

ZAAHIR ALI

CREATING A BYTECODE VIRTUAL MACHINE IN C

CREATING A BYTECODE VIRTUAL MACHINE IN C

ZAAHIR ALI



A Level Computer Science
September 2023 – version 4.2

Zaahir Ali: *Creating a Bytecode Virtual Machine in C*, A Level Computer Science, © September 2023

Ohana means family.
Family means nobody gets left behind, or forgotten.
— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.
1939–2005

ABSTRACT

Short summary of the contents in English...a great guide by Kent Beck how to write good abstracts can be found here:

<https://plg.uwaterloo.ca/~migod/research/beck00PSLA.html>

*We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.*

— Donald E. Knuth [1]

ACKNOWLEDGMENTS

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schliesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio¹, Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, Henri Menke, Claus Lahiri, Clemens Niederberger, Stefano Bragaglia, Jörn Hees, and the whole L^AT_EX-community for support, ideas and some great software.

Regarding L_YX: The L_YX port was initially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and for the contributions to the original style.

¹ Members of GuIT (Gruppo Italiano Utilizzatori di T_EX e L^AT_EX)

CONTENTS

I	DESIGN OVERVIEW	1
1	INTRODUCTION	3
1.1	Dissecting a Programming Language	4
1.1.1	Parsing	4
1.1.2	Static Analysis	4
1.1.3	Intermediate Representations	5
1.1.4	Optimization	6
1.1.5	Code Generation	7
1.1.6	Virtual Machine	8
1.1.7	Runtime	8
1.2	Alternative Methods and Shortcuts	9
1.2.1	Single-Pass Compilers	9
1.2.2	Tree-Walk Interpreters	9
1.2.3	Transpilers	9
1.2.4	Just-In-Time Compilation	11
1.3	Compilers and Interpreters	11
1.3.1	Compilers	11
1.3.2	Interpreters	12
2	THE C- LANGUAGE	13
2.1	A Semantic Introduction	13
2.1.1	Dynamic Typing	13
2.1.2	Memory Management	14
2.1.3	Data Types	15
2.1.4	Numbers	15
2.1.5	Strings	15
2.1.6	Null	16
2.2	Expressions and Statements	16
2.2.1	Arithmetic	16
2.2.2	Comparison and Equality	16
2.2.3	Logical Operators	17
2.2.4	Statements	19
2.3	Variables, Loops and Functions	19
2.3.1	Variables	19
2.3.2	Loops	19
2.3.3	Functions	20
II	APPENDIX	23
A	APPENDIX TEST	25
A.1	Appendix Section Test	25
A.2	Another Appendix Section Test	25
	BIBLIOGRAPHY	27

LIST OF FIGURES

Figure 1	Transpilers	10
----------	-------------	----

LIST OF TABLES

Table 1	Autem usu id	25
---------	--------------	----

LISTINGS

Listing 1	A floating example (listings manual)	25
-----------	--------------------------------------	----

ACRONYMS

GC	Garbage Collection
AST	Abstract Syntax Tree
IR	Intermediate Representation
VM	Virtual Machine
JIT	Just-In-Time
CPU	Central Processing Unit
API	Application Programming Interface
AOT	Ahead Of Time

Part I

DESIGN OVERVIEW

INTRODUCTION

Engineers have been engaged in the development of programming languages since the inception of computing. Upon establishing communication with early computers, it became evident that this interaction posed significant challenges, prompting us to seek computational assistance. It is noteworthy that, despite the remarkable advancements in contemporary computing, characterized by computers that are a million times faster and equipped with substantially increased storage capacities, the foundational principles governing programming language construction have remained relatively consistent.

While the domain explored by language designers is expansive, the avenues they have pioneered within it are relatively circumscribed. While not all programming languages follow identical trajectories, with some opting for expedient shortcuts, they tend to share fundamental commonalities. This continuity is observed across the spectrum, from Rear Admiral Grace Hopper’s seminal COBOL compiler to contemporary languages that transpile into JavaScript and are often accompanied by rudimentary documentation found in a solitary, modestly edited README nestled within a Git repository.

We can analogise the ascending of a mountain to serve to illustrate the process of language implementation. It commences at the base with the program in its raw source code form, consisting solely of alphanumeric characters. Each phase in the development process involves systematic analysis and transformation, culminating in a representation at a higher conceptual level that elucidates the intended semantics—the author’s instructions for the computer. Upon reaching the summit, an all-encompassing perspective of the user’s program is achieved, affording a comprehensive understanding of its intended function. Subsequently, the journey descends along the opposite side of the metaphorical mountain, progressively converting the highest-level representation into successively lower-level forms, drawing closer to a format amenable to execution by the Central Processing Unit ([CPU](#)).

Let us scale through these mountain trails and introduce certain points of interest. We start with the users source code.

1.1 DISSECTING A PROGRAMMING LANGUAGE

1.1.1 *Parsing*

Parsing is done by a parser, a software device that takes input text and builds a data structure called the Abstract Syntax Tree (AST), a hierarchical structure that gives a systemic representation of the input text and checking it against the lexical grammar. This is either done through top-down parsing (also known as the primordial soup approach), a method where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar, or through bottom-up parsing, a strategy that consists of a parser starting with the input and attempting to rewrite it to the start symbol. The parser attempts to locate the most basic elements of the input, and then the elements containing these and so on.

This process starts with lexical analysis, also known as lexing, or more informally, scanning. In this step of the implementation process, a lexer (or scanner) takes in the linear stream of code and then breaks them down into small chunks, known as tokens. There are several different types of these tokens, ranging from special characters such as ; (known as a separator), to string literals ("Hello, World!") and identifiers (max). For example, let us consider the following expression in the C programming language:

```
z = x + y * 3;
```

In this expression, the lexical analysis would produce the following tokens:

```
[(identifier, z), (operator, =), (identifier, x), (operator, +),  
  (identifier, y), (operator, *), (literal, 3), (separator, ;)]
```

These tokens are defined by the languages lexical grammar, consisting of regular expressions and define the possible lexemes of a token. If there's a mistake when comparing with the grammar rules of the language, the parser lets the programmer know by reporting a syntax error. Note that the parsing process is standard across compilers, interpreters and translators.

1.1.2 *Static Analysis*

Now, the individual characteristics of each language start coming into play. At this point, we know the syntactic structure of the code, such as the nested expressions but not much else.

In an expression like $x + y$, we know we are adding x and y , but we don't know what those names refer to. Are they local variables or global? Where are they defined? This is where we use static analysis, where we can examine programs without executing them, compared

to dynamic program analysis which we perform on programs during their execution.

The analysis process starts with binding, also known as resolution. For every identifier within the code, we find out where they are defined and connect the two together, hence defining the scope of the identifier — the region of source code where a name actually refers to a declaration.

In a statically typed language, a language where a variable is known at compile-time instead of at run-time, this is where type checking happens. If there's an error, for example x and y are of two different types which don't support addition, a type error is reported. Note that for dynamically typed languages, such as the one we will be building, the type checking is done during runtime.

In our metaphorical mountain, this would be the peak, where we can see the entirety of the structure and meaning of the user's program. The semantic insight from this position can be stored in several places:

- Most often, it's stored right back on the syntax tree itself as an attribute, extra fields in the tree nodes that aren't initialised during the parsing process but are inserted later during compilation.
- It may also be stored into a symbol table, a lookup table outside of the syntax tree that contains identifiers — the names of declarations and variables.
- However the most powerful method is where we transform the tree into a new data structure that directly expresses the semantics of the code. This is known as an intermediate representation.

This is the end of what's known as the front end of the implementation process.

1.1.3 *Intermediate Representations*

We can think of each stage of the compilation process as aiming to organize the data representing the user's program in a fashion that makes the next stage of the process easier to implement. The front end of the compilation process is specific to the source language of the project, while the back end is concerned with the architecture of the machine that the program will run on. This is the "middle end" of the compilation process.

At this point, code is stored in a Intermediate Representation (IR) that isn't tied to either the source code or the destination architecture, and acts as a bridge, or interface, between the two languages.

This means that if we want to support multiple source languages and target platforms we don't need to write multiple compilers for each possible combination of language and architecture, but instead, using a shared intermediate representation, we write one front end for each source language and one back end for each target architecture. This means we can mix and match, drastically reducing development time. However, this isn't the only reason we'd like to use an intermediate representation.

1.1.4 Optimization

Once we have a semantic overview of the user's program, we can substitute it with a different program that performs the same operations, just more efficiently, optimizing it.

One form of this is constant folding; if one expression always evaluates to the same value (e.g. it is constant) we can evaluate the expression at compile time and replace the code calculating the constant with the resulting number. If the original program was this:

```
footballVolume = 4/3 * 3.14159 * (1)^3;
```

We could calculate this in the compiler and replace the code with:

```
footballVolume = 4.18878667;
```

Another method is known as strength reduction, where expensive operations are replaced with equivalent but less expensive operations. Consider the following C code snippet:

```
int i, sum = 0;
for (i = 1; i <= n; ++i) {
    sum += i;
}
printf("sum: %d\n", sum);
```

Assuming no arithmetic overflow, we can rewrite the code to be:

```
int sum = n * (1 + n) / 2;
printf("sum: %d\n", sum);
```

This makes use of the following mathematical standard summation for consecutive natural numbers:

$$\sum_{n=1}^i n = \frac{1}{2}n(n+1) \quad (1)$$

This same method of choosing more computationally efficient algorithms for performing operations is the key to strength reduction. However, note that the "optimized" version might actually be slower than the original version if n were sufficiently small and the particular hardware happens to be much faster at performing addition and looping operations than multiplication and division.

Furthermore, while the term “optimization” shares its root with “optimal”, it is uncommon for the optimization process to yield a system that is genuinely optimal in all respects. Typically, a system can be optimized, not in an absolute sense, but rather in relation to a specific quality metric, which may conflict with other potential metrics. Consequently, the optimized system usually achieves optimality only within a particular application or for a specific target audience.

For instance, one might choose to reduce the execution time of a program, even if it results in increased memory consumption. In situations where conserving memory space is critical, deliberately opting for a slower algorithm may be a preferable trade-off. Frequently, there is no universally ideal design that can excel in all scenarios. Engineers must make calculated trade-offs to optimize the attributes that are most relevant to the task at hand. Several programming languages, such as CPython and Lua generate relatively unoptimized code and in exchange focus on performance effort during the runtime.

1.1.5 Code Generation

After optimizing the user’s source code, we move to converting into machine code, a form that the machine can actually run. This is known as the generating code, where the code that is generated is usually some permutation of assembly code.

Although somewhat unrelated, we should mention that when code generation takes place at runtime, such as in Just-In-Time (JIT) compilation, it is important that the process is efficient with respect to space and time. Despite its generally generating less efficient code, JIT code generation can take advantage of profiling information that is available only at runtime.

This is the end of the middle end of the compilation process, and now part of the back end, the downward descent down our metaphorical mountain. We have gained an understanding of the user’s program and now devolve the code, each step of the back end making the code more and more primitive so a simple CPU can execute it.

We can either produce instructions for a real CPU or a virtual one. If we generate machine code for a real CPU we are given an executeable that is loaded and executed directly on the chip by the machine’s operating system. This results in fast code, but the generating process is significantly long due to complex pipelines, and a large and complicated instruction set.

Furthermore, the produced executeable is also tied to the machine’s architecture, say x86 or ARM, resulting in a lack of portability. To circumvent this, we can have our compiler produce virtual machine code, code that runs for some hypothetical machine. This is known as bytecode, since each instruction is often just a single byte long.

This synthetic set of instructions are designed to be based more around the properties of the source language rather than the quirks of a computer architecture, sort of like a dense, binary encoding of the language's low-level operations.

1.1.6 *Virtual Machine*

However, this virtual code is useless without a chip that can understand it. We have two options, the first being a 'mini-compiler' for each real architecture we want our program to run to that converts the bytecode to native machine code for that language. This may seem counterproductive since we are trying to circumvent the architecture roadblock, but this step is relatively simple, and we are able to retain the rest of the compiler across the machines we wish to support, meaning the bytecode essentially functions as an intermediate representation.

The other option would be a Virtual Machine (VM), a program that emulates our hypothetical chip and supporting our virtual architecture at runtime. This implementation is slower than the prior method since every instruction must be simulated at runtime before executing, however in exchange a VM offers simplicity and portability.

There are two main types of virtual machines, system virtual machines (also known as full virtualisation VMs) and process virtual machines. System virtual machines are a complete substitute for a real machine, providing the functionality to execute entire operating systems, while process virtual machines are designed to execute certain programs in an isolated environment. This is the type we will be using throughout the project.

1.1.7 *Runtime*

We are finally at the end of the process of moulding the user's program into a form that we can run. The last step is running it. If we had compiled it to real machine code we would just tell the operating system to load and run the executable file that is generated, however with bytecode we need to start the virtual machine and load the program through that.

The runtime, often referred to as the runtime environment, is a crucial component in most programming languages, except for the most basic low-level ones. It provides essential services during program execution. One such service is memory management automation, which necessitates the presence of a garbage collector to reclaim unused memory.

Additionally, if the language supports "instance of" tests to determine object types, a mechanism for tracking the type of each object during runtime is required. The integration of these services into a program depends on the language's nature.

In the case of fully compiled languages, the code responsible for the runtime is directly inserted into the resulting executable. For example, in Go, each compiled application includes its own copy of Go's runtime, embedded within it. Conversely, languages that run within interpreters or virtual machines have the runtime environment residing in those environments. This operational model is commonly employed in languages like Java, Python, and JavaScript.

1.2 ALTERNATIVE METHODS AND SHORTCUTS

This comprehensive journey covers every conceivable phase of implementation, and many languages do indeed traverse this entire route. However, there are a few shortcuts and alternative paths available.

1.2.1 *Single-Pass Compilers*

A single-pass compiler, also known as a one-pass compiler is a compiler that passes through each compilation unit, such as a source file, only once and immediately transforms this into the final machine code that will be run. This means it skips the intermediate representation phase, does not allocate a syntax tree and does not reprocess the compilation unit in each sequential pass. In theory, this produces a smaller and faster processor, but compiler optimization is highly limited due to the limited scope of information available. Furthermore, many effective forms of compiler optimization, such as strength reduction, require multiple passes over structures such as a nested loop.

1.2.2 *Tree-Walk Interpreters*

This approach functions by executing code right after it is parsed to an AST or using it to generate native code just-in-time. This is done by the interpreter traversing the syntax tree one branch at a time, evaluating each node as it parses. This again means each sentence only needs to be parsed once, and the [AST](#) keeps the global program structure and relations (which is lost in bytecode representation). However, an [AST](#) causes large overhead from visiting the tree (a less sequential representation causes traversal of more pointers) and because syntax nodes are useless during the process.

1.2.3 *Transpilers*

Creating a complete back end for a language is often a substantial undertaking. If we happen to possess a pre-existing, generic intermediate representation as our target, we can seamlessly integrate our

front end with it. However, in cases where this isn't an option, we may find ourselves facing a daunting challenge.

Here's an alternative approach to consider: Instead of expending significant effort in the back end to transform the semantics into a lower-level target language, we can generate a string of valid source code for another language that shares a similar high-level nature with our own (compared to a traditional compiler which translates from high level to low level). This opens up the possibility of utilizing the existing compilation tools for that language to facilitate the transition from our elevated position to an executable form.

This methodology was once referred to as a "source-to-source compiler" or "transcompiler." With the emergence of languages that compile to JavaScript for browser-based execution, it has gained a certain trendy reputation and is commonly known as a "transpiler" among the more hip programming circles.

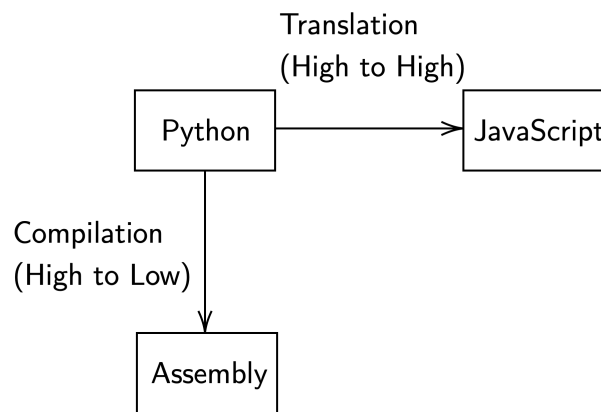


Figure 1: Transpilers

Another valuable application of source-to-source compiling involves the translation of legacy code to adapt it to the next iteration of the underlying programming language or an Application Programming Interface (API) that introduces changes, potentially breaking backward compatibility. This process also includes automatic code refactoring, which is useful when programs are impractically large or time-consuming to be done by hand.

The parser of a transpiler is identical to other compilers, and analysis may be skipped if the source language is similar to the target language, meaning the corresponding syntax is output to the intended language straight after parsing.

However, if the two language are not as syntactically similar, semantic analysis will be conducted, possibly with compiler optimizations as well, analogous to a full compiler. The difference comes at code generation, where instead of resulting in machine code, or some

other binary language, we are produced with the program in the desired language which we can then run through that languages compilation process.

1.2.4 *Just-In-Time Compilation*

This approach is more risky than those above, and involves compiling once the program is loaded at run-time on the user's machine, natively for the architecture of the end machine. This means you we can then compile for a machine architecture which was originally unknown and furthermore at high speed. Although it is possible to compile the source code to machine code at run-time, often it is byte-code that is translated.

This is actually a combination of two previous approaches in translation to machine code, Ahead Of Time (AOT) compilation and interpretation, and JIT tries to combine the advantages of both, albeit with some of the drawbacks. This compilation process results in speeds similar to pre-compiled code and flexibility comparative to interpretation, hence it's use in dynamically typed languages. The overhead of this process however is a combination of the methods it draws upon, compiling and linking (not just interpreting)

At the highest levels JITs insert profiling hooks to understand which areas of the programs are most performance critical and over time recompile those with greater and more advanced optimizations.

1.3 COMPILERS AND INTERPRETERS

We would like to start by defining a clear line between compilers and interpreters, which so far have almost been used interchangeably. In theory, a programming language can have both a compiler and interpreter, but in practice it often just implements just one.

1.3.1 *Compilers*

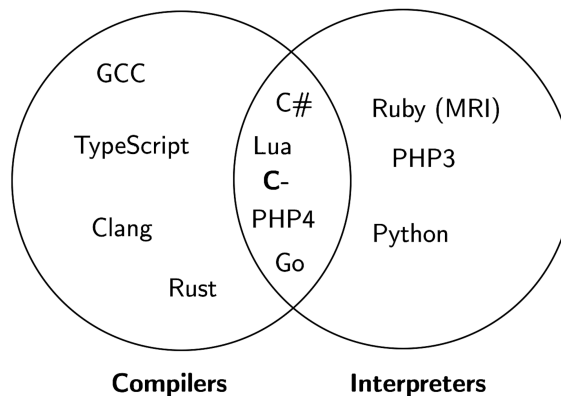
To distinguish compilers from interpreters, it's crucial to understand that compilers are primarily concerned with translating source code into another form, typically a lower-level representation or machine code. This transformation allows for efficient execution of the program in the target environment. For example, when a high-level programming language like C++ is compiled, the source code is translated into machine code, and the result is an executable file. Importantly, the compilation process **does not** involve the direct execution of the program; instead, it produces an independent executable that can be run later. Consequently, once the compilation is complete, the original source code is no longer required to execute the program, making it suitable for distribution and deployment on various sys-

tems, as well as a greater security factor for preventing modification of the source code. This separation of compilation and execution is a key characteristic of compilers, as they focus on generating efficient stand-alone executables.

1.3.2 Interpreters

In contrast to compilers, interpreters operate by directly executing the source code “on the fly”. When a programming language is implemented with an interpreter, it takes the source code as input and immediately begins executing it. This approach is often described as “running from source” because there is no intermediate step of generating a separate executable file. Instead, the interpreter interprets and executes each line or statement of the source code in real-time. This immediate execution can be advantageous for tasks like debugging, as errors are reported as they occur in the source code. However, it can also be less efficient than compiled code since the interpreter must repeatedly analyze and execute the code line by line. Interpreters are commonly used in scripting languages like Python and JavaScript, where rapid development and platform independence are more critical than raw execution speed.

An example of a language utilizing both implementations is CPython, which from the users perspective seems like as an interpreter — they can clearly see the program run from source. However, under the hood the code is parsed and converted into an internal bytecode format, which is then executed within the [VM](#), which is compiler like. We illustrate some other languages and whether they use a compiler, interpreter or both:



Our implementation, C-, lives in the middle area as it internally compiles to bytecode and then sends it to the [VM](#).

THE C- LANGUAGE

Although throughout the rest of this project we'll explore every nook and cranny of the C- language, we spend this chapter to provide a small introduction to the general syntax, as a sort of preview of what our interpreter will use. We'll leave out many of the edge cases and details for later on and start with the mandatory "Hello, World!".

```
print "Hello, world!";
```

```
// Prints Hello, world!
```

The `//` line comment and trailing semicolon at the end of the expression point towards C-like syntax, which is what C- derives from. Note that there are no parenthesis around the string because we plan to have `print` as a built-in statement rather than as a library function.

2.1 A SEMANTIC INTRODUCTION

2.1.1 *Dynamic Typing*

Unlike the language we are using to implement it, C- is dynamically typed, meaning that type is associated with run-time values rather than named variables, as type checking is done at run-time. We show a following example for one of the features that type checking provides.

```
var a = 1;
var b = "two";
```

```
print a + b;
```

```
// Returns a type error; we are trying to sum a string and
integer
```

We choose this method of implementation because of its ease compared to a statically typed language, and because we can implement features such as reflection (the ability for a process to examine or modify its own behaviour) or downcasting (casting a reference of a base class to one of its derived classes). In the context of C++, downcasting manually casts the base class's object to the derived class's object, meaning we must specify the explicit typecast. Furthermore, the derived class can now add new functionality such as new data members which could not apply to the base class. Let's illustrate an example in C++.

```
class Parent
```

```

{
    public:
        void base()
        {
            cout << "This is the Parent class function." << endl;
        }
};

class Child : public Parent
{
    public:
        void derive()
        {
            cout << "This is the Child class function." << endl;
        }
};

int main ()
{
    Parent pobj; // create the Parent's object
    Child *cobj; // create the Child's object

    // explicit type cast is required in downcasting
    cobj = (Child *) &pobj;
    cobj -> derive();

    return 0;
}

// Outputs "This is the Child class function."

```

2.1.2 Memory Management

High level languages typically eliminate the tedious approach of freeing and allocating system memory (such as in C with `free()`), which is the approach we will be taking with C-. There are two approaches to automatic memory management, reference counting and tracing garbage collection, also just known as garbage collection (GC). Note that reference counters are method of “garbage collection” but tracing garbage collection are used so often that both terms are now synonymous.

Reference counters are a programming technique of storing the number of references, pointers, or handles to a resource, such as an object, a block of memory or disk space. These objects are reclaimed straight they can no longer referenced, unlike garbage collection, typically used in systems with limited memory to maintain responsiveness. Furthermore, it means non-memory resources such as operating system objects can be effectively managed, which are often much scarcer than memory. Most languages that started out using refer-

ence counters end up adding a full tracing garbage collector, or at least enough of one to clean up object cycles.

Tracing garbage collection consists of determining which objects should be deallocated (“garbage collected”) by tracing which objects are reachable by a chain of references from certain “root” objects, and considering the rest as “garbage” and collecting them. This approach is typically more difficult due to the need to work at the level of raw memory.

2.1.3 *Data Types*

In the C- programming language, built-in data types are the atomic bricks that store every item of a program, and we only need to employ a few.

Firstly, the boolean, a data type which has only two possible values, and a string literal to indicate each, unlike C which uses an unsigned integer. They are intended to signify true logical and mathematical or the negation of it, and we often see it used in conjunction with comparison operators.

```
true; // not false
false; // not true
```

2.1.4 *Numbers*

Unlike C++, with number types such as `int`, `double` and `long long`, C- only utilizes a floating point double-precision (upto 15 decimal digits) number. This keeps things relatively simple while remaining relatively flexible and covering a wide range of numbers, only sacrificing a small amount of speed and memory. Furthermore, we will not implement hexadecimal or other number system bases, settling for only basic integer and decimal literals.

```
452; // integer
923.00123; // floating point number
```

2.1.5 *Strings*

We’ve encountered this data type multiple times, even right from the beginning in our `Hello, World!` example. These are just a sequence of characters that store typically human-readable data, such as words or sentences and often uses character encoding, which will be one of the challenges to implement later on.

```
""; // empty string
"string"; // standard string
```

2.1.6 *Null*

Finally we have our final built in data type, `null`, which is almost identical to its C counterpart. The null pointer is used to indicate that the pointer does not refer to a valid object. Note that normally the null pointer is not the same as an uninitialised pointer, but we will be using them as one.

2.2 EXPRESSIONS AND STATEMENTS

If data types are the bricks to our C- city, expressions and statements would be our houses. We should make a distinction between statements, which are executed, and expressions, which are evaluated. Expressions always evaluate to a value, which statements do not. However, expressions are often used as part of a larger statement.

2.2.1 *Arithmetic*

These will be familiar and are just the standard basic arithmetic operations, bar modulo.

```
183 + 298; // returns 481
901 - 873; // returns 28
81 * 12; // returns 972
672 / 16; // returns 42
```

We call the number's we are computing the operations upon, the operands and the "+, -, * or /" the operator. As we are performing this process with two operands, we call them binary operators. Furthermore, most of these are infix operators, except - which is both an prefix operator (when used for negation) and a infix operator (when used for subtraction). We also have postfix operators (which come before the operands) and misfix operators, which have more than two operands and the operators are interleaved between them. An example from C is the "ternary" operator, illustrated below:

```
condition ? thenArm : elseArm;
```

All these operators (except +, which can be used to concatenate strings) only work on the number type and if any other type is fed through, a type error will be returned.

2.2.2 *Comparison and Equality*

We can make use of the boolean type by comparing number types using comparison operators. This is especially useful for iterating through loops for a set amount of time, such as if we wanted to print the first ten consecutive natural numbers.

```
var a = 1;
```

```
while (a <= 10) {
    print a;
    a = a + 1;
}
```

We have four different comparison operators, listed below:

```
2 < 3; // less than
2 <= 4; // less than or equal to
5 > 1; // greater than
19 >= 19; // greater than or equal to
```

We can also use equality operators to test any two different types, any this also returns a boolean type. For example:

```
1 == 2 // returns false
"a" != "b" // returns true
```

Note that values of different types can be compared late, and can never be equivalent:

```
1 == "one"; // returns false
```

Equality is also often used to test if an element already exists in a set, or to access to a value through a key.

2.2.3 Logical Operators

In logic, a logical operator is a logical constant which is used to connect logical formulas. Common operators include negation, disjunction, conjunction, implication and equivalence, and are interpreted as truth functions. Logical operators can be used to link zero or more statements, so one can speak about n-ary logical operators. Note that the boolean types, True and False are known as zero-ary operators, while negation is known as a one-ary operator, and so on.

Outside the field of computer science, where we denote this with symbols such as AND and NOT, these symbols are represented non-ambiguously with special symbols. We list them down below.

- Negation (NOT) – This is an operation that is intuitively interpreted as true when some proposition P is false, or not true, and false when some proposition P is true. We can denote this in formal logic as $\neg P$ (the most common notation), $\sim P$ or NP . This is similar to how in mathematical set theory \setminus is used to indicate 'not in the set of' (i.e $U \setminus A$ is the set of all members of U that are not members of A). The truth table is as follows:

P	$\neg P$
True	False
False	True

We will use the ! character as a prefix to denote NOT.

```
!true; // returns false
!false; // returns true
```

- **Conjunction (AND)** – This is an operator that takes two operands, say A and B and is only true when some both are true. We denote this We most often notate this as \wedge and can define it in terms of the NOT and OR functions as $A \wedge B = \neg(\neg A \vee \neg B)$. Below is the truth table for the AND operator:

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False

In C-, we'll just denote this with and.

```
true and false; // returns false
true and true; // returns true
```

- **Disjunction (OR)** – This is an operator that returns True if either of its operands A or B are true and False if both A and B are false. This is typically denoted with \vee and we can define it in terms of other operators as $A \vee B = \neg((\neg A) \wedge (\neg B))$. Below is the truth table:

A	B	$A \vee B$
True	True	True
True	False	True
False	True	True
False	False	False

In C-, we'll again just denote this with the word or.

```
false or true; // returns true
false or false; // returns false.
```

In C-, and programming languages in general and and or are more like control flow structures rather than an operator. For example, if the left operand of a or statement is true, the right statement is skipped, effectively causing a short circuit.

From an overview we can say that all these operations have the same associativity and precedence as in C, and that this order can be modified with ()

2.2.4 Statements

Recall that statements do not evaluate to a value, unlike expressions, and instead aim to produce an effect, say by modifying a state, reading input or producing output. Again, if we go back to our Hello, World! example, we have created a statement as `print` evaluates a single expression and displays the result.

```
print "Hello, world!";
```

An expression followed by a semicolon (;) promotes the expression to an expression statement. Furthermore, we can wrap multiple statements into where only one is expected using a block, which affects scoping.

```
{
  print "a";
  print "b";
}
```

2.3 VARIABLES, LOOPS AND FUNCTIONS

2.3.1 Variables

We can declare variables using `var` statements, meaning we do not usually have to specify the type. If we don't include the initializer statement, the variable's value defaults to `nil`.

```
var a = "a"; // stores the string "a"
var b; // stores the nil type
```

We can then access and remodify variables by calling on it's name.

```
var a = "b";
print b; // returns "b"
```

2.3.2 Loops

Loops are essential for the control flow of a program, meaning we can skip code to execute blocks of code multiple times. Although we could make use of our logical connectives `and`, `not` and `or` for branching and recursion techniques for iterating through code, we feel it would be more convenient to add simple looping statements into C-.

We include three statements from the C language, starting with `if`. When our interpreter finds an `if` statement, it expects a boolean condition, and evaluates that condition. If the condition is true, then the following statements are executed, while if it is false, the nested block is skipped and the code (possibly within an `else` statement) after is executed.

```

if (condition) {
    print "a";
} else {
    print "b";
}

```

We can think of our next statement, the while loop as a set of repeated if statements, a control flow statement that allows code to be repeatedly executed based on some boolean condition. We return to our example of printing the first ten natural numbers

```

var a = 1;
while (a <= 10) {
    print a;
    a = a + 1;
}

```

We have decided to omit the do while statement, as it is similar enough to the while statement. Finally we have for loops, which are used mainly for iteration; running a section of code repeatedly until some condition is satisfied. Let's rewrite our code for printing the first ten natural numbers using a for loop.

```

for (var a = 1; a < 10; a = a + 1) {
    print a;
}

```

Some languages implement a for-in or foreach loops for explicitly iterating over various sequence types, which can be handy, but we'll stick with C's standard for loop.

2.3.3 Functions

We carry over the same syntax from C for calling functions:

```
function(a, b);
```

Note that we can abstain from passing any parameters to the function, which still calls upon it, but we still need the empty parenthesis otherwise we just refer to the function.

Declaring functions can be done with `fun`, similar to `var`. Recall that a declaration binds the function's type to its name so that calls can be type-checked but does not provide a body. A definition declares the function and also fills in the body so that the function can be compiled. This distinction is not very meaningful in C- because it is dynamically typed.

Although these terms are used interchangeably, we also clarify the difference between a parameter and argument. A parameter is a variable that holds the value of the argument inside the body of the function, while an argument is an actual value you pass to a function when you call it.


```
fun sum(a, b) {  
  return a + b;  
}
```

Note that a function body is always a block, and that a function always has to return a value, otherwise it just implicitly returns a value of the `nil` type. We can call multiple functions below, as shown below:

```
fun identity(a) {  
  return a;  
}  
  
print identity(sum)(a, b);
```

We can also declare functions within other functions:

```
fun global() {  
  fun local() {  
    return "a";  
  }  
  
  local();  
}
```


Part II

APPENDIX

APPENDIX TEST

Lorem ipsum at nusquam appellantur his, ut eos erant homero concludaturque. Albucius appellantur deterruisset id eam, vivendum partiendo dissentiet ei ius. Vis melius facilisis ea, sea id convenire referrentur, takimata adolescens ex duo. Ei harum argumentum per. Eam vidit exerci appetere ad, ut vel zzril intellegam interpretaris.

More dummy text.

A.1 APPENDIX SECTION TEST

Test: [Table 1](#) (This reference should have a lowercase, small caps A if the option `floatperchapter` is activated, just as in the table itself → however, this does not work at the moment.)

LABITUR BONORUM PRI NO	QUE VISTA	HUMAN
fastidii ea ius	germano	demonstratea
suscipit instructor	titulo	personas
quaestio philosophia	facto	demonstrated

Table 1: Autem usu id.

A.2 ANOTHER APPENDIX SECTION TEST

Equidem detraxit cu nam, vix eu delenit periculis. Eos ut vero constituto, no vidit propriae complectitur sea. Diceret nonummy in has, no qui eligendi recteque consetetur. Mel eu dictas suscipiantur, et sed placerat oporteat. At ipsum electram mei, ad aequae atomorum mea. There is also a useless Pascal listing below: [Listing 1](#).

Listing 1: A floating example (listings manual)

```
for i:=maxint downto 0 do
begin
{ do nothing }
end;
```


BIBLIOGRAPHY

- [1] Donald E. Knuth. “Computer Programming as an Art.” In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

DECLARATION

Put your declaration here.

High Wycombe, September 2023

Zaahir Ali

COLOPHON

This document was typeset using the typographical look-and-feel classicthesis developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". classicthesis is available for both \LaTeX and \LyX :

<https://bitbucket.org/amiede/classicthesis/>

Happy users of classicthesis usually send a real postcard to the author, a collection of postcards received so far is featured here:

<http://postcards.miede.de/>