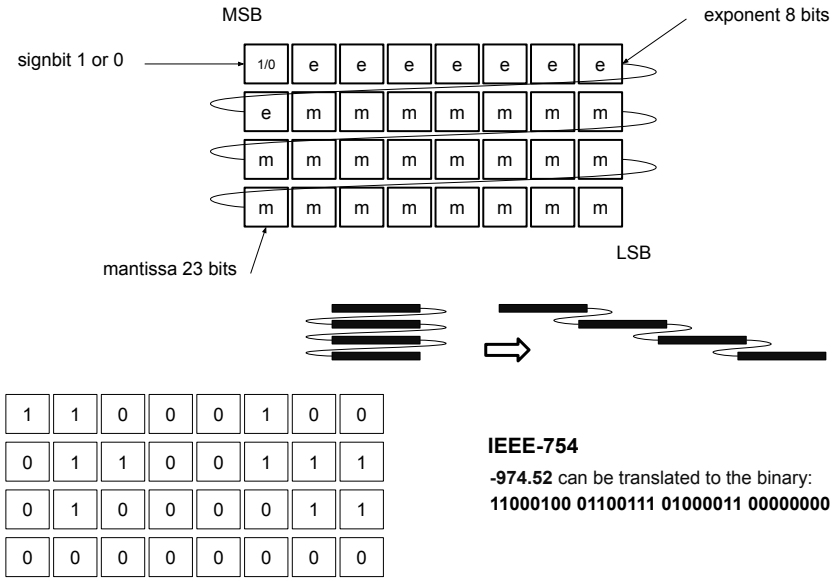
2's complement representation. When using positive numbers the highest bit, most significant bit (MSB), is 0. The remaining bits represent the

number in binary form, just like unsigned integers. When using negative numbers MSB is 1. The remaining bits are used to represent the magnitude of the negative number using a specific method.

1.1.2 Floating-point numbers

Floating-point numbers, which represent real numbers (like 3.14 or -2.7), are more complex. They are stored using a standard format known as IEEE 754 in most systems, which divides the number into three parts: the sign, the exponent, and the mantissa.

This structure allows computers to represent a wide range of real numbers, including very large or very small values, by encoding them in scientific notation -9.7452×10^2 into binary 32-bits approximation.



1.1.3 Characters and ASCII

Characters in computers, such as letters, numbers, and punctuation marks, have historically been represented using standardised encoding systems. One of the earliest and most influential of these systems is ASCII (American Standard Code for Information Interchange).

For example:

The letter `A` is represented as `01000001`.

The letter `a` is represented as `01100001`.

The character `0` (zero) is represented as `00110000`.

The space character is 00100000.

The English alphabet, with its Latin characters, has the advantage of being relatively limited in scope, which simplifies its representation in digital form. However, representing other languages has proven more challenging, leading to various modifications, replacements, and adaptations.

UTF-8 (8-bit Unicode Transformation Format) was introduced in the early 1990s as part of the Unicode standard, which aimed to solve the limitations of ASCII by providing a single, unified character encoding system that could represent virtually every character used in human languages.

1.1.4 Strings

A string is simply a sequence of characters, so it's stored as a sequence of 8-bit binary numbers, one for each character. For example, the string "Hello" would be stored in memory as:

`H`: 01001000

`e`: 01100101

`l`: 01101100

`l`: 01101100

`o`: 01101111

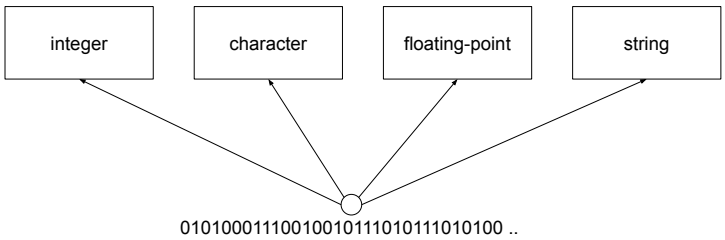
These binary sequences are stored consecutively in memory, and a special character, often called a null character (with a binary value of 00000000), is used to indicate the end of the string.[1]

1.1.5 Representations and types

In essence, everything in a computer—whether it's numbers, text, or instructions—is represented by sequences of binary digits. Understanding these basic concepts of binary representation helps in grasping how computers store and manipulate different types of data, whether it's an integer being processed in a calculation, a floating-point number being used in scientific computation, or a string of text being displayed on a screen.

To distinguish between different binary representations, we use types. Types not only define the kind of data (such as integers, floating-point numbers, or characters) but also how that data is interpreted and manipulated. In some cases, types can be converted from one to another, depending on their compatibility. Additionally, types can encompass more than just raw binary data; they can also include the operations that can be performed on them. For instance, in object-oriented programming, types can represent "objects" that combine both data and the methods (or tools) that operate on that data.

[1] There are actually other ways to terminate a string, such as having the length of the string occupy the first bytes of the representation.



1.1.6 Summary

Simple data types

Digital computers use binary (ones and zeros) to represent all types of data. An 8-bit word length (a group of 8 binary digits) is commonly used to represent a unit of data, called a byte. Different types of data, like integers, floating-point numbers, characters, and strings, are represented using binary in distinct ways.

Integers

Integers are represented by a sequence of binary digits, and in an 8-bit system, each integer value corresponds to a combination of 0s and 1s. You can represent values from 0 to 255 (unsigned) or -128 to 127 (signed). Two's complement is a common way to represent signed integers.

Characters

Characters, such as letters and symbols, are encoded in binary using systems like ASCII. Each character corresponds to a unique binary value. ASCII was extended by Unicode (UTF-8) to represent more characters from various languages.

Floating-point numbers

Floating-point numbers represent real numbers, stored using a format like IEEE 754, which divides the number into sign, exponent, and mantissa. This format allows for a wide range of values by storing numbers in a scientific notation-like structure.

Strings

Strings are sequences of characters, and each character in a string is represented as a binary value. A special character, typically a null character (00000000), is

used to indicate the end of the string.

Representation and types

Everything in a computer is represented by binary digits. Data types define how this binary data is interpreted and manipulated. In some cases, data types can be converted between formats if compatible. Types can also represent more complex structures, like objects in object-oriented programming.

1.2 Variables

In mathematics, a variable is a symbol that represents an unknown or changeable value. Variables are fundamental in expressing mathematical formulas, equations, and functions. For example, in the equation The formula $y = 2x + 3$, x and y are variables, where x can take on different values, and y is determined based on x. Variables allow mathematicians to generalise problems and work with abstract concepts rather than specific numbers.

In computer science though, a variable is a storage location in a computer's memory that is associated with a symbolic name (identifier) and can hold a value. This value can change during the execution of a program, making variables essential for dynamic behaviour in software. Variables can hold various types of data, such as numbers, strings, or more complex structures (like arrays and objects). For example, in a program, you might declare a variable age to store a user's age, and this value can be updated as needed.

As mentioned earlier, variables can store different kinds of values and are therefore classified into different types based on the nature of these values. For instance, a variable of type "8-bit positive integer" may hold a binary number, which could also correspond to an ASCII character. They hold the same space of binary digits. However, the type of the variable determines how we interpret its content—whether as an integer or as a character.

1.2.1 Assignment

Assignment in computer science refers to the operation of *assigning a value to a variable*. It's a fundamental concept in programming, as it allows developers to store data, manipulate it, and control the flow of a program.

Assignment is a basic yet essential operation in programming. It allows developers to store, update, and manipulate data within a program. Understanding assignment, especially in the context of different programming paradigms and data types, is crucial for writing effective and efficient code.

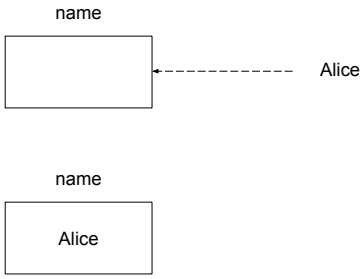
What is assignment? Assignment is the process of setting or updating the value of a variable. In most programming languages, this is done using the assignment operator, typically represented by an equals sign ('='). The general syntax for assignment is in Python:

```
variable_name = value
```

And an example:

```
x = 5
name = "Alice"
```

The variable to the left ("name") is assigned to the value on the right ("Alice").



Memory allocation. When you assign a value to a variable, the computer allocates memory to store that value. The variable name acts as a reference to the location in memory where the data is stored.

Overwriting. If a variable already holds a value and a new value is assigned to it, the old value is overwritten. The memory location is updated with the new value, and the previous data is typically discarded (unless the language has garbage collection or other memory management features).

Simple assignment. A variable assigned a direct value.

```
age = 30
```

Compound assignment. These are shorthand operations that combine an arithmetic operation with assignment. For example:

```
x += 5 # same as x = x + 5
y *= 2 # same as y = y * 2
```

Multiple assignment. Some languages, such as Python, allow assigning multiple variables in a single statement, assigned in sequence.

```
a, b, c = 1, 2, 3
```

Type consistency. In *statically-typed* languages, the type of the variable must be declared beforehand, and the value assigned must be of that type. In *dynamically-typed* languages, the type can be inferred at runtime based on the value assigned.

Static in C:

```
int number = 10; // 'number' must be an integer
```

Dynamic in Python:

```
var = 10 # 'var' can later hold a different type, like a string
var = "hello"
```

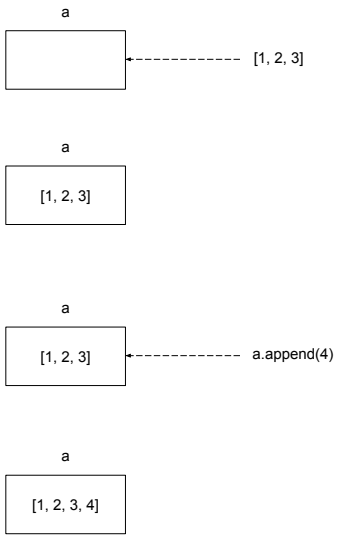
1.2.2 Mutable and immutable variables

Mutable and *immutable* variables are fundamental concepts in programming. Mutable variables can be more memory-efficient since they can be changed in place without allocating new memory. However, immutability can lead to safer, more predictable code, as it reduces the risk of accidental changes to data. Immutable variables are also inherently thread-safe, meaning they can be shared across multiple threads without the need for locks or other synchronization mechanisms, which makes them ideal for concurrent programming.

Mutable variables are those whose values can be changed *after* they have been initialized. When a mutable variable is modified, the program can update the data stored in the same memory location. This means you can alter, add, or remove data from the variable without creating a new instance of it.

In Python, lists and dictionaries are mutable. You can change elements, append new items, or remove items directly.

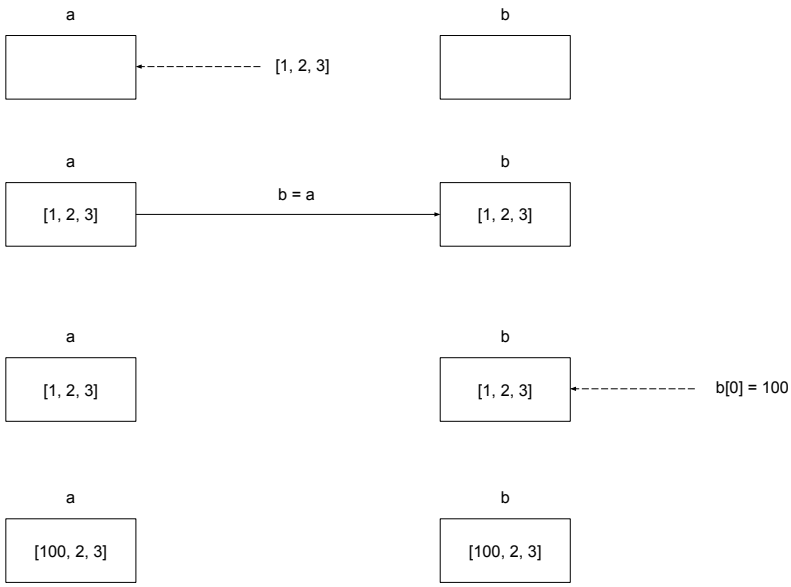
```
a = [1, 2, 3]
a.append(4) # a is [1, 2, 3, 4]
```



When *reference assigning* complex data types like objects or arrays, the variable often holds a reference to the original data. Modifying the new variable can affect the original data.

```
# reference assignment with lists (mutable)
a = [1, 2, 3] # assign a list to variable 'a'
b = a # 'b' points to the same list as 'a'
b[0] = 100 # changing 'b' affects 'a'

print(a) # output: [100, 2, 3] (changed)
print(b) # output: [100, 2, 3]
```



Immutable variables on the other hand, *cannot be changed* once they have been created. Any operation that seems to modify an immutable variable actually creates a new instance with the updated value. When you attempt to change an immutable variable, the program creates a new object with the new value, leaving the original object unchanged.

Examples in Python, strings and tuples are immutable. Any operation that modifies a string will result in a new string.

```
astring = "hello"
astring = astring + " world" # astring is a new string "hello world"
```

When *value assigning* primitive data types, a copy of the value is made. Changing the new variable does not affect the original.

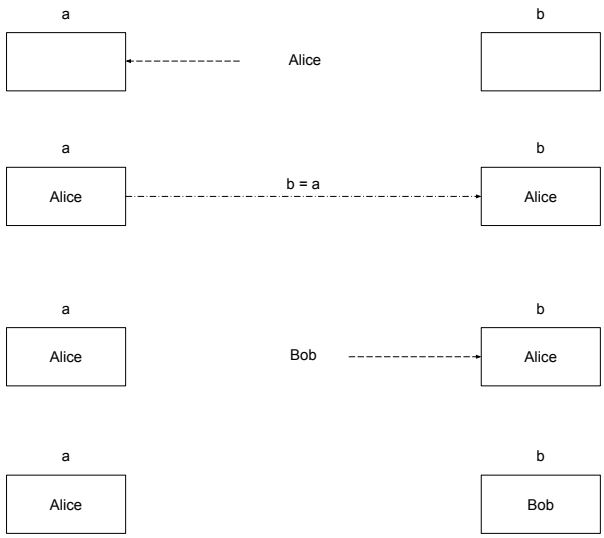
```
# value assignment with integers (immutable)
a = 10 # assign 10 to variable 'a'
b = a # 'b' gets a copy of the value stored in 'a'
b = 20 # changing 'b' from 10 to 20 does not affect 'a'

print(a) # output: 10 (unchanged)
print(b) # output: 20
```

Or another sample with strings.

```
# value assignment with strings (immutable)
a = "Alice" # assign 'Alice' to 'a'
b = a # 'b' gets a copy of the value stored in 'a'
b = "Bob" # changing 'b' from 'Alice' to 'Bob' does not affect 'a'

print(a) # output: Alice (unchanged)
print(b) # output: Bob
```



Assignment in functional programming. In functional programming paradigms, assignment is often discouraged or restricted. Instead of reassigning values, new variables or data structures are created. A variable can be reassigned new data, but the previous data remains unchanged—it is simply discarded. This known as the above *immutability*, ensures that data remains constant, helping to prevent unintended side effects, and is one of the more beneficial features of functional languages.

1.2.3 Summary

Assignments

Assignment refers to the process of assigning a value to a variable using an assignment operator (like '=' in many programming languages). It allows values to be stored, updated, and manipulated. Compound and multiple assignments are shorthand techniques used for efficiency. Assignment behavior varies between statically and dynamically typed languages.

Memory allocation

When a value is assigned to a variable, memory is allocated to store it. Overwriting an existing variable replaces its memory content. In statically typed languages, types must be declared, while dynamically typed languages infer types at runtime.

Mutable versus immutable variables

Mutable variables can be changed after being created, while immutable variables cannot. Changing a mutable variable modifies the original data, but changing an immutable variable creates a new object. Mutability affects memory efficiency and thread safety in concurrent programs.

Mutable variables

Mutable variables can be updated in place, which means changing their value doesn't create a new memory object. Examples include lists and dictionaries in Python. Changing a reference to a mutable object can alter the original data.

Immutable variables

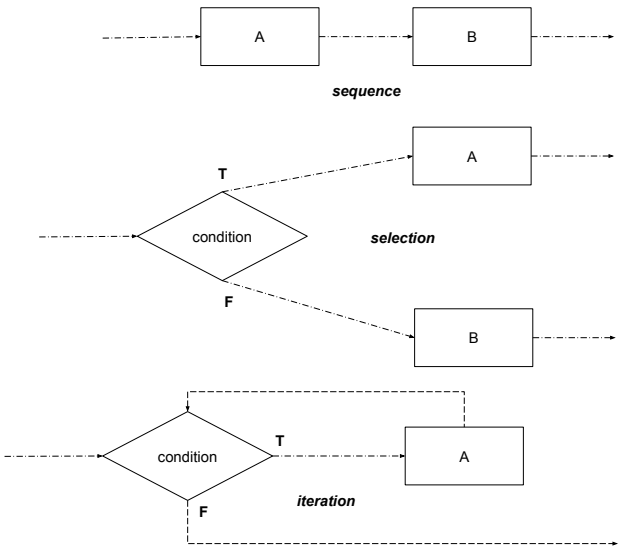
Immutable variables, such as strings and tuples, cannot be changed once created. Modifying an immutable variable creates a new instance rather than updating the existing one. This behavior ensures data safety and prevents unintended side effects.

Functional programming

In functional programming, assignment is often discouraged, and immutability is favored. Instead of reassigning values, new variables are created. This helps prevent side effects and ensures data consistency throughout the program's lifecycle.

1.3 Control structures

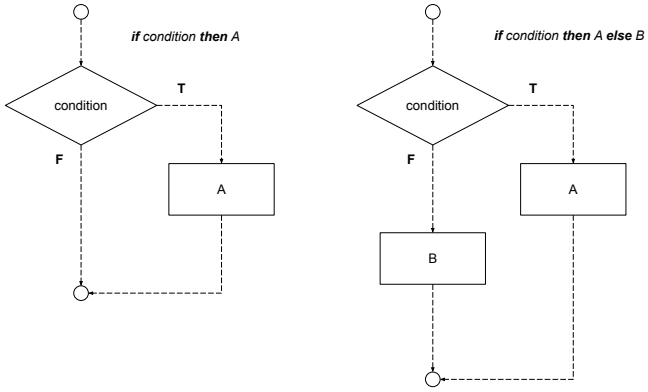
Traditional programming but also applicable concepts has often been based on imperative languages, where statements dictate what should happen in a kind of command-like manner. These instructions typically involve operations like taking two given numbers, multiplying them, and assigning the result to a variable. In this type of programming, it is particularly easy to adopt three "primitives" concerning control structures: *sequence*, *iteration*, and *selection*.



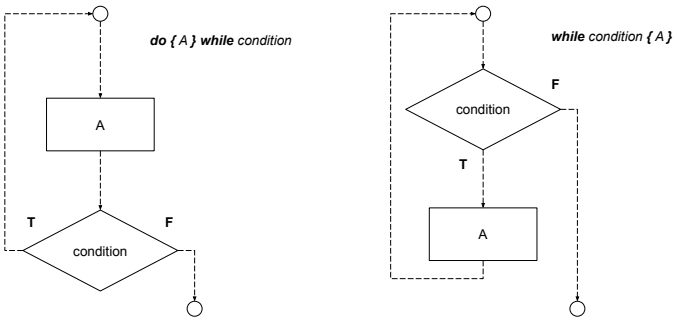
1.3.1 Conventional control structures

In computer science, control structures are essential constructs that dictate the flow of execution in a program. They allow a program to make decisions, repeat actions, and branch into different paths based on conditions. Even if the three primitives are exhaustive for many variations of control structures, there are some convenient structures that often are used in programming languages.

Sequential Execution is the most basic form of control, where instructions are executed one after the other in the order they appear. For example, in a simple program that calculates the sum of two numbers, the operations are performed sequentially.

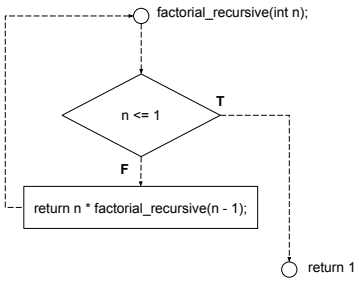


Conditional Statements (`if-then-else`) allow a program to execute certain blocks of code only if specific conditions are met. For example, an if-else statement might check if a user input is valid and then proceed with different actions based on that input.

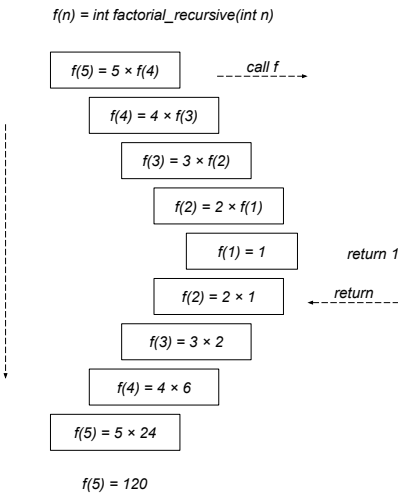


Loops (`for`, `while`, `do-while`) enable the repetition of a block of code multiple times, either a fixed number of times (for loop) or until a condition is met (while loop). For example, a for loop might be used to iterate over an array and perform operations on each element.

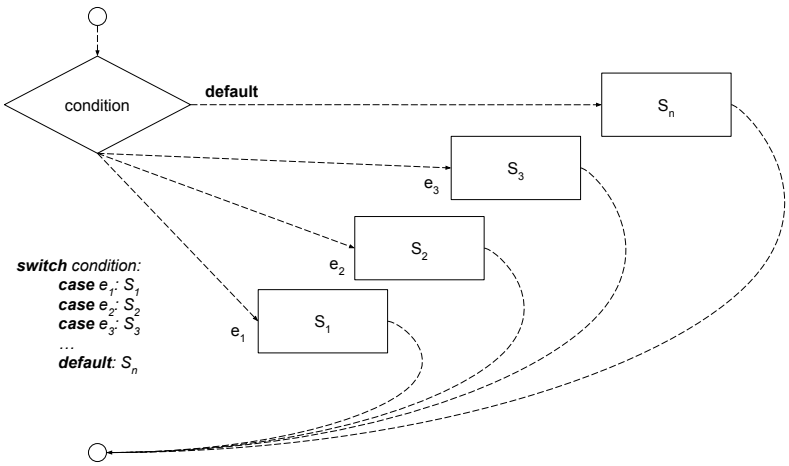
```
int factorial_recursive(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial_recursive(n - 1);
}
```



Recursion is a control structure where a function calls itself to solve smaller instances of the same problem. This is particularly useful for problems that can be divided into similar subproblems, like traversing a tree data structure or calculating the factorial of a number.



Recursive calls building up, and then collapsing back to give the final result. In this example: $5 \times 4 \times 3 \times 2 \times 1 = 120$.



Switch-Case is a multi-way branch statement that allows a variable to be tested for equality against a list of values. Each value, known as a case, leads to a block of code that is executed if the case matches the variable.

1.3.2 Control structures in computers

Here we are closing in on the main topic of virtual machines, which will be addressed in later parts. In computers, control structures are implemented at the hardware and software level to manage the flow of instructions executed by the CPU. The CPU uses a combination of instruction pointers, condition flags, and branching mechanisms to implement these control structures.

Instruction pointers. The CPU executes instructions stored in memory sequentially, using an instruction pointer to track the address of the next instruction. Sequential control is managed by simply incrementing the instruction pointer.

Conditional branching. Conditional control structures like if-else statements are implemented using conditional branching. The CPU evaluates a condition and, based on the result, may branch to a different memory address to continue execution. This is typically done using flags in the CPU's status register, which are set or cleared based on the results of comparisons.

Loops and recursion. Loops in software translate to repeated execution of a set of instructions until a condition is met. The CPU handles this by repeatedly evaluating the loop condition and branching back to the start of the loop if the condition is still true. Recursion is handled similarly, but it involves storing return addresses on the stack to ensure the CPU can return to the correct place after the recursive calls are completed.

Hardware implementation. At a lower level, control structures are embedded in the microarchitecture of processors. Conditional branches, loops, and other control mechanisms are managed by a combination of the CPU's control unit, arithmetic logic unit (ALU), and registers, ensuring efficient execution of complex instruction sequences.

1.3.3 Summary

Control structures

Control structures in programming dictate the flow of execution. The three fundamental structures are sequence, iteration, and selection. Control structures guide how a program makes decisions, repeats actions, and branches into different paths based on conditions.

Sequential execution

Sequential execution refers to the simplest form of control, where instructions are executed one after another in the order they appear in the code. This flow is typical for basic programs, such as calculating the sum of two numbers step-by-step.

Conditional statements

Conditional statements allow a program to choose between different actions based on whether a condition is true or false. The if-then-else structure checks a condition and executes different blocks of code based on the result.

Loops

Loops allow a block of code to repeat either a fixed number of times (as in for loops) or until a condition is met (as in while and do-while loops). Loops are fundamental for performing repetitive tasks, like iterating through an array or list.

Recursion

Recursion is a control structure where a function calls itself to solve a problem by breaking it down into smaller subproblems. It is particularly useful for tasks like calculating factorials or traversing hierarchical data structures (like trees).

Switch-case

The switch-case statement allows for multi-way branching based on the value of a variable. It tests the variable against a list of possible cases and executes the corresponding block of code for the matching case.

Control structures in computers

In computing, control structures are implemented at both hardware and software levels. CPUs manage flow control using mechanisms like instruction pointers, conditional branching, and stack-based recursion. These structures ensure efficient execution of instructions and complex operations.

Instruction pointers

The CPU uses an instruction pointer to track the next instruction to execute. Sequential control is managed by incrementing this pointer after each instruction.

Conditional branching

Conditional branching enables the CPU to change the flow of execution based on evaluated conditions, such as through if-else structures. Branching is achieved using flags in the CPU's status register.

Loops and recursion in hardware

Loops are managed in hardware by repeatedly checking conditions and branching to the beginning of the loop if necessary. Recursion involves storing return addresses on the stack so the CPU can continue execution after recursive calls.

Control in hardware

At the hardware level, control structures are implemented using various CPU components, including the control unit, arithmetic logic unit (ALU), and registers. These components work together to execute instructions, manage branching, and ensure efficient processing of control structures.

1.4 Functions

In mathematics, a function is a relation between a *set of inputs* and a *set of possible outputs* where each input is related to exactly one output. For example, the function $f(x) = x^2$ takes a number x and returns its square. The essential idea is that a function consistently produces the same output for the same input, making it a fundamental concept for understanding relationships between quantities.

However in computer science, a function is a block of organised, reusable code that performs a single, specific task. Functions take input in the form of parameters, process these inputs, and return a result. They are fundamental to structuring programs, allowing for code reuse, modularity, and abstraction. For example, a function in a programming language might take two numbers as input, add them together, and return the sum. Functions help manage complexity in software by encapsulating behaviour that can be invoked repeatedly throughout a program.

What is interesting in the context of practical computers, functions are typically implemented as *sequences of instructions stored in memory that the CPU can execute*. When a function is called, the computer temporarily halts the execution of the current sequence of instructions, switches to the function's instructions, and then returns to the original sequence once the function has completed. Functions in computers can handle everything from basic arithmetic to complex algorithms, and they can call other functions, enabling intricate chains of computation.

Across these domains, mathematics, computer science and practical computers the concept of a function shares a common theme: a *mapping from inputs to outputs*. In mathematics, this is a clear, abstract relationship between sets; in logic, it's about assigning outcomes based on logical rules; and in computer science and computing, it translates these ideas into actionable *instructions that a machine can execute to produce a result*. This shared concept underpins the theoretical foundations of computing and the practical implementation of software, illustrating the deep connections between these fields.

1.4.1 Calling functions

The easiest way of implementing a call in a stack machine is to use a stack. Either a call stack separate from the stack of operations, or using the same stack as the operations. When doing a call, the return address can be kept on the stack, and to get back the address is popped from the stack and the instruction pointer is set by its value. Then the program immediately continues after the calling address.

But often arguments are carried with the function call. You might have heard of the concepts "call by reference," "call by value," "call by name," and other related mechanisms. They are all ways in which arguments (or parameters) are passed to functions in programming languages. These mechanisms determine how a function interacts with the variables passed to it, which in turn affects the behaviour of the program. Here are some often used variations:

Call by Value *The function receives a copy of the actual data. Any changes made to the variable inside the function do not affect the original variable outside the function.*

Call by Reference *The function is passed a direct reference to the original variable, allowing it to modify the variable's actual data in the calling environment.*

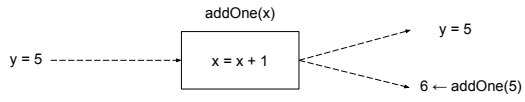
Call by Sharing *Objects are passed by reference, so changes to the object inside the function affect the original, Primitive values are passed as copies, leaving the original values unaffected.*

1. Call by Value

When a function is called by value, the actual value of the argument is passed to the function. This means that the function works on a copy of the data, not the original data. As a result, changes made to the parameter inside the function do not affect the original variable outside the function.

```
! pseudo-code
function addOne(x) {
    x = x + 1;
}

y = 5;
addOne(y);
```



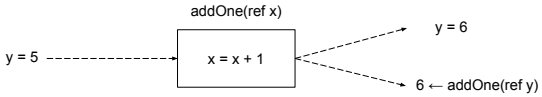
Example: After the function call, 'y' remains '5' because the function 'addOne' only modifies a copy of 'y'.

2. Call by Reference

In call by reference, instead of passing the value of an argument, a reference (or pointer) to the actual data is passed. This means the function can directly modify the original variable, and any changes made to the parameter inside the function will affect the variable outside the function.

```
! pseudo-code
function addOne(ref x) {
    x = x + 1;
}

y = 5;
addOne(ref y);
```



Example: After the function call, 'y' will be '6' because the function modifies the original data through the reference.

Even though C does not pass by reference when call with primitive types, how the referencing and dereferencing works can be enlightening with an example.

```
#include <stdio.h>

// function to add one, using a pointer to pass the variable by reference
void addOne(int *x) {
    *x = *x + 1;
}

int main() {
    int y = 5;
    addOne(&y); // pass the address of y
    printf("y = %d\n", y); // output: y = 6
    return 0;
}
```

3. Call by sharing, or call by object reference

Call by sharing is common in languages like Python and JavaScript. In this approach, references to objects (like lists or dictionaries) are passed to functions,

not the actual objects themselves. This means that the function can modify the contents of the object, and those changes will be reflected outside the function. However, if the reference is reassigned inside the function, it won't affect the original object outside.

For simple data types like integers or booleans, they are passed by value. This means if you try to modify them within the function, only a copy is changed, leaving the original unchanged.

```
! pseudo-code
function modifyList(lst) {
    lst.append(4); ! modifies original list
}

alist = [1, 2, 3];
modifyList(alist);
```

Example: In this case, `alist` is passed by reference, so the `append(4)` operation inside the function modifies the original list. However, if you were to reassign `lst` within the function to a new list (e.g. `lst = [5, 6]`), the original `alist` would remain unchanged outside the function. In Python:

```
def modify_list(lst):
    lst.append(4) # modifies the original list

alist = [1, 2, 3]
modify_list(alist)
print(alist) # output: [1, 2, 3, 4]

def reassign_list(lst):
    lst = [5, 6] # does not affect the original list

reassign_list(alist)
print(alist) # output: [1, 2, 3, 4]
```

1.4.2 Summary

Functions

In mathematics, a function is a mapping from inputs to outputs where each input corresponds to one output. In computer science, a function is a reusable block of code that takes input (parameters), processes it, and returns an output. This concept is fundamental to structuring software by promoting modularity and reuse.

Calling functions

In programming, functions can be called using different argument-passing mechanisms, such as call by value, call by reference, and call by sharing. These mechanisms determine how the function interacts with the variables passed to it and whether or not it can modify the original variables.

Call by value

In call by value, a copy of the argument's value is passed to the function, and changes made to the parameter inside the function do not affect the original variable outside the function.

Call by reference

In call by reference, a reference (or pointer) to the original variable is passed to the function, allowing the function to modify the original data directly. Changes made to the parameter inside the function will affect the original variable outside.

Call by sharing

In call by sharing (or call by object reference), a reference to an object is passed to the function, allowing the function to modify the object. However, if the reference is reassigned inside the function, the original object remains unchanged. Primitive types like integers are passed by value, while complex types like lists or dictionaries are passed by reference.

1.5 Practice

Workbook: Chapter 1.
https://github.com/Feyerabend/bb/tree/main/workbook/ch01
Scan the QR code to access exercises, code, resources and updates.

The first chapter of this book outlines some of the groundwork for understanding fundamental programming concepts. It begins by introducing key elements such as variables, control structures, and functions. It also mentions simple data types, illustrating how different data forms—such as integers, floating-point numbers, characters, and strings—are represented in binary. It highlights the significance of understanding binary representation.

However, the landscape of programming is vast, encompassing numerous fundamental concepts, branches of study, and pathways for further exploration. To navigate this complexity, you should tailor your learning journey based on existing knowledge and specific interests. Consequently, the questions one poses may vary widely depending on these factors.

Example: Floating-point numbers

After reading the section on "Floating-point numbers," consider leveraging tools like Large Language Models (LLMs) to investigate the subject more thoroughly. Here are some thought-provoking questions you might explore:

1. *What is the standard 'IEEE 754'?* Understanding this standard is beneficial for grasping how floating-point numbers are represented and manipulated in computer systems.

2. *How does arithmetic works with floating point numbers?* Delve into the intricacies of floating-point arithmetic, including addition, subtraction, multiplication, and division.

3. *Provide an implementation in Python of floating point numbers arithmetic, but not using built-in operations.* Implementing basic arithmetic manually can deepen your understanding of how floating-point representation functions.

More extensive projects can also be found at the repository. By engaging with these questions and projects, you can further your exploration of floating-point numbers and the broader realm of programming concepts, enhancing your overall comprehension and skills.

Example: Koch snowflakes

Niels Fabian Helge von Koch (1870–1924): Von Koch is famous for creating the Koch snowflake, one of the earliest-recognized fractals. The construction of this snowflake involves a recursive process—each side of a triangle is repeatedly divided to create an infinitely complex boundary. This idea directly links to the concept of recursion, where a process calls itself with smaller subsets, similar to recursive functions in programming. Fractals like the Koch snowflake have inspired recursive algorithms used in computer graphics and complex problem-solving.

Ask a LLM: Can you tell me something about Koch and Koch snowflakes?

Read and ask follow-up questions to enhance you knowledge.

Simple exercise: *Drawing the snowflake with simple loops.*

Objective. Use basic programming loops to create the outline of the Koch snowflake without recursion.

Task. The Koch snowflake can be created by starting with a simple triangle and then applying a transformation to each side. In this task, you will use loops to approximate the Koch snowflake shape based on a list of points for each side of the triangle. Before making a program, explore suitable points on a piece of graph paper, and also draw first iteration. Going to the program:

1. Start with three points that form an equilateral triangle. Investigate suitable start of the coordinates of these three points.

2. Manually add a few extra points along each side, forming a "V" shape to simulate the fractal's look.

3. Write a loop that connects all these points in sequence to form an outline of the Koch snowflake.

4. Experiment with repeating these loops on smaller segments (like halfway points) to create a rough approximation of more fractal detail.

Goal. Write a Python program that loops through the list of points and draws a rough outline of the snowflake using only for loops and print statements or a simple graphics library. This exercise will help you understand the symmetry and structure of the Koch snowflake without requiring recursion.

Advanced exercise: *Koch snowflake with dynamic levels of recursion.*

Objective. Create a program that adjusts the Koch snowflake's recursion depth in real time based on user input.

Task. The Koch snowflake is a fractal shape that gets more complex with each level of recursion. In this task, you will write a Python program that allows the user to specify the recursion depth of the snowflake and then display it on the screen. You'll use recursion to break each side into more detailed segments based on the chosen depth.

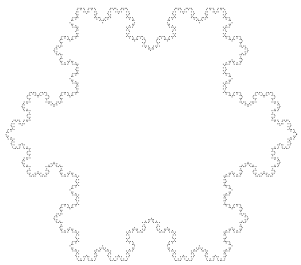
1. Step 1: Start by defining a function that draws a single Koch curve segment based on a starting and ending point, and a specified depth.

2. Step 2: Write a function that will ask the user for an integer value representing the depth. This value will determine the number of times your Koch function is called recursively, making the snowflake more complex with higher values.

3. Step 3: Use this depth value to adjust the number of recursive calls in your program, regenerating the snowflake each time the user inputs a new value. If possible, use a simple graphics library to display the snowflake visually and refresh the display after each input.

4. Step 4: Experiment with different recursion depths and observe how the complexity changes. What happens when you use a depth of 1, 2, 3, and so on?

Goal. The goal of this exercise is to help you understand how recursion depth affects fractal detail. Write a program in JavaScript that dynamically updates the Koch snowflake based on user input. Focus on how each additional recursion level changes the visual complexity of the shape and reflect on the computational cost as the depth increases.



Chapter 2

Understanding VMs

2.1 Simple VMs

A *programming language virtual machine* (VM) is an abstract model that simulates a computer for executing programs. This book focuses on this specific model, as other concepts like *cloud computing* (also referred to as 'VM') do not illustrate the same principles and fall outside our scope. Henceforth, *VM* will refer to the programming language virtual machine.

The programming language virtual machines can be implemented in various forms, such as stack-based or register-based architectures. Stack machines use a simple memory structure called Last In, First Out (LIFO), which pushes and pops data during execution. This approach simplifies the execution of operations, using a method from a notation called reverse Polish notation (RPN), where operators come after the operands, making it easier to evaluate expressions without needing parentheses for operator precedence.

The virtual machine operates through an interpreter, which follows a continuous cycle of fetching instructions, identifying them, executing the corresponding operations, and then moving to the next instruction. This fetch-execute cycle forms the backbone of how virtual machines process commands in a sequence, making them highly adaptable for different types of programs.

Note that the code provided here serves as illustrations of specific concepts and is not intended for further development. It reflects the particular perspectives being emphasized while deliberately omitting others. To start with, we will take a look at a simple example of a stack-based VM is VM1, which performs basic arithmetic operations like addition. In this machine, instructions are pushed onto a stack, executed, and then the results are either printed or the program is halted. In contrast, REGVM, a more complex register-based machine, operates directly on registers, providing more flexibility and efficiency for tasks such as loops and conditional statements. By directly accessing registers, REGVM can perform operations faster and with more control over program flow.

Virtual machines are highly portable, providing an abstraction layer that makes programs independent of the underlying hardware. This allows code to run consistently across different platforms and environments.

2.1.1 The stack

The stack works as a Last In, First Out (LIFO) memory. Below is an illustration of how the stack evolves during execution of simple operations like addition. First we have put on the stack some numbers, where they are stacked from the bottom up. First we push number ↑ '1' onto the stack:

```
1
```

Then we put number ↑ '2', pushed on to the stack:

```
1
2
```

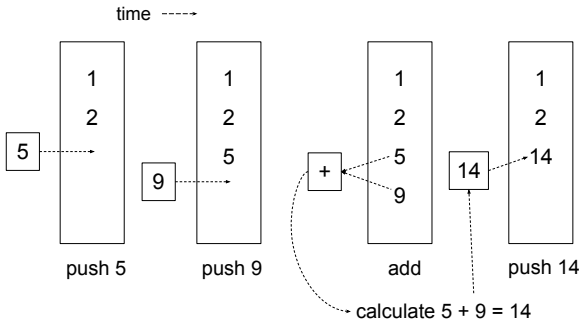
Let's say we have pushed also ↑ '5' and after that ↑ '9' on the stack:

```
1
2
5
9
```

If we use an operator called 'addition' which takes (pops) two integers and adds them, and after the addition push ↑ the resulting number back on the stack:

```
 1
 2
14
```

The machine works with a what is called reverse Polish notation (RPN), where operations are written after the operands, e.g. "5 9 +". However in this case the addition operator isn't included on the stack, which it sometimes can be. The RPN simplifies expression evaluation and eliminates the need for parentheses, for capturing operator precedence.



2.1.2 Interpreter technique

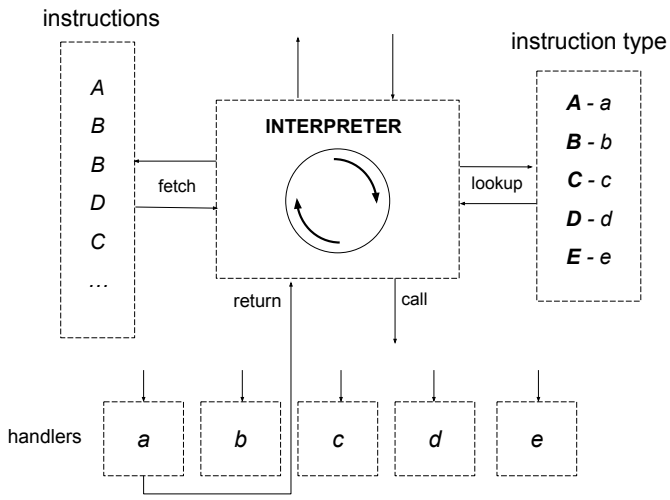
A virtual machine interprets the instructions as follows:

i. Fetch the next instruction.

ii. Look up the instruction type.

iii. Execute the corresponding handler.

iv. Return to the interpreter for the next instruction (go to i).



Thus in this simple stack machine case, the interpreter fetches instructions that uses the stack or operates on the stack in a sequence.

2.1.3 VM1 implementation

So a simple stack machine should emulate a program that operates some arithmetic on numbers on a stack, e.g. A sample program in our first machine VM1, data written in a C array with support of constants for instructions, looks as follows:

```
int program[] = {
  SET, 33,
  SET, 44,
  ADD,
  PRINT,
  HALT
};
```

The instructions in this program push two numbers onto the stack (33 and 44) with a instructions called 'SET', adds them 'ADD', and then print the result (77) with 'PRINT'. Then it stops 'HALT'.

The corresponding operations in an "assembly" notation:

```
SET 33    ; Push 33 on stack
SET 44    ; Push 44 on stack
ADD       ; Add the top two numbers (44 + 33)
PRINT     ; Print result (77)
HALT      ; Halt program
```

To run the program, compile and execute both the virtual machine 'vm1.c' and the included hardcoded program, where ('>' marks the command line prompt):

```
> gcc vm1.c -o vm1
> ./vm1
77
> _
```

Details

The VM supports a small set of instructions, each represented by (or corresponds to) an integer opcode, the 'machine code':

- `HALT`: Stops execution.
- `SET`: Pushes a value onto the stack.
- `ADD`: Pops two values from the stack, adds them, and pushes the result back.

- `SUB`: Pops two values, subtracts them, and pushes the result back.
- `MUL`: Pops two values, multiplies them, and pushes the result back.
- `PRINT`: Pops a value from the stack and prints it.

The complete C-program can be studied on-line, see Section 2.4.

2.1.4 REGVM implementation

Let us for a moment have a look at another a more complex implementation. You might be more familiar with register-based machines, if you have experience with machine code or assembly language on a typical PC. The assembly code for register machines and stack machines will naturally differ. In practice, many machines combine elements of both architectures, allowing programs to leverage the strengths of each approach. The A register can in this case be looked upon as an integer variable, from above.

Instructions pointer and program counters

In the programs 'regvm.c' and 'regvm.py' you will see how simple virtual register machines work, see Appendix **??**. You are encouraged to compare the register machine and the stack machine, particularly in how they manage execution order. In this register machine, instructions are processed sequentially, with the instruction pointer advancing step by step. The pointer might change when jumps, returns etc. occurs, but it is still sequential. The fetch cycle retrieves the instruction from the current pointer location, along with any optional arguments, *and then increments the pointer* to the next instruction.

In contrast, the stack machine uses a program counter, which does not distinguish between instructions and arguments. Each instruction may be followed by one (or maybe more arguments), which are fetched during the execution of the instruction. The program counter is first incremented, *and then the code at that location is fetched*, regardless of its type.

The differences between these machines affect how operations like jumps are handled with addresses. For example, in the register machine REGVM, the instruction pointer can be directly set to a specific address to perform a jump. In contrast, in this stack machine VM1, the program counter manages jumps by adjusting itself based on the next instruction, without distinguishing between addresses and arguments. This distinction leads to slight variations in how jumps and other control flow operations are executed.

Factorial sample

Compiling and running the C-program:

```
> gcc -o regvm regvm.c
> ./regvm
```

```
Register A: 120
> _
```

Running the Python-program:

```
> python3 regvm.py
Register A: 120
> _
```

The register A is 120 as the result from calculating the factorial of 5, which can be done in assembly:

```
1  MOV A 1
2  MOV B 5
3  CMP B 0
4  JZ 7
5  MUL A B
6  SUB B 1
7  JMP 2
8  PRINT A
```

1. **MOV A 1** Load register `A` with the value `1`. This register will store the result of the factorial.
2. **MOV B 5** Load register `B` with the value `5`. `B` will hold the current number to multiply.
3. **CMP B 0** Compare the value in `B` with `0`. This checks if the loop is complete.
4. **JZ 7** Jump to instruction 7 (`PRINT A`) if `B` equals `0`, ending the loop.
5. **MUL A B** Multiply the value in `A` by `B`, storing the result in `A`. This performs a step of the factorial calculation.
6. **SUB B 1** Subtract `1` from `B`, preparing for the next loop iteration.
7. **JMP 2** Jump back to instruction 2 (`CMP B 0`) to continue the loop until `B` equals `0`.
8. **PRINT A** Print the value in `A`, which now holds the result of the factorial calculation.

In a register-based machine, the focus shifts from manipulating a stack to direct operations on registers, which function similarly to variables. This change has a significant impact on how the machine handles instructions, particularly with branching and control flow.

A summary of instructions for the register machine REGVM can be seen in Table 2.1.

Registers and direct access

Unlike a stack machine, where data is implicitly stored and retrieved in a last-in-first-out (LIFO) manner, registers provide explicit, direct access to data. Each register has a specific name (such as `A`, `B`, etc.), and operations reference them directly. This allows the machine to:

- Perform operations without needing to push/pop from a stack.
- Store intermediate results in registers for later use.
- Efficiently execute repetitive tasks like loops or conditional operations without the overhead of stack manipulation.

In the factorial example, registers `A` and `B` act as placeholders for the running result and the current multiplier, respectively.

Comparisons and conditional jumps

An important addition to the register machine is the ability to compare register values and make decisions based on the outcome. In a stack machine, control flow may rely on the state of the stack, but in a register machine, comparisons provide more flexibility and transparency:

- `CMP B 0` compares the value in register `B` with `0`. This comparison is crucial for controlling the loop.
- `JZ 7` checks the result of the comparison. If `B` is zero, the machine jumps to instruction 7, ending the loop and proceeding to print the result. Otherwise, the loop continues.

This comparison-based approach enables structured control flow, such as:

- Conditional jumps, where execution continues at a different point in the program if a condition is met.
- Loops, where the program can return to an earlier instruction (like `JMP 2`) to repeat a sequence of operations until a condition changes.

Jump instructions and control flow

In a register machine, jumps are essential for implementing loops and conditional logic. By using jumps, the machine can control which instructions to execute next based on the values in the registers:

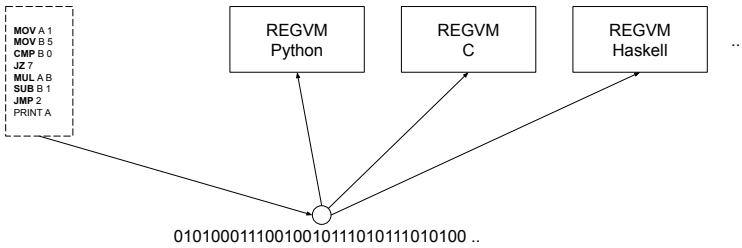
- `JMP 2` sends the program counter back to instruction 2, allowing the program to repeat the multiplication and subtraction until `B` reaches zero.
- This type of jump loop is common in register machines, and the jump mechanism can be used to create both conditional loops (like this one) and more complex control structures (e.g. nested loops or function calls).

Instruction	Flags	Example
`MOV <reg> <val>`	-	`MOV A 1` sets `A` to 1.
`ADD <reg> <val>`	`Z`, `N` updated	`ADD A B` adds `B` to `A`.
`SUB <reg> <val>`	`Z`, `N` updated	`SUB A 1` subtracts 1 from `A`.
`MUL <reg> <val>`	`Z`, `N` updated	`MUL A B` multiplies `A` by `B`.
`CMP <reg> <val>`	`Z` set if values are equal	`CMP B 0` sets `Z` if `B` is 0.
`JMP <addr>`	-	`JMP 4` jumps to address 4.
`JZ <addr>`	-	`JZ 8` jumps to address 8 if `Z`.
`PRINT <reg>`	-	`PRINT A` prints the value in `A`.

Table 2.1: REGVM instructions with flags and examples

2.1.5 Portability

In the realm of software development, virtual machines play a crucial role by providing an abstraction layer that emulates a physical computer. This abstraction allows programs to run in a consistent environment that is isolated from the underlying hardware and operating system. The primary advantage of this abstraction is *code portability*—the capability of software to operate across different computing environments without requiring modification.



Here with REGVM a virtual machine abstracts the details of the host hardware by offering a uniform set of registers and an execution environment, but naturally a stack machine also follows the precise same rules as it is also portable. These kinds of uniformity ensures that programs written for the VM are not tied to any specific hardware configuration, thus allowing them to be executed consistently across various platforms. For instance, by implementing a virtual machine REGVM in both C and Python, we demonstrate that code designed for the VM can run seamlessly in different virtual environments, reinforcing the concept of portability.

The use of virtual machines offers several practical benefits. It allows developers to create applications that can function across multiple operating systems or hardware platforms, eliminating the need for separate versions of the software for each environment. This capability is particularly valuable for reaching

a broad user base with diverse system configurations. Moreover, VMs facilitate the running of legacy software on modern hardware, preserving access to older applications that were designed for discontinued operating systems.

In addition, VMs enhance security by providing an isolation layer between applications and the host system. This isolation helps mitigate the risk of vulnerabilities by preventing direct interactions with the host's hardware and operating system. Furthermore, the controlled environment of a VM simplifies the development and testing processes, as developers can ensure that their applications behave consistently across different systems.

Virtual machines embody an abstract machine model, which standardizes the execution environment by defining a consistent set of instructions and registers. This model allows developers to focus on programming for the VM rather than dealing with hardware-specific details. Additionally, VMs often use intermediate representations (IR) of code, such as *bytecode*, which can be compiled or interpreted to run on various hardware platforms, further supporting code portability.

2.1.6 Summary

A virtual machine (VM) is essentially an abstract model of a computer that can run programs. It can be designed in various ways, such as a stack-based machine or a register-based machine. Stack machines are simpler and rely on a Last In, First Out (LIFO) memory model, whereas register machines use a set of registers for data manipulation. This section introduces the concept of virtual machines and focuses initially on stack machines, which will be explained in detail.

The stack

The stack in a virtual machine operates as a LIFO structure, where data is added and removed in a sequential manner. Instructions are executed using reverse Polish notation (RPN), meaning operators come after operands. For example, adding two numbers involves pushing them onto the stack, then popping and adding them, and finally pushing the result back onto the stack. This simple mechanism eliminates the need for managing operator precedence with parentheses.

Interpreter technique

The interpreter in a virtual machine repeatedly follows a simple fetch-execute cycle. It retrieves the next instruction, identifies its type, executes the corresponding operation, and returns to fetch the next instruction. This cycle continues until the program ends, forming the core of how a VM processes commands in sequence.

VM1 implementation

VM1 is a simple stack-based virtual machine that executes basic arithmetic instructions. A sample program pushes two numbers onto the stack, adds them, prints the result, and halts. The VM1 system translates these operations into machine instructions (opcodes), such as 'SET' to push values and 'ADD' to sum two numbers. The VM can be implemented in C and supports basic instructions like addition, subtraction, multiplication, and printing.

REGVM implementation

REGVM is a more complex implementation based on a register machine. Unlike a stack machine, which relies on implicit memory (LIFO stack), a register machine uses explicit registers for data storage and manipulation. Instructions are executed in sequence, and the machine directly modifies the registers. Programs in REGVM are more efficient for operations like loops or conditional statements, as registers allow for direct access to values and more flexible control flow.

Factorial sample

A sample program demonstrates how a register machine calculates the factorial of a number. It uses a loop with conditional jumps and comparisons to multiply numbers stored in registers until the factorial is computed. This showcases how register-based VMs handle iterative processes and manage control flow through direct manipulation of the instruction pointer.

Portability

Virtual machines provide an abstraction layer that enables code portability across different platforms. By standardizing the execution environment, VMs like REGVM and stack machines allow programs to run consistently, regardless of the underlying hardware. This abstraction also enhances security, simplifies development, and supports legacy software. The portability of code across different environments is one of the key benefits of using virtual machines in software development.

2.2 Stack-based VM

We will now shift our focus from register-based machines back to stack-based machines. This time, however, we are working with a higher level of abstraction than the typical machine code we associate with traditional CPUs. The design of VM2 is inspired by the programming language Forth[1], which emphasizes stack-based operations. The fundamental operations of VM2 closely resemble

[1] See: https://en.wikipedia.org/wiki/Forth_(programming_language). Forth have many interesting properties besides being a stack-based. One such is the closeness to the machine, which we use here.

the stack-based behavior and low-level machine language constructs found in Forth, making it a useful model for simulating high-level tasks with a stack-oriented architecture.

Fibonacci sample

In VM2 we can implement the Fibonacci series: $1, 1, 2, 3, 5, 8, 13, \ldots$

```
0 + 1 = 1
    1 + 1 = 2
        1 + 2 = 3
            2 + 3 = 5
                3 + 5 = 8
                    5 + 8 = 13
                        8 + 13 = ..
```

This using some simple basic stack-based operations:

```
TWODUP
ADD
ROT
DROP
```

Running iterations of these instruction, the series can be recognised in the starting position (a), then at (e) and at (i) representing the stack after operations in two interations:

(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	..
1	1	1	2	2	2	2	3	3	
1	1	1	1	1	1	1	2	2	
	1	2	1		2	3	1		
	1				1				

The TWODUP operation duplicates the top two stack items, ADD adds the top two numbers, ROT rotates the stack, and DROP discards the top, or the most recently pushed, item on the stack.

TWODUP operation duplicates the top two elements of the stack. This is particularly useful in stack-based languages like FORTH because it allows you to preserve values on the stack for further use in calculations. Here's how it is implemented i C:

```
case TWODUP:
    b = pop(vm);
    a = pop(vm);
    push(vm, a);
    push(vm, b);
    push(vm, a);
    push(vm, b);
    break;
```

```
(before)          (after)
  2                 2
  67                67
  9                 9
                    67
                    9
```

ROT operation shifts the top three elements of the stack in a circular fashion. This operation allows you to reorganize the stack without disturbing the rest of the data. Specifically, it takes the third element from the top and moves it to the top, while pushing the other two elements down one position.

```
case ROT:
  c = pop(vm);
  b = pop(vm);
  a = pop(vm);
  push(vm, b);
  push(vm, c);
  push(vm, a);
  break;
```

```
(before)          (after)
  2                 67
  67                9
  9                 2
```

DROP operation discards the top element of the stack. This is useful when a value is no longer needed, and you want to reduce the stack size. The implementation is just to pop the "top" of the stack, and not doing anything with the value.

```
(before)          (after)
  2                 2
  67                67
  9
```

2.2.1 Comparisons

The comparisons in this VM2 works with the stack. They are as before used to check the relationship between two values (e.g. whether one is smaller than the other, or if two values are equal). These comparisons result in either TRUE (1) or FALSE (0) being pushed onto the stack, which is then used for making decisions, like whether to jump or not.

Less than LT:

```
case LT:
    b = pop(vm);
    a = pop(vm);
```

```
    push(vm, (a < b) ? TRUE : FALSE);
    break;
```

LT pops two values, 'a' and 'b', from the stack. It checks if 'a' is less than 'b' and pushes TRUE (1), if 'a $<$ b', otherwise it pushes FALSE (0).

Equal EQ:

```
case EQ:
    b = pop(vm);
    a = pop(vm);
    push(vm, (a == b) ? TRUE : FALSE);
    break;
```

EQ pops two values, 'a' and 'b', from the stack. It checks if 'a' is equal to 'b'. It pushes TRUE if 'a $==$ b', otherwise it pushes FALSE.

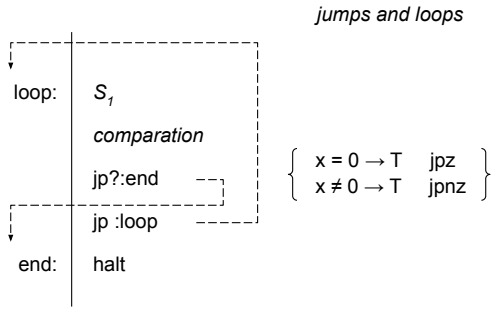
Equal to zero EQZ:

```
case EQZ:
    a = pop(vm);
    push(vm, (a == 0) ? TRUE : FALSE);
    break;
```

EQZ pops one value, 'a', from the stack and checks if 'a' is equal to zero. It pushes TRUE if 'a $==$ 0', otherwise FALSE.

2.2.2 Iterations

The comparison results (TRUE or FALSE) are usually used with conditional jumps (JPZ, JPNZ). For example: If the result is TRUE, the program might jump to a different part of the code. If it's FALSE, it will continue executing the next instructions. In this way, the VM can make decisions based on comparisons, such as repeating loops or branching to different parts of the program.



Jumps are used to change the flow of the program, allowing it to go to a different instruction rather than just moving to the next one.

Unconditional jump: JP makes the program jump directly to a specified instruction.

```
case JP:
    vm->pc = nextcode(vm); // set the program counter (pc) to a new address
    break;
```

This moves the program counter (pc) to a different part of the code, skipping everything in between.

Conditional jump: JPZ or JPNZ jumps only if a condition is met (e.g. if a value is zero or non-zero).

```
case JPZ:
    addr = nextcode(vm); // get the address to jump to
    v = pop(vm); // pop a value from the stack
    if (v == 0) { // jump only if the value is zero (or FALSE)
        vm->pc = addr;
    }
    break;
```

This checks a value, and if it meets the condition, the program jumps to the given address. If not, it just continues as normal.

Loops allow the program to repeat a block of code. In this VM, loops are often created by combining conditional jumps with a block of code. For example, you might pop a value, check if it's zero, and if not, jump back to repeat the block.

2.2.3 Error handling

So far, we have kept error handling to a minimum to avoid the additional code it often generates, which can be distracting when reading through the main logic. Therefore, we include this note here to address error handling considerations.

When writing software, it's *essential* to expect and plan for errors. Errors are a natural part of programming as a craft—they can happen for many reasons, such as incorrect inputs, running out of memory, or trying to access invalid memory locations. By handling errors gracefully, a program can prevent crashes and give helpful feedback instead of failing unexpectedly. Bugs can also be detected earlier in the process.

Memory allocation: If the VM fails to allocate memory (e.g. during a 'malloc()' call), it should detect this and handle the error by either exiting the program or displaying a clear error message. Example: After calling 'malloc()' to allocate memory for the stack, you should check if the returned pointer is 'NULL'. If it is, print an error and stop the program safely:

In program VM2:

```
    vm->stack = (int*) malloc(sizeof(int) * STACK_SIZE);
    if (vm->stack == NULL) {
        return NULL;
    }
```

Can be swapped for a new one:

```
vm->stack = (int*) malloc(sizeof(int) * STACK_SIZE);
if (vm->stack == NULL) {
    fprintf(stderr, "Error: Unable to allocate memory for the stack.\n"
        );
    exit(1);
}
```

Stack overflow or underflow: When pushing or popping from the stack, you need to ensure that you don't exceed the stack's capacity (overflow) or pop an item from an empty stack (underflow). Error handling can check the stack's boundaries and issue a warning or stop the program if these limits are violated.

```
int pop(VM* vm) {
    int sp = (vm->sp)--;
    return vm->stack[sp];
}

void push(VM* vm, int v) {
    int sp = ++(vm->sp);
    vm->stack[sp] = v;
}
```

Can be replaced by the more informative:

```
int pop(VM* vm) {
    if (vm->sp < 0) {
        fprintf(stderr, "Error: Stack underflow\n");
        exit(EXIT_FAILURE); // terminate program (or optionally return
            code)
    }
    return vm->stack[(vm->sp)--];
}

void push(VM* vm, int v) {
    if (vm->sp >= STACK_SIZE - 1) {
        fprintf(stderr, "Error: Stack overflow\n");
        exit(EXIT_FAILURE); // terminate program (or optionally return
            code)
    }
    vm->stack[++(vm->sp)] = v;
}
```

Opcode errors: In a VM, you might run into invalid instructions or unsupported opcodes. Adding a basic check can ensure that the VM either ignores the unknown instruction or stops with an informative error message. Example: In the 'switch' statement that processes opcodes, handle the default case by reporting an invalid opcode:

```
default:
    break;
```

Replace with:

```
    default:
        fprintf(stderr, "Error: Unknown instruction at PC %d\n", vm->pc);
        exit(1);
```

Common error-handling techniques

- *Return codes.* Functions can return special values (e.g. '-1', 'NULL') to indicate that something went wrong (or had success). The calling code should check these values and respond accordingly.
- *Error messages.* When an error is detected, print a helpful message so the user or developer knows what went wrong and where to start troubleshooting.
- *Exiting gracefully.* If the program encounters a critical error that prevents further execution, use a function like 'exit()' to stop the program gracefully, providing a meaningful exit code.

In C and similar languages, these basic techniques serve as the foundation for error management. In languages such as C++ or Python they often rely on exception handling, and other more advanced mechanisms.

We will later on discuss error handling in another more hardware related context, see Section 4.9.

2.2.4 Summary

VM2, inspired by the stack-based language FORTH, focuses on stack operations to simulate high-level tasks. It provides a higher abstraction than traditional CPU machine code, making it effective for tasks suited to stack-oriented architectures.

Fibonacci sample

The implementation of the Fibonacci series in VM2 uses basic stack-based operations such as TWODUP, ADD, ROT, and DROP. These operations manipulate the stack to produce the Fibonacci sequence by repeatedly adding the two topmost numbers and adjusting the stack with rotation and duplication. The process is iterative, with the stack reflecting the series after each set of operations. The TWODUP duplicates the top two elements, ADD sums the top two numbers, ROT rotates the stack, and DROP removes unnecessary values.

Comparisons

VM2 implements comparisons like "less than" (LT), "equal" (EQ), and "equal to zero" (EQZ) using the stack. These operations pop values off the stack, perform the comparison, and push either TRUE (1) or FALSE (0) back onto

the stack. For example, LT checks if one value is less than another, while EQ checks for equality, and EQZ checks if a value equals zero. These comparison results can be used with conditional jumps (JPZ for zero, JPNZ for non-zero), enabling control flow such as loops or decision-making in the program.

Iterations

Jumps control program flow in a virtual machine (VM). Unconditional jumps (JP) redirect the program counter (pc) to a specified address, skipping intermediate instructions. Conditional jumps (JPZ or JPNZ) occur only if a condition is met, such as jumping if a value is zero (JPZ) or non-zero (JPNZ). Loops are created by combining conditional jumps with code blocks, allowing the program to repeat instructions based on evaluated conditions (comparisons).

Error handling

Error handling is essential in programming, though here minimised for simplicity. Errors like memory allocation failures (e.g. during 'malloc()') can be handled by checking if memory allocation returned 'NULL', then printing an error and exiting the program. Stack overflow or underflow is addressed by ensuring that pushes and pops stay within the stack's capacity, and providing clear error messages when limits are violated. Additionally, opcode errors are managed by checking for invalid instructions and printing informative error messages before halting execution. Common error-handling techniques include returning error codes, printing descriptive messages, and exiting gracefully to ensure a robust and fault-tolerant program.

2.3 Memory and functions

VM3, below, introduces more advanced features compared to VM1 and VM2, including jump instructions, comparison operators, and memory storage, which allow for loops, conditionals, and function calls. The machine uses *activation records* to handle function calls and returns, storing arguments, local variables, and return addresses on the stack.

Assembly, machine code and bytecode

The role of an assembler is to translate this assembly code into machine code, which consists of binary instructions that the hardware can directly execute. This process maps the mnemonics to their corresponding binary representations, handles labels, and manages data addresses. The 'mnemonics' is here the assembly language.

However, VM3 operates differently. Instead of letting the assembly code directly be represented as some machine code (integers), the assembler converts text into what is known as *bytecode*. Bytecode is an intermediate code format

used by the virtual machine. Unlike machine code, which is specific to hardware, bytecode is designed to be executed by the virtual machine, which provides a layer of abstraction between the code and the underlying hardware.

This abstraction is significant because it allows programmers to write assembly code without needing to worry about the specifics of the machine code for different hardware platforms. The assembler simplifies this process by converting the assembly language into bytecode, which the virtual machine can then interpret and execute. This approach offers greater portability and flexibility in managing code execution across various environments.

Example: Prime numbers

```
1   # prime numbers generator
2
3   INIT:
4      SETZ
5      ST 0 # c = 0
6      SET 1
7      ST 1 # j = 1
8
9   LOOP:
10     LDARG 1. # i
11     LD 1 # j
12     MOD # i % j
13     EQZ # = 0 ?
14
15     JPZ :NOINC # jump if = 0 false
16
17     LD 0  # c ->
18     INC  # c++
19     ST 0  # -> c
20
21  NOINC:
22     LD 1  # j ->
23     INC   # j++
24     ST 1  # -> j
25
26     LD 1  # j
27     LDARG 1  # i
28     LT  # j < i ?
29     JPNZ :LOOP # true, loop
30
31     LD 1  # j
32     LDARG 1  # i
33     EQ  # j = i ?
34     JPNZ :LOOP # true, loop
35
36     LD 0  # c
37     SET 2
38     EQ  # c = 2 ?
39     JPZ :EXIT # no? exit call
40
41     LDARG 1  # prime arg
```

```
42      PRINT  # print
43
44   EXIT:
45      DROP  # drop excess on stack
46      RET
47
48   START:
49      SET 99
50      STORE 0 # n = 99
51
52      SET 1
53      STORE 1 # i = 1
54
55   NEXT:
56      LOAD 1  # i
57      INC  # i++
58      DUP  # i i
59      STORE 1 # i global reg
60
61      STARG 1  # i arg for call
62      CALL :INIT  # call with 1 argument on stack
63
64      LOAD 1  # i
65      LOAD 0  # n
66      LT   # i < n ?
67      JPNZ :NEXT # true, next
68
69      LOAD 0
70      LOAD 1
71      EQ   # i = n ?
72      JPNZ :NEXT # true, next
73
74      HALT
```

The program's goal is to find and print prime numbers in the range from 1 to 'n' (99 in this case), by checking each number to see if it's prime. A number is prime if it has exactly two divisors: 1 and itself. This is accomplished by counting the number of divisors (variable 'c'), and checking if it's exactly 2 for each number.

This algorithm could be described in pseudo-code.

```
for i from 2 to n:
    count = 0 # init divisor count
    for j from 1 to i:
        if i % j == 0:
            count += 1 # Increment count, if j divides i
    if count == 2: # only two divisors (1 and itself)
        print(i) # i is prime, so print
```

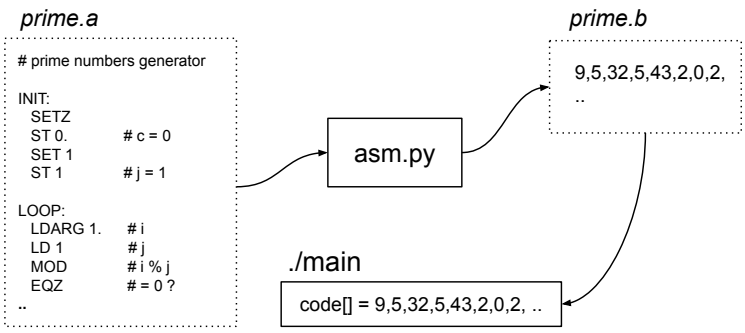
Key sections in implementation

- INIT line 3 to 7

  - Initializes 'c ' (divisor count) and 'j' (current divisor candidate) to zero and one, respectively, before starting the loop that checks for divisibility.

- LOOP line 9 to 19
  - This section checks if 'i % j = 0'. If true, 'c' is incremented.
  - 'j' is then incremented, and the program checks whether 'j < i' to continue finding divisors.
  - When 'j = i', the loop ends, and the program checks if 'c = 2'; if so, 'i' is printed as a prime number.
- NEXT line 55 to 74
  - Increments 'i', checks if 'i < n', and repeats the process if true.

This is a straightforward algorithm for checking prime numbers, but it can be optimised in several ways for better performance. One such is skipping even numbers after 2, another is that storing and loading numbers can be reduced. We will return to optimisation later on, see Section 3.4.

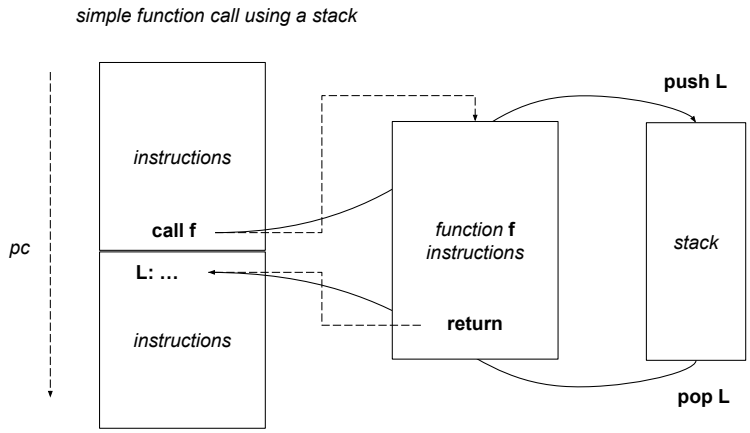


Compilation and execution

The assembling of 'prime.a' to 'prime.b' and then running, works through a make file to ease development. The 'vm3.c' is also compiled in these steps.

```
> make FILE=prime
> python3 ./asm.py -i prime.a -o prime.b
> ./main prime.b
> loading ..
> code length = 84
> running ..
```

```
> - - - - - - - - - - - -
> 2
> 3
...
> 97
> - - - - - - - - - - - -
> duration 0.000494 seconds
> done running.
> _
```



2.3.1 Frame pointer

VM3 adds complexity to the virtual machine by supporting function calls, but also the previous loops, and conditionals. In this virtual machine implementation, the *frame pointer (fp)* plays a crucial role in managing function calls, particularly for local variables and arguments. Let's break down how the frame pointer is used and how it interacts with the call stack.

Calling a function: When a function is called, the VM saves the current frame pointer and program counter on the stack, establishes a new frame pointer (pointing to the current top of the stack), and then jumps to the function's code.

Returning from a function: The VM restores the previous frame pointer and program counter (return address) from the stack, effectively discarding the current stack frame and resuming execution from where the function was called.

The frame pointer is used to keep track of the current "stack frame," which contains local variables, arguments, and return addresses for a particular function call. Each function call creates a new stack frame, and the frame pointer

ensures that the virtual machine can access the correct local variables and arguments relative to that frame.

Frame pointer ('fp') in the code:

```
vm->fp = 0;
```

The frame pointer is initialized to '0' when the virtual machine is created. This means the initial stack frame starts at the beginning of the stack.

Function **CALL**

```
case CALL:
    addr = nextcode(vm);
    push(vm, vm->fp); // save current frame pointer
    push(vm, vm->pc); // save return address (program counter)
    vm->fp = vm->sp; // set new frame pointer to current stack pointer
    vm->pc = addr; // jump to function address
    break;
```

Save current frame pointer (fp): Before calling the function, the current frame pointer is pushed onto the stack. This allows the VM to return to the previous frame after the function completes.

Save return address (pc): The return address (program counter) is also saved on the stack so the VM knows where to resume after the function returns.

Set new frame pointer (fp): The new frame pointer is set to the current stack pointer (sp). This establishes a new "stack frame" for the function, which starts where the current top of the stack is.

Jump to function: The program counter is set to the function's address, which effectively jumps to the function's instructions.

RET from function

```
case RET:
    rval = pop(vm); // get return value
    vm->sp = vm->fp; // restore stack pointer to frame pointer
    vm->pc = pop(vm); // restore the program counter (return address)
    vm->fp = pop(vm); // restore previous frame pointer
    push(vm, rval); // push return value onto stack
    break;
```

Restore stack pointer: The stack pointer (sp) is restored to the current frame pointer (fp), effectively removing the current stack frame.

Restore program counter: The return address (stored earlier) is popped from the stack, allowing the VM to resume execution at the point where the function was called.

Restore frame pointer: The previous frame pointer is restored, returning the VM to the calling function's stack frame.

Push return value: The function's return value is pushed onto the stack so that the calling function can use it. But first in RET, the return value is at the

top of the stack, so it is has to be temporarily saved.

Frame pointers ..

- establishes a new stack frame for each function call,
- tracks the base of the current stack frame, allowing local variables and arguments to be accessed relative to it,
- allows the VM to return to the correct point in the program and restore the previous stack frame when a function completes, and
- simplifies managing function calls and returns by keeping track of the stack frame, ensuring that local variables and return addresses are correctly managed.

2.3.2 Local storage

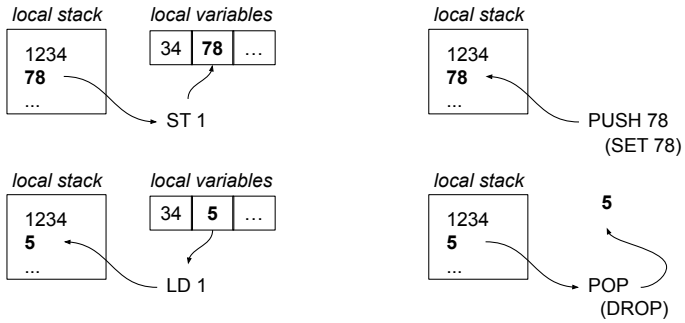
Local storage in local variables in use by LD and ST:

```
case LD:
   offset = nextcode(vm);
   v = vm->locals[vm->fp + (offset * OFF + OFF)];
   push(vm, v);
   break;

case ST:
   v = pop(vm);
   offset = nextcode(vm);
   vm->locals[vm->fp + (offset * OFF + OFF)] = v;
   break;
```

Local variables for the current function are accessed relative to the frame pointer. As there is a common area for the local variables, they are separated by an offset relative to fp. A constant OFF marks how many there can be of local variables in a frame. The frame pointer marks the beginning of the stack frame for the current function. By adding an offset to the frame pointer (OFF), the VM accesses the local variable.

Push and pop operations simulate stack-based behavior typical in function calls or computations. The function 'push' pushes a value onto the frame's local stack by incrementing the stack pointer and placing the value in the stack. And 'pop' pops a value from the stack by returning the value at the top of the stack and then decrementing the stack pointer.



2.3.3 Memory management

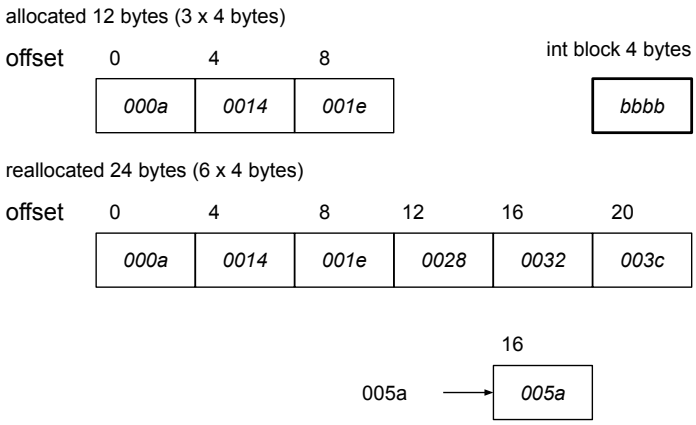
Let's briefly discuss memory usage. Currently, we allocate fixed-size arrays for variables, local storage, and function arguments, all of which use the same size in VM3. This approach is simple to work with, not ideal for practical use. The stack can grow arbitrarily, and code may occupy redundant memory space. Given that memory is often limited, it's crucial for the virtual machine to manage memory efficiently and adaptively.

Physical machines typically have fixed, limited memory resources. However, within these constraints, the virtual machine should manage resources more efficiently. This can involve allocating, reallocating, and deallocating memory in smaller chunks. For instance, dividing memory into blocks is one way to address these constraints effectively. Here we enter the difference between *the stack* and *the heap*.

Stack Memory. This is what we have already learned. This is often where local variables, function parameters, and control flow (e.g. return addresses) are stored. The key characteristic is its Last-In-First-Out (LIFO) structure. Every time a function is called, a new "stack frame" is created. When the function returns, the frame is destroyed. An example from the VM, every time we push or pop values onto the stack, we're using this part of memory. But the stack is limited in size. Stack overflows can occur if too many functions are called recursively or too much data is pushed onto the stack without popping.

Heap Memory. This is where dynamically allocated memory is stored. It's used for variables that need to persist longer than a single function call or for data whose size isn't known in advance. Its key characteristic is that the heap is more flexible, but also more complex to manage. You must manually allocate and free memory when using the heap. An example the VM, when you use 'malloc()' to allocate arrays for the stack, arguments, or variables, you're working with heap memory.

Imagine a memory allocation of blocks:



1. Memory allocation: allocate a block large enough to hold 3 integers. Each integer takes 4 bytes. This acts as our starting "dynamic array."

2. Storing values: store integers at different offsets within the block, simulating array access: 10 (0x0a) at 0 offset, 20 (0x14) at 4 offset, 30 (0x1e) at 8 offset.

3. Reallocation (or expanding): when more space is needed (to store more integers), we reallocate the block to a larger size, 3 more integers: 40 (ox28), 50 (0x32) and 60 (0x3c). This is similar to how a dynamic array might double its size when full.

4. Accessing values: retrieve values from the block using the offset. We can replace at a certain offset e.g. replace 50 with 90 (0x5a) at offset 16.

An example of this approach is demonstrated in the implementation found in MEM, as detailed in the Workbook, see Section 2.4. At present, the virtual machine implementations does not include any advanced memory management features to keep the code simple.

2.3.4 Frame stack

The code illustration VM4 operates on a stack, much like the previous machines we've worked with. However, this VM has two distinct types of stacks. The first is the familiar data stack, which holds integers and allows operations like addition, multiplication, and possibly other arithmetic functions. The second stack, introduced here, is dedicated to managing frames. While we've previously allocated and deallocated memory blocks for data in a more arbitrary manner, this stack helps us structure the process more effectively, specifically by dealing with frames.

Frames in this case are more complex structures (as in C 'struct') that consist of several key components: a smaller stack, a stack pointer, an array

for storing the local variables' data, a slot for the return address, and a slot for the return value.

Virtual machines can often be designed with specific programming languages in mind, and in this case, the focus is on supporting function calls. Functions in programming languages can be understood as mechanisms that closely resemble the allocation and deallocation of memory on a stack.

- 'Frame' represents a single stack frame, which holds its own data stack, local variables, a return address and a return value.
- 'FrameStack' manages multiple frames, representing the call stack. Here the frame pointer tracks the top of the stack.

```
typedef struct {
    int stack[STACK_SIZE]; // data stack for frame
    int locals[LOCALS_SIZE]; // local variables
    int sp; // stack pointer
    int returnValue; // special: return value for frame
    int returnAddress; // special: return address
} Frame;

typedef struct FrameStack {
    Frame* frames[STACK_SIZE]; // stack of frames
    int fp; // frame pointer
} FrameStack;
```

Push and pop frames

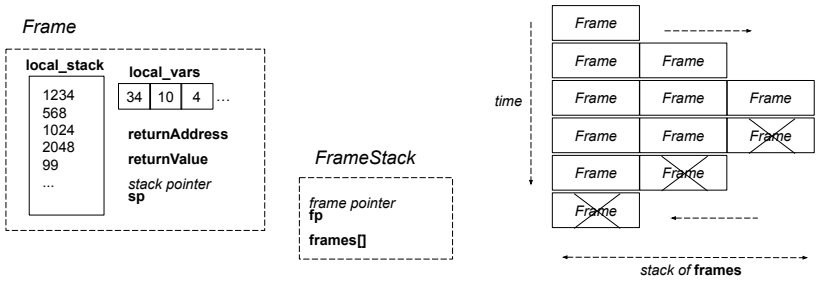
The allocation/deallocation is managed through pushing and popping frames to and from the frame stack or 'call stack'. Some bounds checking ensures that neither the stack nor the frame stack overflows or underflows.

- 'pushFrame' allocates a new stack frame and pushes it onto the frame stack. It sets the internal stack pointer to -1, the return value to 0, and also the return address to 0. I at last returns which frame pointer it is represented by.
- 'popFrame' removes the top frame from the frame stack. Before it frees memory, the program counter is set to the return address (from an assumed function call filled that in). Then it decreases the stack pointer for the frames, and returns the frame pointer number of the now deleted frame.

```
int pushFrame(VM* vm) {
    if (vm->fstack.fp >= STACK_SIZE - 1) {
        error(vm, "Frame stack overflow");
    }
    Frame* frame = (Frame*) malloc(sizeof(Frame));
```

```
    frame->sp = -1;
    frame->returnValue = 0;
    frame->returnAddress = 0;
    vm->fstack.frames[++(vm->fstack.fp)] = frame;
    return vm->fstack.fp;
}

int popFrame(VM* vm) {
    if (vm->fstack.fp < 0) {
        error(vm, "Frame stack underflow");
    }
    Frame* currentFrame = vm->fstack.frames[vm->fstack.fp];
    vm->pc = currentFrame->returnAddress;
    free(currentFrame);
    vm->fstack.fp--;
    return vm->fstack.fp + 1;
}
```



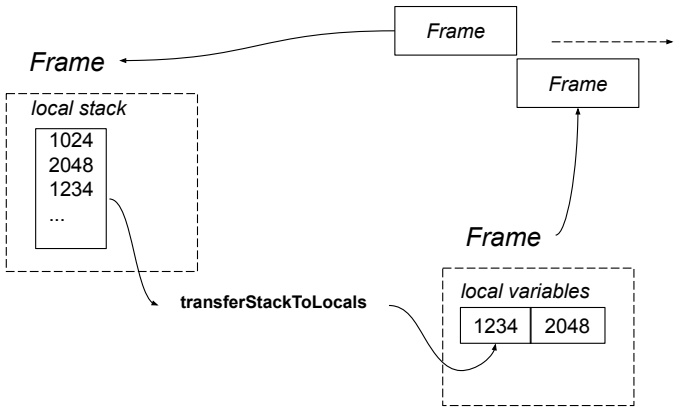
Call with or without arguments

The opcodes CALL and RET abstract *argument passing and value returning*, functioning similarly to how higher-level languages manage function calls.

The *CALL* opcode sets up a new function call. It pushes a new frame onto the frame stack, saves the current program counter as the returnAddress in the new frame, and transfers any specified arguments from the calling frame's stack to the new frame's local variables.

After this setup, CALL updates the program counter to the target address of the called function, effectively "jumping" to the start of the function.

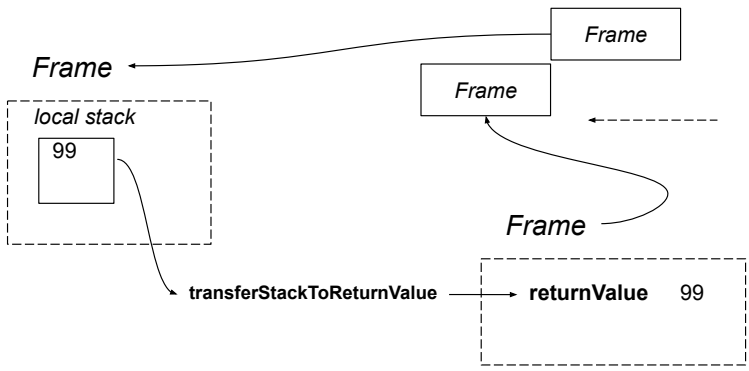
```
case CALL:
    num = next(vm);
    addr = next(vm);
    int frm = pushFrame(vm);
    fr = getFrame(vm, frm);
    fr->returnAddress = vm->pc;
    if (num > 0) {
        transferStackToLocals(vm, num);
    }
    vm->pc = addr;
    break;
```



The *CRET* opcode retrieves the return value of a function call. It pushes the returnValue stored in the current frame onto the stack. This allows the main program or a calling function to use the result of the last RET operation after a function finishes execution.

The *RET* opcode completes a function call. It optionally transfers a computed result to the calling frame's returnValue, pops the current frame from the frame stack, and restores the program counter to the stored returnAddress, allowing the program to resume from where the function was called.

```
case RET:
    if (vm->fstack.fp > 0) {
        transferStackToReturnValue(vm);
    }
    fr = vm->fstack.frames[vm->fstack.fp];
    vm->pc = fr->returnAddress;
    popFrame(vm);
    break;
```



These opcodes support recursive functions by maintaining separate frames

for each function call, allowing return addresses and local data to be preserved independently.

Example: Function simulation

The factorial of a number n can be defined recursively as:

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$$

The function then can represented in our familiar VM.

```
// Main Program
PUSH, 5, // Push the initial value n = 5
CALL, 1, 8, // Call factorial(5)
CRET, // Retrieve the result of factorial(5)
PRINT, // Print the result
HALT, // End of program

// Function factorial (Address 8)
LD, 0, // Load n (the argument)
PUSH, 1, // Push constant 1
SUB, // Subtract to check if n == 1
JZ, 28, // Jump to base case if n == 1

// Recursive Case (if n > 1)
LD, 0, // Load n again (the argument)
LD, 0, // Load n again (for factorial(n - 1))
PUSH, 1, // Push constant 1
SUB, // Calculate n - 1
CALL, 1, 8, // Call factorial(n - 1)
CRET, // Get factorial(n - 1)
MUL, // Calculate n * factorial(n - 1)
RET, // Return the result of n * factorial(n - 1)

// Base Case (Address 28)
PUSH, 1, // Push 1 (factorial of 1 is 1)
RET // Return 1
```

2.3.5 Summary

Memory and functions

VM3 introduces advanced capabilities compared to previous virtual machines (VM1 and VM2), including jump instructions, comparison operators, and memory storage that allow for loops, conditionals, and function calls. VM3 manages function calls using "activation records," which store function arguments, local variables, and return addresses on the stack.

Assembly, machine code, and bytecode

In VM3, the assembler converts assembly code into "bytecode," an intermediate format that is executed by the virtual machine. The bytecode is represented as a series of integers. Unlike machine code, bytecode adds a layer of abstraction that increases portability across hardware platforms. This process simplifies execution and makes the VM more versatile.

Prime numbers example

A sample program demonstrates how VM3 generates prime numbers from 1 to 99. The algorithm counts divisors for each number and prints the number if it has exactly two divisors (i.e., it's a prime). Although this approach works, there are potential Optimisations such as skipping even numbers after 2 or reducing redundant storing and loading operations.

Compilation and execution

The program's assembly code is compiled into bytecode using a makefile process and then executed by the VM3 engine. The prime number program runs efficiently, displaying the prime numbers within the specified range.

Frame pointer

The frame pointer is essential in VM3 for managing function calls. Each function call creates a new stack frame, and the frame pointer keeps track of local variables and arguments relative to the current frame. During a function call, the frame pointer is saved and a new frame is created. When returning from a function, the frame pointer is restored to manage the calling function's stack. The VM3 operates within the bounds of already allocated memory, so in all there is no allocation and deallocation of memory in total.

Memory management

VM3 uses fixed-size arrays for variables, local storage, and function arguments. While simple, this method is inefficient for practical use because the stack can grow arbitrarily and lead to redundant memory usage. An improved memory management strategy would involve more dynamic allocation techniques, such as reallocating memory in smaller, adaptable blocks, as demonstrated in a sample implementation in MEM.

Allocation and deallocation of frames

MEM uses dynamic memory allocation (‘malloc‘) to allocate memory for local variables and stacks at runtime. This reduces memory usage compared to static allocation at compile time. ‘allocFrame‘ and ‘deallocFrame‘ functions manage

memory by allocating space for local variables and freeing it when no longer needed.

Local storage

The code handles local variables in a stack-based virtual machine using LD (load) and ST (store) instructions. Local variables are accessed relative to the frame pointer (fp), and an offset (OFF) is used to separate them within the frame. The push operation places values onto the stack, while pop retrieves them, adjusting the stack pointer accordingly. The "local" concept is logical, meaning the order of variables is managed by offsets, rather than their physical location in memory.

Frame Stack

The VM4 operates with two stacks: a data stack for computations and a frame stack for managing function calls. The frame stack handles more complex structures like local variables, a local stack, return addresses, and return values. Frames are pushed onto or popped from this frame stack as functions are called or completed.

Call with arguments

The CALLV instruction manages function calls by transferring arguments from the caller's stack to the callee's local variables. RETV handles function returns by transferring the return value back to the caller's stack.

2.4 Practice

Workbook: Chapter 2.
https://github.com/Feyerabend/bb/tree/main/workbook/ch02
Scan the QR code to access exercises, code, resources and updates.

In this chapter the focus is on exploring different implementations and techniques related to virtual machines (VMs). The chapter starts by introducing some simple VMs and provides a breakdown of their core components, including the stack and the interpreter technique. It goes on to describe the VM1 implementation and how it works, followed by an explanation of the REGVM implementation, with attention to aspects like portability. It also examines how registers work compared to stack operations in a VM.

The chapter then delves into the VM2 implementation, highlighting topics such as performance comparisons and error handling. Moving further, it introduces VM3 implementation, which incorporates more advanced concepts like the frame pointer, local storage, memory management, and the frame stack. The VM4 goes even further with especially function calls and returns.

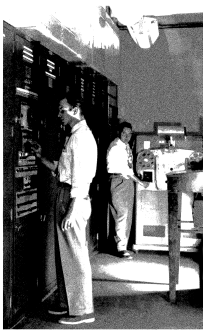
Example: Exploration exercises

1. *What is a virtual machine (VM)?* Describe the concept of a virtual machine and its purpose. Explain how it provides an abstraction layer between the software and hardware, allowing code to run in different environments.

2. *What is the history of VMs, and what has changed?* Discuss e.g. the origins of virtual machines in the 1960s. Describe the evolution of VMs from resource management on mainframes to their current applications in cloud computing and development environments.

3. *What is the difference between stack-based and register-based VMs?* Compare and contrast the architectures of stack-based and register-based virtual machines, highlighting the advantages and disadvantages of each.

Example: Project to build a von Neumann machine



The EDVAC (Electronic Discrete Variable Automatic Computer), completed in the late 1940s, was one of the earliest stored-program computers. It was designed by a team that included John von Neumann, who contributed a groundbreaking concept known as the von Neumann architecture. This architecture proposed that a computer's memory could store both data and instructions, allowing the computer to read and execute instructions directly from memory—a major departure from previous designs where programming was done by manually rewiring hardware.

Build you own VM

Build a simple virtual machine (VM) in your preferred programming language, whether it's Haskell, Rust, Lua, Go, or something else.

If you have a lot of experience on programming in some language other than C or Python but so far not explored virtual machines, consider making a plan for an implementation of a VM of your own. Overall, this exercise builds a small foundation in systems programming, compilers, and computer architecture.

1. Instruction set:

    - Start by defining a minimal set of instructions (e.g. arithmetic operations, memory manipulation, jumps). Keep the instruction set simple to avoid overwhelming complexity.

- Decide whether your VM will support stack-based or register-based instructions. A stack-based VM is easier to implement initially, while a register-based VM can be more efficient.

2. Memory model:

   - Define how your VM will handle memory. Will you implement a stack for function calls, local variables, and intermediate values? Will you use registers or memory addresses?
   - Consider how to handle global variables, heap memory, and memory bounds checking to avoid crashes.

3. Errors:

   - Handle errors gracefully, such as illegal instructions, memory access violations, and stack overflows. Implement clear error messages that help debug your VM's behavior.
   - Include handling for invalid program counter jumps, infinite loops, and incorrect use of instructions (e.g. division by zero).

4. Instruction fetch and execute cycle:

   - Ensure that your VM has a well-structured instruction fetch-decode-execute loop. This should increment the program counter after fetching each instruction and execute the corresponding operation.
   - Plan how you'll decode the instructions and their arguments (e.g. instruction format and operand handling).

5. I/O:

   - Plan for how your VM will interact with the outside world. Will it take input from files or allow console-based interaction?
   - Think about adding instructions for basic I/O operations like printing or reading from standard input.

6. File format:

   - Decide how your programs will be represented in files. You could use a simple text-based format with instructions in plain text, or a binary format that your VM can directly interpret.
   - If you're working with text-based assembly, you'll also need to write an assembler to translate human-readable code into machine code for your VM.

7. Testing and debugging:

- Start with simple programs to test your VM, like basic arithmetic or looping. Debugging VMs can be tricky, so implement logging or a simple debugger to print the current state (e.g. the value of the program counter, stack, registers).
- Write test cases for all instructions to catch bugs early and ensure that your VM behaves as expected.

8. Extensibility:

   - Keep your design modular. As you progress, you might want to add new features like function calls, more complex data types, or additional instructions, so it's important to keep your initial design flexible enough to accommodate growth.

9. Performance:

   - In the beginning, focus on correctness rather than performance. Later, you can explore optimizations like instruction caching or efficient memory access patterns.

Chapter 3

Debugging, Optimisation, and Tests

3.1 Prerequisites

Before diving into this chapter, you will need access to a standard computer (such as a Mac, PC, or Linux machine), along with a text editor, a C compiler, and a Python interpreter. It is assumed that you already have a solid understanding of programming in both C and Python.

3.2 Basic tools

Computers have inherent limitations, such as processing power, memory capacity, and input/output capabilities, which can significantly affect program performance, particularly with large datasets or demanding computations. Consequently, programmers must strive for not only bug-free code but also robust and optimised solutions.

Stability is critical; a stable program behaves predictably under various conditions, which is vital in fields like healthcare and finance. Optimisation plays a key role in enhancing efficiency, reducing resource consumption, and improving user experience. This includes writing efficient algorithms, optimising memory usage, and ensuring a responsive user interface.

Successful programming requires a holistic approach that encompasses thorough testing, debugging, and a deep understanding of the software's context. As technology evolves, programmers must remain adaptable and continuously refine their skills to meet the challenges of creating efficient and effective software solutions.

3.3 Debugging

Let's first explore the three common types of bugs in programming: syntax errors, logical errors, and runtime errors.

Syntax errors

These occur when the code violates the rules or grammar of the programming language. They are caught by the compiler or interpreter before the program runs.

```
#include <stdio.h>

int main() {
    printf("Hello, World!"
    return 0;
}
```

Here, there's a missing closing parenthesis ')' in the 'printf()' statement. When the code is compiled, the compiler will generate an error, indicating that the syntax is incorrect. Syntax errors prevent the program from running at all.

How they manifest:

- The program will not compile or run.
- The compiler/interpreter will provide an error message pointing out the location of the issue.

Logical errors

Logical errors occur when the program runs, but it produces incorrect or unintended results. These errors are caused by flawed logic in the program, and they are the hardest to detect because the code doesn't crash or halt. It simply gives the wrong output.

```
#include <stdio.h>

int main() {
    int a = 10;
    int b = 20;
    int sum = a * b; // Should be a + b instead of a * b
    printf("Sum is: %d\n", sum);
    return 0;
}
```

Here, the programmer intended to calculate the sum of a and b, but instead, the product is calculated due to using the wrong operator (* instead of $+$). This is a logical error because the program runs successfully but produces an incorrect result: "Sum is: 200" instead of "Sum is: 30".

How they manifest:

- The program runs without errors.
- The output or behavior of the program is incorrect or unexpected.
- The compiler cannot catch this kind of bug since the syntax is valid.

Runtime errors

Runtime errors occur while the program is running. These errors are typically caused by invalid operations like dividing by zero, accessing memory that doesn't exist, or using resources improperly. These errors cause the program to crash or terminate unexpectedly.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = 0;
    int result = x / y; // division by zero
    printf("Result: %d\n", result);
    return 0;
}
```

In this case, trying to divide x by y results in a division-by-zero error, which is undefined behavior (in most programming languages). This runtime error will cause the program to crash.

How they manifest:

- The program starts but crashes or halts unexpectedly.
- The error occurs only during execution, often accompanied by an error message (e.g. "Segmentation fault" or "Division by zero").

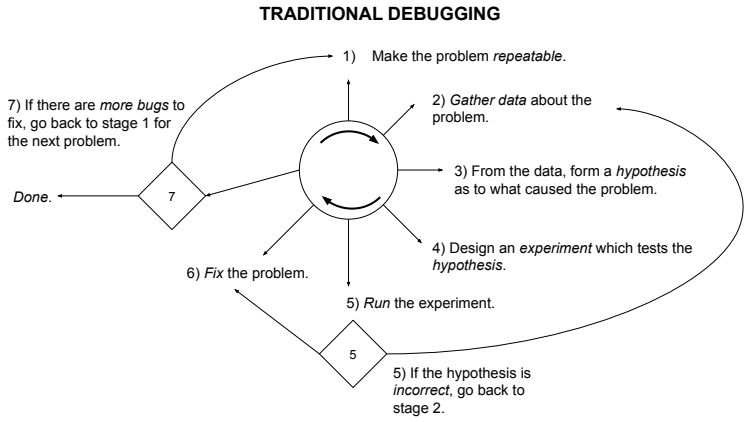
Summary

- Syntax errors are detected by the compiler or interpreter before execution, preventing the program from running.
- Logical errors occur when the program runs but produces incorrect results due to faulty logic.
- Runtime errors occur during program execution, causing the program to crash or behave unexpectedly.

Understanding the differences between these types of bugs helps in identifying and fixing errors more effectively during the development process.

3.3.1 Process

Debugging is the process of identifying and resolving defects or issues in a program. Certain bugs are easily identified and resolved, but some can be persistent and very hard to spot.



The stages of non-trivial debugging are:

1. Ensure the problem can be *consistently reproduced*. This typically involves identifying which specific inputs trigger the issue.
2. Collect *detailed data* about the problem. This step is essential, so approach it with patience and attention to detail.
3. Based on the gathered data, *develop a hypothesis* about the underlying cause of the problem. This educated guess should be informed by the patterns or clues found in the data.
4. Create an experiment that directly *tests the hypothesis*. The key is to structure the experiment so that it yields a clear yes or no result, confirming or disproving the hypothesis.
5. *Run the experiment*. If the hypothesis is incorrect, go back to stage 2.
6. Resolve the problem by *applying the appropriate solution* based on the outcome of the experiment. Ensure that the fix addresses the root cause and thoroughly test it to verify the issue is fully resolved.
7. If there are *additional bugs* to resolve, return to stage 1 and repeat the process for each new problem. This ensures a systematic and thorough approach to debugging.

To effectively tackle a bug or issue, the first step is to make the problem repeatable. This typically requires identifying the specific inputs or conditions

that consistently lead to the issue, ensuring that the problem can be triggered reliably. Once this is achieved, it's essential to gather as much data as possible about the problem. Carefully observe any error messages, crash dumps, or logs, as these often contain valuable clues that should not be overlooked. This stage requires patience and attention to detail, as premature conclusions can lead to missed insights.

With the data in hand, the next step is to form a hypothesis about the root cause of the problem. Based on the information gathered, you can then design an experiment to test the hypothesis. The key here is to ensure that the experiment produces a clear, binary result—either the hypothesis is supported or it is disproven. After running the experiment, if the hypothesis turns out to be incorrect, it's crucial to return to the data-gathering phase and refine your understanding before trying again.

Once the correct hypothesis has been confirmed, the issue can be fixed. If multiple bugs remain, the process starts anew, following the same structured approach. This methodical, iterative process not only helps solve the current problem but also improves debugging skills by reinforcing a disciplined, logical approach to problem-solving.

3.3.2 Tools

We present a selection of tools that, while simple and perhaps somewhat basic, serve as useful starting points. Although there are many tools worth exploring for debugging and related tasks, we will focus on just a few key examples.

The following highlights open-source tools commonly used in Linux, such as GDB and GCC, which are also compatible with Microsoft Windows and macOS. Additionally, commercial equivalents of these tools are available for both Windows and macOS platforms.

Print statements

One of the simplest and earliest debugging techniques is inserting print statements into code to monitor its execution at runtime. This method provides visibility into variable values, program flow, and state at various points in the code.

Developers insert lines like 'print()' in Python or 'printf()' in C at critical places in the code to output values to a console or log file. For example, you might print the state of a loop counter, input values, or function return values.

```
int main() {
    for (int i = 0; i < 10; i++) {
        printf("Iteration %d\n", i);
    }
    return 0;
}
```

- Pros:

  - Extremely simple to implement.
  - No special tools or setup is required.
  - Great for quick checks on values and flow.

- Cons:
  - Invasive as it requires changing the code itself, and too many print statements can clutter the output.
  - Doesn't work well for timing bugs or complex interactions, especially in real-time systems.

Logging

Logging is a more structured version of print debugging where developers log critical events, data, and state to a file for post-execution analysis. Historically, this was done with simple file output, but logging libraries have since become common to handle this in a more organised way.

Instead of using print statements, you can use logging frameworks to log messages, variable values, and error states to a file or output stream, often with different severity levels (e.g. INFO, DEBUG, ERROR).

```
import logging

# set up logging configuration
logging.basicConfig(level=logging.DEBUG,
                format='%(asctime)s - %(levelname)s - %(message)s')

def example_function():
    logging.debug('This is a debug message.')
    logging.info('This is an info message.')
    logging.warning('This is a warning message.')
    logging.error('This is an error message.')
    logging.critical('This is a critical message.')

if __name__ == '__main__':
    example_function()
```

- Pros:
  - Non-intrusive as logs can be left in the code and turned on/off as needed.
  - Works well in production environments where real-time debugging isn't possible.
  - Logs provide a timeline of program execution, helping diagnose complex, asynchronous, or time-sensitive issues.

- Cons:

  - Can introduce performance overhead if logging is too verbose or frequent.
  - Requires careful planning of what to log to avoid overwhelming the developer with data.

LEDs and serial ports

In the early days of embedded systems (and still common today), developers would use serial output or LEDs to debug their hardware programs. When running on hardware, especially microcontrollers with limited output capability, toggling an LED or sending data over a serial port provided a way to communicate program state.

Example was to blink an LED if a certain part of the program is reached, or use a serial port to send debug information (e.g. UART communication, see Section 4.5.1).

```
gpio_set_function(25, GPIO_OUT); // Raspberry Pi Pico example
gpio_put(25, 1); // turn LED on when certain conditions are met
```

- Pros:
  - Non-intrusive to program logic; you can toggle LEDs or send serial data without altering the main code structure.
  - Works well on hardware with no display or debugging interfaces.
- Cons:
  - Limited in terms of the amount of information conveyed (e.g. toggling an LED is binary).
  - Less useful in many cases for debugging complex software states or logic errors.

Breakpoints and single stepping

Breakpoints and single stepping are classic debugging techniques used with early debuggers, such as the 'GNU Debugger' (GDB), 'dbx' on Unix, or hardware debuggers (see a separate part below on 'Hardware debugging'). Breakpoints allow you to pause program execution at specific points, while single stepping lets you execute one instruction at a time.

1. A breakpoint is set at a specific line or address in the program, causing the program to halt execution when that point is reached.
2. Once the program is halted, you can inspect the state of variables, memory, and CPU registers.

3. Single stepping then allows you to execute one instruction at a time, monitoring how the state of the system changes.

Breakpoint example. After starting GDB with your compiled program, you can set a breakpoint at the main function. This halts the program's execution at the very beginning, allowing you to inspect the environment before any code has been executed.

```
..
(gdb) break main
(gdb) run
```

Once the breakpoint is hit, you have a wide array of commands available to examine the program's state. For example:

- `info registers`: Shows the current values in the CPU's registers, which is useful for low-level debugging.
- `info locals`: Displays the current values of local variables in the scope of the function where the program is halted.
- `print var`: Allows you to print the value of any variable in scope by using its name (replace var with the actual variable name).
- `step` and `next`: Step into a function or advance to the next line of code, respectively.

These tools allow you to navigate through your code, check variable values, and monitor the program's flow.

- Pros:
  - Provides full control over the execution flow.
  - You can inspect the entire program state at the point of the breakpoint.
  - Great for tracking down logical errors and understanding program behavior step by step.
- Cons:
  - Requires knowledge of how to use the debugger effectively.
  - Sometimes difficult in multithreaded or asynchronous environments where many things happen simultaneously.

Core dumps

A core dump[1] is a file that captures the memory state of a program when it crashes. Core dumps have been used historically (and still are) to analyze what went wrong after a crash. The dumped memory includes the stack, heap, and CPU registers at the time of failure.

When a program crashes (due to segmentation faults, illegal instructions, etc.), the operating system can generate a core dump. This can then be loaded into a debugger like GDB to inspect the state of the program at the point of failure.

```
> gdb ./program core
```

- Pros:
  - Allows for post-mortem debugging: You can analyze the crash long after it happened.
  - Useful for hard-to-reproduce bugs (e.g. random crashes or memory corruption).
- Cons:
  - Core dumps can be large and difficult to interpret without a detailed understanding of the system's memory and state at the time of the crash.
  - Requires a debugger capable of loading and analyzing core files.

Memory and hex dumps

Memory dumps and hex dumps provide raw output of a program's memory at a given time, and developers can manually analyze these dumps to inspect variables, buffers, and structures. A hex dump displays memory contents in hexadecimal format, sometimes alongside ASCII interpretations.

Programs like 'hexdump', 'od' (octal dump), or 'xxd' are used to output memory contents. On systems with no advanced debugging tools, developers would often dump memory into a 'file.bin' and manually inspect it for corruption or unexpected values.

```
> hexdump -C file.bin
```

- Pros:

[1] The term core dump originated from the era of core memory. A core dump refers to a snapshot of the contents of memory at a specific point in time, often when a program crashes or encounters a critical error. Initially, this was literally a "dump" of the contents of core memory to an external file or output device. The purpose was to capture the entire state of a running program—its variables, the contents of registers, and the memory it was using—so that developers could analyze what went wrong. Core memory was an early type of memory using magnetic rings to store bits, in early computers.

  - Provides a very granular view of memory, useful for debugging low-level memory corruption or buffer overflows.
  - Works on systems without access to a sophisticated debugger.

- Cons:
  - Manually analyzing memory in hexadecimal is time-consuming and error-prone.
  - Requires a strong understanding of memory layouts, offsets, and program internals.

Symbolic debugging

Symbolic debugging involves using debuggers that can map binary code back to high-level code constructs (like variable names, function names, etc.) using symbol tables. This became common with tools like 'GDB', 'dbx', and others.

When a program is compiled with debugging information (e.g. using the -g flag in GCC), the resulting binary contains *symbol information* that maps variables and functions to their addresses in the binary. A symbolic debugger can use this information to display variable values, function calls, and lines of code during debugging.

```
> gcc -g -o program program.c
```

- Pros:
  - Makes debugging much easier because the debugger can display high-level information rather than raw memory addresses or assembly.
  - Critical for debugging large, complex programs.

- Cons:
  - Larger binary size due to the inclusion of debug symbols.
  - Requires the code to be compiled with debug symbols, which may not always be possible (e.g. in third-party libraries).

Hardware debugging

Hardware debugging involves using specialized tools that interface directly with the microcontroller to inspect its internal state and control execution at the hardware level. For Raspberry Pi Pico and other microcontrollers, tools like the *Pico Debug Probe* and SWD (Serial Wire Debug) interface allow developers to halt execution, set breakpoints, and step through code in real-time, providing deep insight into the behavior of embedded systems.

With the Pico Debug Probe, developers can monitor registers, memory, and peripherals during program execution. This hardware-based approach offers a

higher level of control compared to software-only methods, making it invaluable when working with real-time systems or when precise timing is critical.

- Pros:
  - Direct access to hardware state, allowing for more precise and low-level debugging.
  - Essential for debugging timing issues, peripheral interaction, or hardware faults.
- Cons:
  - Requires additional hardware, such as a debug probe or SWD adapter. However, a second Pico can also be used as an alternative, functioning as a debug probe.
  - Can be more complex to set up compared to software-only debugging.

We will explore the Raspberry Pi Pico in the following Chapter 4, but for our purposes, we will rely on a UART connection instead of using the hardware debug option. This approach will suffice for the simple examples and educational focus of this material. While software-based debugging, like print statements or UART logging, provides basic insights into the program's flow, hardware tools such as the Pico Debug Probe offer more detailed control over elements like timing, performance, and real-time execution. This hardware-based option becomes especially valuable when working on larger, more complex Pico projects.

3.3.3 Summary

This section covers essential prerequisites, basic debugging techniques, and tools for programming on standard computers or microcontrollers. Readers are expected to have experience with C and Python. Debugging is broken down into syntax errors, logical errors, and runtime errors. Effective debugging includes systematically identifying reproducible issues, collecting data, forming and testing hypotheses, and applying solutions. Tools for debugging range from simple print statements and logging to more advanced methods like breakpoints, core dumps, memory dumps, and symbolic debugging, each offering unique advantages for different scenarios. Hardware debugging is also highlighted for its precision, especially in real-time systems, with tools like the Pico Debug Probe offering deep insights into hardware states valuable in embedded systems.

3.4 Optimisation

Optimising programs refers to the process of improving a program's performance in terms of *speed*, *memory usage*, or other resources like *disk* and *network I/O*.

The goal is to make the program more efficient without changing its output or behavior. Program optimisation is essential in many contexts, from embedded systems where resources are limited, to high-performance computing where even small inefficiencies can be magnified.

Typical process

1. *Profiling.* Start by profiling your program to identify performance bottlenecks or hotspots.

2. *Targeting bottlenecks.* Focus your optimisation efforts on areas that contribute the most to performance degradation (e.g. slow loops, excessive memory usage, or I/O delays).

3. *Algorithmic improvements.* Once you've identified bottlenecks, explore whether a more efficient algorithm can replace the existing one.

4. *Fine-tuning with compiler optimisations.* Use compiler flags and optimisations to improve low-level performance. Profile-guided optimisations (PGO) can further enhance this.

5. *Memory and I/O optimisation.* After computational optimisations, focus on optimising memory access patterns and I/O operations, which may still be limiting performance.

Optimising programs is a multi-step process that involves analyzing the performance of a program, identifying bottlenecks, and applying techniques ranging from compiler flags to algorithmic changes and parallelism. The right tools and techniques vary depending on the type of program and its use case. Profilers, memory analyzers, and parallelization tools are essential to this process, along with a deep understanding of the underlying hardware and algorithms.

Types

- *Compiler.* These are optimisations that happen during the compilation of source code to machine code. Compilers often have multiple levels of optimisation flags (like -O1, -O2, -O3 in GCC or Clang) that can help improve performance. Examples include:

  - Inlining: Replacing a function call with the body of the function to reduce overhead.
  - Loop unrolling: Expanding loops to reduce the number of iterations.
  - Dead code elimination: Removing code that is never executed.
  - Constant folding: Computing constant expressions at compile time instead of runtime.

- *Algorithmic.* This involves selecting more efficient algorithms for a given task. Often, significant improvements can be achieved by changing the algorithm, for example:
  - Replacing a bubble sort with quick sort or merge sort.
  - Using more efficient data structures like hash maps instead of lists for certain lookup operations.
- *Memory.* These optimisations reduce the memory footprint of an application, which can also lead to performance improvements. Examples include:
  - Memory pooling: Reducing the number of expensive memory allocation and deallocation operations.
  - Cache optimisation: Rearranging data structures to maximize cache locality, which reduces cache misses.
  - Compression: Using techniques like data compression to reduce memory usage at the expense of additional CPU overhead.
- *Parallelism and concurrency.* These optimisations involve taking advantage of multiple CPU cores, hardware threads, or distributed systems. Examples include:
  - Multithreading: Splitting a program into multiple threads to run on different cores.
  - Vectorization: Leveraging SIMD (Single Instruction, Multiple Data) instructions to perform the same operation on multiple data points simultaneously.
  - GPU acceleration: Using Graphics Processing Units (GPUs) for tasks that can be parallelised, such as deep learning or graphics rendering.
- *I/O.* Many programs are bottlenecked by input/output operations, especially disk or network access. Optimising I/O can have a significant impact on performance:
  - Buffering: Using buffers to reduce the number of read/write operations.
  - Asynchronous I/O: Allowing the program to continue executing while waiting for I/O operations to complete.
  - Caching: Storing frequently accessed data in memory to avoid costly disk or network operations.

Tools

We will focus on some basic optimisation techniques below. Each tool you implement should serve a distinct purpose in enhancing the overall performance of a program or application. This will help you grasp how different optimisation methods target specific aspects of program efficiency.

1. *Profilers.* Profilers are tools that help developers understand the performance characteristics of their programs, pinpointing bottlenecks and inefficient code. Some well-known profilers include:
    - gprof: A profiling tool for Unix-based systems.
    - perf: A performance analysis tool available on Linux that provides various metrics like CPU cycles and cache misses.
    - Valgrind: A tool that detects memory issues, including leaks and invalid memory accesses.
    - Visual Studio Profiler: For .NET and Windows applications, providing detailed performance metrics.
2. *Static analyzers.* These tools analyze source code without running it, looking for potential optimisations and performance issues:
    - Clang Static Analyzer: Provides warnings about inefficient code and potential bugs.
    - Coverity Scan: Another static analysis tool used for finding bugs and performance issues.
3. *Compiler tools.* Modern compilers come with optimisation capabilities built-in. For example:
    - GCC/Clang: Offers various levels of optimisations (-O1, -O2, -O3, -Ofast) and flags for specific types of optimisations.
    - LLVM: A framework for building custom optimizers, often used for more advanced or domain-specific optimisations.
    - Intel's ICC: A compiler optimized for Intel hardware, providing advanced optimisations for vectorization, parallelism, and CPU features.
4. *Memory optimisers.* These tools help analyze and reduce the memory usage of a program:
    - Massif: A heap profiler from Valgrind that helps understand memory consumption.
    - dmalloc: A debug malloc library to check memory leaks and errors.

Excluded from the above are e.g. parallelization and I/O tools. Example of the former is the well know CUDA, and of the latter IOzone.

Example: Profiling in steps

Profiling a C program involves measuring its performance to identify areas that may benefit from optimization, such as slow-running functions or inefficient code paths. Here's a outline of how to profile a typical C program using common tools like 'gprof', which is a widely used profiler for Unix-like systems.

1. *Compile the program with profiling support.* First, you need to compile your C program with profiling enabled. This is typically done by passing the -pg flag to gcc, which instructs the compiler to include additional instrumentation in the program for profiling.

   ```
   > gcc -pg -o program program.c
   ```

   This compiles the 'program.c' file and produces an executable named 'program', with profiling support included.

2. *Run the program.* Next, you run the compiled program as usual. While the program runs, it collects profiling data about the function calls, how often each function is called, and the time spent in each function.

   ```
   > ./program
   ```

   After the program finishes executing, it generates a file called 'gmon.out' in the working directory, which contains the collected profiling data.

3. *Analyze the data.* To interpret the profiling data, you use 'gprof', which reads the 'gmon.out' file and generates a report. You can view this report by running (n.b. the first '>' is here an indicator of the prompt, only the second you enter):

   ```
   > gprof program gmon.out > profile.txt
   ```

   This command tells 'gprof' to analyze the executable 'program' and its associated 'gmon.out' file, and to output the results to a file called 'profile.txt'. You can open this file to review the profiling report.

4. *Interpret the report.* The profiling report generated by 'gprof' will include two main sections:

   - *Flat profile.* This shows how much time was spent in each function, both in terms of absolute time and as a percentage of the program's total runtime.
   - *Call graph.* This section provides detailed information about which functions are called which, how many times they were called, and how much time was spent in each function.

Using this information, you can identify performance bottlenecks, such as functions that are called frequently or take up a large proportion of the program's execution time. You can then optimise those functions to improve performance.

Example of a 'flat profile':

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 30.0      0.03     0.03      2000     0.01     0.02  functionA
 20.0      0.05     0.02      1500     0.01     0.03  functionB
 50.0      0.10     0.05      1000     0.05     0.05  functionC
```

5. *Optimise.* Next you follow the procedure from above. After identifying the bottlenecks, you can revise the code to optimise the problematic sections, recompile, and repeat the profiling process to assess the impact of the changes.

3.4.1 Memory

Memory optimisations aim to reduce the memory footprint of a program, which not only conserves system resources but also improves speed, particularly in memory-bound applications. Here are some common techniques:

1. Reducing memory usage

   - *Efficient data structures.* Selecting the right data structure is critical for memory optimisation. For example, replacing an array with a more compact structure like a bit vector can save significant memory when storing binary data. Using linked lists can be space-efficient when the dataset size is unpredictable, but contiguous arrays may be better when the size is fixed and known.
   - *Smaller data types.* Avoid using larger data types where smaller ones suffice. For instance, if an integer only needs to store values between 0 and 255, use an 8-bit uint8_t instead of a 32-bit int.
   - *Object pooling.* If your program frequently creates and destroys objects, you can use object pools to reuse objects rather than constantly allocating and deallocating memory. This minimizes the overhead of memory allocation and reduces fragmentation.
   - *Lazy initialization.* Postpone the creation of objects or data until they are absolutely needed. By not initializing everything upfront, you can reduce the initial memory overhead.

2. Improving memory access patterns

- *Cache locality.* CPUs rely heavily on caches (small, fast memory stores) to reduce the latency of memory access. Programs that access data in a predictable pattern (e.g. accessing elements in an array sequentially) tend to benefit from better cache locality. Structuring your data to allow for more cache-friendly accesses can significantly speed up execution.
- *Temporal locality.* Reusing the same data within a short period.
- *Spatial locality.* Accessing contiguous memory locations (as in array traversal).
- *Avoiding memory fragmentation.* In long-running programs, continuous allocation and deallocation can lead to memory fragmentation, where free memory is split into small, non-contiguous chunks. This can be mitigated by:
  - Pre-allocating memory in chunks if possible.
  - Using memory allocators that minimize fragmentation (e.g. arena allocators or slab allocators).

3. Garbage collection tuning (in managed languages)

   For languages with automatic memory management (like Java, C#, or Python), optimizing the garbage collector (GC) can have a significant impact on both memory and speed.

   - *GC tuning.* Tuning garbage collection involves adjusting parameters such as heap size, GC intervals, and pause times. For example, a larger heap might reduce GC frequency but increase memory consumption, whereas a smaller heap results in more frequent GC cycles but conserves memory.
   - *Minimizing object creation.* In languages with garbage collection, reducing the creation of short-lived objects can reduce the load on the GC and prevent memory leaks.

4. Memory pools

   Using memory pools (also called memory arenas) for allocating memory in bulk can reduce fragmentation. This is especially useful in embedded systems or applications with constrained memory. Memory pools allocate large blocks of memory and hand out chunks from this pre-allocated pool, speeding up allocation/deallocation since these operations don't involve the system's general-purpose allocator.

3.4.2 Time

Speed optimisations aim to reduce the time a program takes to execute by making it more computationally efficient or by reducing bottlenecks in processing.

1. Algorithmic optimisation

   Time complexity. One of the most significant factors in speed optimisation is choosing algorithms with better time complexity. For example, an $O(n \log n)$ sorting algorithm (like quicksort in best case) is significantly faster than $O(n^2)$ algorithms (like bubble sort) for large datasets.[2]

   Avoiding redundant computations. Reusing previously computed results instead of recalculating them can significantly improve performance. Techniques like memoization (storing the results of expensive function calls) or tabulation (building up a table iteratively) help reduce redundant operations, particularly in recursive algorithms. Additionally, dynamic programming is a powerful approach to optimize such problems by solving subproblems once and reusing their results.

2. Loop optimisations

   Loop unrolling. Manually or automatically (via compiler optimisations) expanding loops so that multiple iterations are executed per cycle. This reduces the loop control overhead (like incrementing a counter or checking the loop condition). (Compare with the possibilities for unrolling in the Fibonacci series example, as discussed in Section 2.2.)

   Reducing loop nesting. Minimizing nested loops can have a dramatic effect on performance since the complexity increases with the depth of nesting. Where possible, flattening loops or breaking complex nested operations into simpler forms can improve speed.

   Minimizing expensive operations in loops. Move calculations or conditions outside the loop if they don't need to be recalculated each iteration. For instance, calculating something like 'array.length' repeatedly in a loop is inefficient if the length doesn't change. (However this might also depend on e.g. compilers that are smart enough to optimise this.)

3. Concurrency and parallelism

   Multithreading Taking advantage of multiple CPU cores by running different parts of the program in parallel can lead to massive performance improvements. Libraries like OpenMP or pthreads in C/C++ or thread libraries in Python and Java allow you to parallelize tasks.

   Asynchronous programming. Non-blocking I/O operations allow a program to perform useful work while waiting for I/O tasks to complete. Asynchronous frameworks (e.g. Python's 'asyncio') or event-driven models (e.g. 'Node.js') allow for concurrent I/O, significantly improving responsiveness in I/O-bound applications.

[2] Big O notation is a way in computer science to describe how an algorithm's time or space needs grow as the input size gets larger. It helps compare algorithms by focusing on their efficiency, especially in the worst case, without worrying about small details that don't affect overall growth.

4. Minimizing I/O bottlenecks

   Batching I/O. Instead of handling I/O operations individually (which can be expensive), batch multiple operations together. For example, writing data to a file in larger chunks rather than frequently writing small amounts can improve performance.

   Disk and network caching. Caching frequently used data in memory reduces the number of slow disk or network accesses.

Combining memory and speed optimisations

Memory-speed trade-offs. Often, memory and speed optimisations require a balance. For example, using larger data structures (e.g. precomputed lookup tables) might increase memory usage but reduce computation time, leading to faster execution. Conversely, using smaller, more compact data structures may conserve memory but result in slower processing due to increased computation or less cache efficiency.

Data locality. Good memory management often enhances speed by reducing cache misses. Rearranging data structures to ensure that frequently accessed data is close together in memory can lead to significant speedups by reducing memory latency.

3.4.3 Confusion matrix

Machine learning, a subfield of artificial intelligence (AI), focuses on creating algorithms and models that learn patterns from data to make predictions or decisions without explicit programming for specific tasks. For example, instead of writing hard-coded rules to classify emails as "spam" or "not spam," a machine learning algorithm identifies patterns in labeled examples and generalizes them to unseen emails.

Teaching machine learning early equips you with tools to analyze complex systems, make data-driven decisions, and understand modern technologies. Beyond building models, a critical skill in machine learning is the ability to evaluate and refine them. This brings us to the concept of debugging in machine learning, which is inherently different from traditional software debugging.

In traditional software development, debugging typically involves identifying and fixing specific coding errors or logic flaws—answering questions like "how did this error occur?" Debugging machine learning models, however, focuses on understanding why a model's performance is suboptimal. This might include questions like:

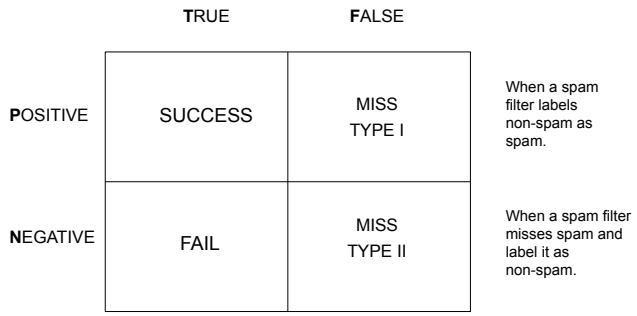
- Why is the model misclassifying certain data points?
- Why does it underperform on specific subsets of the data?

These questions arise because machine learning models derive their behavior from complex interactions among data patterns, model architectures, and

features. Debugging in this context involves examining data distributions, interpreting model decisions, and identifying root causes of errors. This makes machine learning debugging as much about insight and understanding as it is about resolving specific bugs.

A confusion matrix is an essential diagnostic tool in machine learning that provides a detailed breakdown of a classification model's predictions. For binary classification, it is structured as follows:

- **True Positive (TP)**: The model correctly predicts the positive class (e.g. a spam filter correctly identifies spam).
- **True Negative (TN)**: The model correctly predicts the negative class (e.g. a spam filter correctly identifies non-spam).
- **False Positive (FP)**: The model incorrectly predicts the positive class (e.g. a spam filter labels non-spam as spam). This is also called a *Type I error*.
- **False Negative (FN)**: The model incorrectly predicts the negative class (e.g. a spam filter misses spam and labels it as non-spam). This is a *Type II error*.



A *confusion matrix* is commonly used in machine learning to summarize these values for an entire dataset. From this, various performance metrics can be derived, such as precision, recall, and F1-score. These metrics allow developers to fine-tune and troubleshoot model accuracy and the balance between sensitivity (true positive rate) and specificity (true negative rate).

Model debugging

In debugging and optimisation, a confusion matrix helps identify weaknesses in a model. For example, if the false positive rate is high, the model might be "over-calling" the positive class, which could be mitigated by adjusting the decision threshold. Alternatively, if false negatives are high, the model might

be missing important cases, prompting a closer look at data features or model complexity.

The confusion matrix is therefore critical for diagnosing and improving model reliability and for understanding the types of errors the model is most likely to make, both of which are essential for fine-tuning algorithms in real-world applications.

The confusion matrix is a valuable debugging tool because it reveals the types of errors a model is making, which can guide developers in refining and adjusting the model. When debugging a model, analyzing each component of the confusion matrix—true positives, true negatives, false positives, and false negatives—can highlight where the model is strong and where it needs improvement. Here's how this works in practice:

High False Positives (FP). When the model has a high rate of false positives, it means it frequently predicts the positive class when it shouldn't. For example, in a spam filter, high false positives mean non-spam emails are being marked as spam. This issue could frustrate users, as important emails may be missed. To debug this, developers might:

- Adjust the decision threshold for the positive class, making it more conservative.
- Re-evaluate feature importance to check if certain features are overly influencing the positive predictions.
- Inspect data balance to see if the training set has more positive cases, leading the model to over-predict this class.
- Consider additional features or fine-tune existing ones that could help differentiate between positive and negative classes.

High False Negatives (FN). A high rate of false negatives indicates that the model often fails to detect the positive class. In a medical diagnosis model, for instance, high false negatives mean the model is missing cases of disease, which could have severe consequences. To debug this, developers might:

- Lower the decision threshold for classifying a positive case, making the model more sensitive.
- Explore feature engineering to see if additional informative features could help identify positives more accurately.
- Analyze data quality to ensure important features are well-represented in the training data.
- Use data augmentation techniques if the positive class is underrepresented, which could help the model learn to identify these cases better.

Examining True Positives (TP) and True Negatives (TN). While true positives and true negatives indicate correct predictions, it can be useful to review these instances as well. For example:

- Review True Positives: Checking the features in true positives can help confirm whether the model is using relevant information correctly, giving insight into whether similar features could improve predictions for borderline cases.
- Review True Negatives: Similarly, understanding what makes a true negative may help refine the model to avoid false positives.

Using the confusion matrix to guide model tuning. Debugging often involves iterating over different aspects of the model, and the confusion matrix helps focus these efforts. For instance, depending on whether the model's goal is higher precision (fewer false positives) or higher recall (fewer false negatives), developers can use the confusion matrix to tune hyperparameters or decision thresholds accordingly.

Incorporating feedback loops. As part of a debugging and improvement cycle, the confusion matrix is often examined over multiple stages of training and deployment. In real-world applications, user feedback (e.g. users marking emails as "not spam" when falsely classified) can be fed back into the training process. The model can then be updated to reduce the specific error rate, such as lowering false positives in a spam filter.

The confusion matrix is like a 'map of model errors,' allowing developers to "zoom in" on specific issues in performance. By systematically analyzing each quadrant, developers can identify not only where the model is failing but also use these insights to make targeted adjustments to the training process, feature set, or decision-making strategy of the model. This makes the confusion matrix an essential tool for debugging and iterative improvement in model development.

Example: Temperatures

Assume we have time series data of temperature measurements along with corresponding temperature predictions for each time point. By correlating these actual and predicted temperature pairs, we can evaluate and refine our prediction model to improve its accuracy. Using a confusion matrix, we can systematically identify patterns in the model's errors, allowing us to debug and adjust the model for better alignment with the actual temperature trends. Let's illustrate this with an example in Python.

1. Define accuracy thresholds: Since temperatures are continuous values, we need a threshold to classify each prediction as a "hit" or "miss." For example, you could consider a prediction "accurate" if it's within $+/-2$ degrees of the actual temperature. This is naturally an assumption that can be changed.

2. Classify each prediction: Loop through each actual and predicted temperature pair in the log and classify them based on the defined threshold(s).

3. Count hits and misses: Use these classifications to populate a confusion matrix. For simplicity, we can start with a binary confusion matrix with categories like "accurate" (True Positive and True Negative) and "inaccurate" (False Positive and False Negative).

```
# sample: each element is (actual temperature, predicted temperature)
temperature_log = [
    (20, 21), (25, 24), (30, 28), (15, 18), (22, 19), (27, 29),
    (18, 17), (21, 23), (16, 15), (20, 20), (24, 26), (19, 18),
    (23, 24), (28, 27), (19, 20), (17, 18), (22, 23), (26, 25),
    (29, 31), (21, 22), (18, 16), (24, 22), (20, 21), (25, 27)
]

# def. threshold for prediction as "accurate"
threshold = 2 # plus or minus 2 degrees range

true_positive = 0 # accurate positive predictions
false_positive = 0 # predicted higher, inaccurate
true_negative = 0 # accurate negative predictions
false_negative = 0 # predicted lower, inaccurate

# populate confusion matrix
for actual, predicted in temperature_log:
    difference = abs(actual - predicted)

    if difference <= threshold:
        # within acceptable range
        if predicted >= actual:
            true_positive += 1 # slightly higher or equal
        else:
            true_negative += 1 # slightly lower
    else:
        # outside acceptable range
        if predicted > actual:
            false_positive += 1 # overestimated and inaccurate
        else:
            false_negative += 1 # underestimated and inaccurate

print("Confusion matrix")
print(f"True Positives (TP): {true_positive}")
print(f"False Positives (FP): {false_positive}")
print(f"True Negatives (TN): {true_negative}")
print(f"False Negatives (FN): {false_negative}")

# optional: calculate metrics
accuracy = (true_positive + true_negative) / len(temperature_log)
precision = true_positive / (true_positive + false_positive) if (
     true_positive + false_positive) > 0 else 0
recall = true_positive / (true_positive + false_negative) if (true_positive +
      false_negative) > 0 else 0

print("\nPerformance metrics")
```

```
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

We define an error threshold (e.g., +/-2 degrees) for what counts as an accurate prediction. This threshold can be adjusted based on the acceptable margin of error. For each pair in the log, the program calculates the absolute difference between the actual and predicted temperatures. If the difference is within the threshold, the prediction is counted as "accurate"; otherwise, it's "inaccurate." We then assign these to the true/false positive/negative categories based on whether the prediction was above or below the actual temperature.

Metrics

Recall maybe the lesser known metric. In the context of a confusion matrix, recall is a metric that measures a model's ability to identify all relevant instances of the positive class. It is also known as the true positive rate or sensitivity. Recall tells us the proportion of actual positive cases that the model correctly identifies as positive. It is calculated as the ratio of true positives (TP) to the sum of true positives and false negatives (FN).

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

High recall indicates that the model successfully captures most of the positive instances, which is crucial in applications like disease screening, where missing positives can be costly.

Precision measures the proportion of predicted positive cases that are actually correct. It is the ratio of true positives to the sum of true positives and false positives (FP).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

High precision indicates that when the model predicts a positive, it is likely to be correct. Precision is essential in applications where false positives are undesirable, such as in spam filtering.

Accuracy represents the overall correctness of the model, measuring the proportion of all correct predictions (true positives and true negatives) out of the total predictions made.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Accuracy is a general measure of model performance, but can be misleading if the classes are imbalanced.

The unimplemented F1 score is a metric that combines precision and recall into a single value, offering a balance between the two.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.4.4 Summary

This section explores optimization and debugging techniques in programming, with a focus on enhancing performance, memory usage, and algorithm efficiency. Key strategies include profiling to identify bottlenecks, applying algorithmic and memory optimizations, and leveraging compiler settings for efficiency. Debugging is categorized by error types—syntax, logical, and runtime—with tools such as print statements, logging, breakpoints, and memory analysis. For machine learning models, the document examplify using confusion matrices to diagnose performance issues, detailing how metrics like precision, recall, and accuracy help refine model predictions and guide tuning for improved reliability.

3.5 Tests and testing

Testing is a crucial aspect of software development that ensures code behaves as expected under various conditions and meets the specified requirements. It helps identify and address defects before the software is deployed, thereby enhancing the reliability and quality of the final product.

There are several types of tests, each with a specific focus and purpose. Here's a brief overview of some types of tests used in development, along with benefits and how they work. As can be noted, tools such as 'assert' in Python can be used in various types of tests and is not associated with only one type.

Unit tests

- Purpose: To verify that individual units of code (e.g. functions, methods, classes) work correctly in isolation.
- Why: Catching bugs early in the development process and ensuring that each small piece of code behaves as expected.
- How: Each unit test focuses on a single function or method, providing known inputs and checking the outputs against expected values. For example, testing a function that adds two numbers would ensure that given 2 and 3, it returns 5.

```
def test_add():
    assert add(2, 3) == 5
```

When developers modify code—whether through adding new features, refactoring, or fixing bugs—there's a risk that the changes could unintentionally affect parts of the system that were previously functioning as expected. A

regression happens when a modification introduces errors or failures in those unaffected areas.

Unit tests are often automated and run frequently to detect these regressions early. If a unit test that previously passed starts failing after a change, it signals that a regression has occurred, and the developers can investigate and fix the issue before it becomes a bigger problem.

Integration tests

- Purpose: To ensure that different units or modules work together correctly.
- Why: Verifying interactions between components and making sure integrated systems function as expected.
- How: These tests check whether multiple components (like database access and APIs) work together in real-world scenarios. For example, an integration test might check if a web service correctly retrieves data from a database and formats it for display.

```
    def test_service_integration():
        response = get_data_from_service()
        assert response.status_code == 200
```

Integration tests run after unit tests to ensure the correct behavior when components are combined.

Functional tests

- Purpose: To validate the software against functional requirements and ensure it behaves according to the specifications.
- Why: Testing whether specific features work as expected from the end-user's perspective.
- How: Functional tests focus on verifying that specific functionalities (e.g. user login, form submission) behave correctly. They are often higher-level than unit tests and can simulate real user actions.

```
def test_user_login():
    assert login(username="test", password="correct") == "Login successful"
```

These tests ensure that the software meets its functional requirements.

Regression tests

- Purpose: To ensure that new changes or updates to the codebase do not introduce bugs or break existing functionality.
- Why: Maintaining stability and catching issues introduced by recent changes.
- How: Regression tests are a *collection of previously written tests* (often unit, integration, or functional tests) that are rerun after updates. If a test that previously passed now fails, the developer knows the recent changes caused a problem.

Example: If a new feature was added to a project, regression tests would check whether the existing features continue to work as expected after the change.

Smoke Tests

- Purpose: To perform a quick check that the most critical parts of the software work correctly.
- Why: Running quick validations before a more thorough testing phase.
- How: Smoke tests are a subset of tests that ensure that the core functionality of the application is operational. They are often run after a build to confirm that the system is stable enough for further testing.

Example: A smoke test for a web app might ensure that the application loads and that users can log in without errors.

Performance Tests

- Purpose: To evaluate the software's performance under specific conditions (e.g. load, stress).
- Why: Ensuring that the system performs efficiently and remains responsive under high usage.
- How: Performance tests measure various aspects like response time, throughput, and resource usage. This may include load testing (to check how the system handles multiple users) and stress testing (to evaluate the system's behavior under extreme conditions).

Example: A performance test might simulate 10,000 users accessing an API at the same time and measure the response time or memory consumption.

Security Tests

- Purpose: To ensure that the application is free from vulnerabilities and that sensitive data is protected.
- Why: Identifying security flaws such as SQL injection, cross-site scripting (XSS), or unauthorized access.
- How: Security tests simulate attacks on the system to expose weaknesses. These can be automated or using manual penetration tests.

Example: A security test might try to inject malicious SQL code into a login form to check if the system is vulnerable.

Summary of testing types

- Unit: Validate individual components in isolation. Very granular, tests small pieces of code. Often used after each build.
- Integration: Ensure components work together correctly. Tests the interaction between various system components. Used after adding or modifying major components.
- Regression: Ensure that new changes don't break existing functionality. Re-running previous tests on affected areas. After code changes, used before releases.
- Smoke: Quickly validate critical parts of the system after a build. Essential, high-level functionality check. Often applied after each build.
- Performance: Check system performance under load and stress. Tests system performance under various conditions (e.g., load, stress). Run before major releases or updates.
- Security: Identify and mitigate security vulnerabilities. Tests for vulnerabilities, such as SQL injection, or cross-site scripting (XSS). Regularly runs, especially before a public release.
- Functional: Verify specific functions or features work as expected. Tests end-to-end business processes. After adding or modifying features, checking.

Higher level tests slightly more user oriented and excluded from above are e.g. End-to-End (E2E), usability or acceptability tests.

3.5.1 Automated testing and continuous integration

Automated tests are a key component in ensuring software stability as code is modified, enhanced, or refactored. These tests—whether they are unit tests, integration tests, or functional tests—are designed to run automatically, often in response to code changes. By automating tests, developers gain several advantages:

1. Immediate feedback: When tests fail after a change, developers are alerted right away, helping to catch and fix issues early.

2. Regression detection: Automated tests prevent regressions by verifying that new code doesn't break existing functionality.

3. Faster development: Automated tests allow for rapid, iterative development, freeing developers from manual testing and allowing them to focus on feature development.

Using make for test automation

The 'make' tool is often used to automate test execution, especially in C/C++ development but also in many other environments. make allows developers to define rules in a 'Makefile' that specify how to build, test, and deploy code. With make, we can create a standardized set of commands for compiling code, running tests, and generating reports, which can then be executed with a simple command.

For example, here's a basic 'Makefile' that includes targets for running unit tests:

```
# Makefile

# compile code
compile:
    gcc -o my_program main.c

# run tests
test: compile
    ./run_tests.sh

# cleaning up generated files
clean:
    rm -f my_program
```

In this example 'make compile' compiles the program. The 'make test' first compiles the code and then runs a script to execute tests. And 'make clean' removes any files generated during compilation, keeping the environment clean.

By running 'make test', a developer can execute all tests, which is especially helpful when testing frequently.

Continuous Integration (CI) pipelines

Continuous Integration (CI) pipelines are systems that automatically run tests and other checks each time code is committed to the repository. Tools like GitHub Actions, GitLab CI, Jenkins, and CircleCI are commonly used for setting up CI pipelines. A typical CI pipeline includes stages such as:

1. Building the code: Compiling or setting up dependencies to ensure code can be built from scratch.
2. Running tests: Executing unit tests, integration tests, and sometimes functional tests.
3. Static analysis: Checking code for style, quality, and common errors using tools like linters.
4. Deployment (optional): Pushing changes to staging or production environments if all tests pass.

Here's a simplified example of a CI pipeline configuration in GitHub Actions:

```
# .github/workflows/ci.yml

name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run unit tests
        run: pytest tests/
```

In this configuration:

- The CI pipeline triggers on every push to the main branch and on pull requests.

- It sets up the environment, installs dependencies, and runs all tests in the tests/ directory.

This CI pipeline ensures that tests run on every code change, immediately notifying the team if any test fails.

Unit testing automation

By automating unit tests, developers can rapidly verify that the foundational parts of the code are working correctly, providing confidence when changes are made.

Tools like Pytest for Python, JUnit for Java, and Google Test for C++ facilitate unit testing by providing frameworks to:

- Write test cases in a consistent, organised manner.
- Run all tests automatically and generate reports on the results.
- Use mocking and stubbing for testing in isolation by simulating dependencies.

For instance, in Python with Pytest, a unit test might look like this:

```
def add(a, b):
    return a + b

def test_add():
    assert add(3, 4) == 7
    assert add(0, 0) == 0
    assert add(-1, 1) == 0
```

Combined process

By using tools like make, CI pipelines, and automated unit tests, we can create a robust and continuous validation process. Here's how these elements interact:

1. Developers write code and create *unit tests* to verify each part of their code.
2. Automated unit tests can be run locally with make test (or a similar command), ensuring new code functions as intended.
3. When code is committed to the repository, the *CI pipeline* triggers, automatically compiling the code, running tests, and generating feedback for the team.
4. If all tests pass, the pipeline may proceed to *deployment*; if any tests fail, developers receive feedback to address the issues immediately.

This process enhances software quality, speeds up development, and reduces the likelihood of bugs reaching production, making automated testing an essential part of the development workflow.

3.5.2 Summary

This section highlights the importance of various software testing types—unit, integration, functional, regression, smoke, performance, and security tests—in ensuring code functions as intended and meets requirements. Unit tests verify individual components, while integration and functional tests check the interaction between parts and overall system behavior from the user's perspective. Regression tests ensure changes don't break existing functionality, and smoke tests provide quick checks on core features. Performance and security tests assess system efficiency and identify vulnerabilities. Automated testing, often managed with tools like make and CI pipelines, ensures continuous validation, providing immediate feedback on code changes, detecting regressions, and speeding up development, all of which contribute to improved software quality and reduced risk of bugs.

3.6 Practice

Workbook: Chapter 3.
https://github.com/Feyerabend/vm/tree/main/chip8
Scan the QR code to access exercises, code, resources and updates.

jsd askjd kjasnd

Exercises

1. *Explain the limitations of debugging using print statements and compare it with the advantages of using breakpoints in a debugging tool like GDB.* Understanding the pros and cons of different debugging techniques can help you decide the best tool for various scenarios.

2. *Describe a situation where optimizing for memory usage could negatively impact speed, and vice versa. Provide an example from your own experience or research.* Knowing the trade-offs between memory and speed is crucial in writing efficient code.

3. *Implement an algorithm that optimizes memory usage by using a more efficient data structure, and explain how it improves performance.* Understanding how data structure selection impacts memory efficiency is key to writing optimized code.

4. *Explore limitations of verification and testing by examining cases where testing alone cannot ensure program correctness.* Understanding these boundaries emphasizes the need for complementary methods, such as formal verification, in developing robust software systems. Also explore for a deeper understanding what verification entails.