ZAAHIR ALI

CREATING A BYTECODE VIRTUAL MACHINE IN C

[September 25, 2023 at 20:55 – classicthesis version 4.2]

CREATING A BYTECODE VIRTUAL MACHINE IN C

ZAAHIR ALI

A Level Computer Science

September 2023 – version 4.2

We ought not to be ashamed of appreciating the truth and of acquiring it wherever it comes from, even if it comes from peoples distant and nations different from us. For the seeker of truth nothing takes precedence over the truth, and there is no disparagement of the truth, nor belittling either of him who speaks it or of him who conveys it. The status of no one is diminished by the truth; rather does the truth ennoble all.

— Abū Yūsuf Ya'qūb ibn Ishāq al-Kindī,
On First Philosophy (ca. 833-842 AD)

ABSTRACT

This project is a bytecode virtual machine interpreter for the custom language C-, submitted as part of the requirements for the A Level Computer Science qualification. As part of the GUI requirements of the project we also include a playground for C-, where code is syntax highlighted, and can be executed while viewing the intermediate bytecode and Virtual Machine (VM) trace.

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
- 2.4 Classes 21
- 2.5 The Standard Library 23

II ANALYSIS 25

3 A BYTECODE IMPLEMENTATION 27
- 3.1 Why not AST? 27
- 3.2 Why Bytecode? 27

3.3 Laying the Foundation 28
3.3.1 Dynamic Arrays 29
3.3.2 Disassembling Chunks 34

BIBLIOGRAPHY 37

LIST OF FIGURES

Figure 1 Transpilers 10
Figure 2 Implementation Process 28

LIST OF TABLES

LISTINGS

ACRONYMS

GC Garbage Collection
OOP Object Orientated Programming
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

1 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 hierachal 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 seperator), 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 entirity 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 if 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 langauge 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 a 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 reducation, 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:

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

This same method of choosing more computationally efficient algorithms for performing operations is the key to strength reducation. 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 coverting 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 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 out 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 able to retain the rest of the compiler across the machines we wish to support, meaning the bytecode essentially functions as a intermediate representation.

The other option would be a VM, a program that emulates our hypothetical chip and supporting out 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, provdiing the functionality to execute entire operating systems, while process virtual machines are designed to execute certain program in an isolated enviroment. 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 executeable 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 visitng 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.

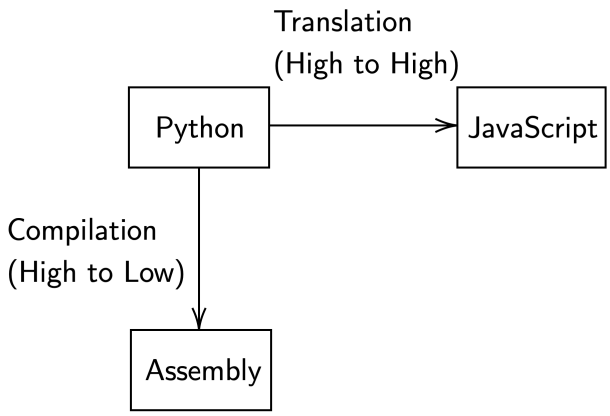


Figure 1: Transpilers

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 furtheremore at high speed. Although it is possible to compile the source code to machine code at run-time, often it is bytecode 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 advantges of both, albeit with some of the drawbacks. This compilation process results in speeds similar to pre-compiled code and flexibility comparitive 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*

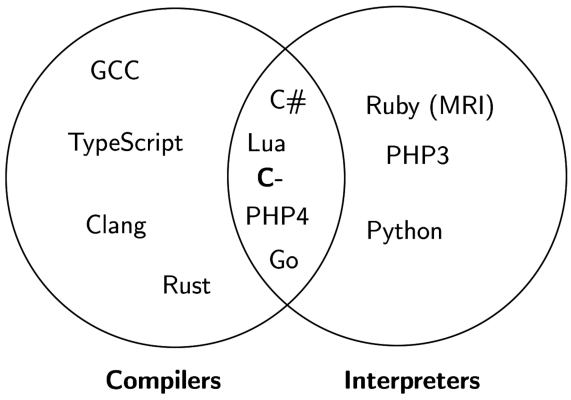
To distinguish compilers from interpreters, it's crucial to understand that compilers are primarily concerned with translating source code into another form, typically a lower-level representation or machine code. This transformation allows for efficient execution of the program in the target environment. 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 lagnuages 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.

2 THE C- LANGUAGE

Although throughout the rest of this project we'll explore every nook and cranny of the C- languge, we spend this chapter to provide a small introduction to the general syntax, as a sort of preview of what our intepreter 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 it's 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 << "parent" << endl;
        }
};

class Child : public Parent
{
    public:
        void derive()
        {
            cout << "child" << 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 "child"
```

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 collecter, 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 indentical 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 unitialised 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, including modulo.

```
183 + 298; // returns 481
901 - 873; // returns 28
81 * 12; // returns 972
672 / 16; // returns 42
254 \% 32 // returns 30
```

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. Futhermore, 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, conjuction, 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 represneted non-ambigiously 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 false. 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
```

- Disjuntion (`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 langauges 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 modifified with `()`

2.2.4 *Statements*

Recall that statements do not evaluate to a value, unlike expressions, and instead aim to produce and 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` evalues 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 convinient 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 evalues 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 ikmplement 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();
}
```

Here, `local()` accesses a local variable declared outside of its body in the surrounding function, which works by `local()` holding on to references to any surrounding variables that it uses so that they stay around even after the outer function has returned. Functions, such as `local()`, that do this are known as closures.

2.4 CLASSES

Classes are a key feature of Object Orientated Programming (OOP), a programming paradigm based on the concept of objects, which contain data and code. This data is stored as fields (which is known as attributes or properties), and the code is in the form of procedures (known as methods).

Although most languages using OOP such as C++, C# and Java uses classes, we also have another possible type of objects; prototypes.

In regular class-based languages, the two main ideas are classes (extensible templates for creating objects) and instances (concrete occurences of any object). Classes contain the methods and inheritance chain. To call a method on an instance, there is always a level of indirection, meaning you have look up the instance's class and then you find the method there.

Prototype-based languages merge these two concepts. There are only objects—no classes—and each individual object may contain state and methods. Objects can directly inherit (known as delegate in prototype-based languages) from each other, meaning that prototypal languages are more fundamental than classes. This also makes

them a lot easier and quicker to implement and have a lot more options for abstract and unusual data patterns.

However, we find that often programming languages utilizing prototypes are less comfortable to the average user, which is an important factor for desigining C-.

We can declare a class and it's methods as follows:

```
class a {
  b() {
    print "c";
  }

  d(e) {
    print "f" + e;
  }
}
```

The body of a class contains its methods, which are declared like functions, but without the keyword. When the class declaration is executed, C- creates a class object and stores that in a variable named after the class. To keep things simple in C-, the class itself is a factory function for instances:

```
var a = b();
print b; // returns the b instance
```

Classes aren't very useful by themselves, and we need to implement some sort of method to encapsulate behvaiour and state together, which can be done using the notion of fields. Like other dynamically typed languages, C- lets us freely add properties onto objects.

If we want to access a field or method on the current object from within a method we can use the `this` statement. Note that assigning to a field creates it if it doesn't already exist.

```
class a {
  init(b, c) {
    this.b = b;
    this.c = c;
  }
}

var d = a("e", "f");
d.g("h");
```

Part of encapsulating data within an object is ensuring the object is in a valid state when it's created, which can be done by defining an initializer, `init()`. Any parameters passed to the class are forwarded to its initializer and it is called automatically when the object is constructed.

C- supports single inheritance, which can be declared by using a less-than (<).

```
class a < b {
  ;
}
```

Here `a` is the derived class (also known as the subclass) while `b` is the base class (also known as the superclass). Every method defined in the superclass, even `init()`, is also available to its subclasses. To call a method on our own instance without hitting our own methods can be done with `super`.

In a true OOP language every object is an instance of a class, even primitive values like numbers and Booleans, but in C- we aim to keep the feature set relativelt minimal.

2.5 THE STANDARD LIBRARY

This is the last section in the overview, the standard library, whcih is the core set of instructions and functionality we directly implement in the interpreter and that all user-defined behavior is built on top of.

We've already seen the `print` function which can be used to output data to the user, and we'll need a function which can be used to record the amount of time a program takes to be able to compare different optimizations, so we use `clock` which returns the number of seconds since the program started. Other utility functions include:

- `exit(x)` – exits the process with the number `x` as the exit code. `x` must be a whole number between 0 and 255 inclusive.
- `str(x)` – converts the value `x` into a string.
- `type(x)` – returns the type of the value `x` as a string.
- `eprint(x)` – prints the value v to the standard error stream
- `argc()` – returns the number of command line arguments the interpreter was launched with.
- `argv(i)` – returns the command line argument at position `i`; zero should be the interpreter program, and numbers up to (but not including) the return value of `argc()` are additional arguments.
- `chr(x)` – returns a one-character string whose first byte is the number represented by `x`. The valid range for `x` depends on how the platform treats C's char type; for signed char it can be any whole number between -128 and 127 inclusive.

We also add some basic mathematical functions:

- `ceil(x)` $\lceil x \rceil$ – returns the number `x` rounded up to the nearest whole number. For example, 0.4 would be rounded to 1.

- `floor(x)` $\lfloor x \rfloor$ – returns the number `x` rounded down to the nearest whole number. For example, 1.9 would be rounded to 1.
- `round(x)` – returns the number `x` rounded to the nearest whole number. For example, 1.4 would be rounded to 1 and 1.6 would be rounded to 2.

We plan to add trigonometric functions such as `sin(x)` as an extenstion to the project.

Part II

ANALYSIS

3 A BYTECODE IMPLEMENTATION

3.1 WHY NOT AST?

We choose a bytecode implementation for it's speed compared to the main rival candidate for an interpreter; a tree-walk interpreter. We've explained the general reason for this in our design overview.

In more depth, when we write any piece of code that's to be compiled using an AST, it's just not memory efficient. Each fragment of code becomes an AST node. Take the expression `1 + 2` turns into a flurry of objects with pointers, each adding an extra 32 or 64 bits of overhead to the object. Not only this, when spreading our data across the heap in a loosely connected web of objects, we have problems with spatial locality.

Spatial locality (also known as data locality) refers to the use of data elements within relatively close storage locations. Modern CPUs can process data much faster than they can retreive it from RAM, leading to the use of (multiple layers of) caching. If a piece of memory it needs is already in the cache, it can be loaded more quickly, even upto two orders of magnitude faster.

The CPU "predicts" what data we need by pulling in adjacent bytes to what is currently being read from RAM and stores them in cache. If our program next requests some data close enough to be inside that cache line, we end up with a faster program experience. To take advantage of this, the way we represent code in memory should be dense and ordered like it's read.

However, when using an AST, the sub-objects can be stored anyway. Every step the tree-walker takes where it follows a reference to a child node may step outside the bounds of the cache and force the CPU to stall until the next data can be recalled from RAM. Just the overhead of the tree nodes with all of their pointer fields and object headers tends to push objects away from each other and out of the cache.

3.2 WHY BYTECODE?

Let's consider the other end of the speed spectrum first, compiling to native machine code. Compiling directly to the native instruction set the chip supports is what the fastest languages do, where the CPU cranks through the instructions, decoding and executing each one in order. There is no tree structure like our AST, and control flow is handled by jumping from one point in the code directly to another. This comes at a cost, as firstly this comilation process is not easy,

especially with newer chips which more complicated architectures. Furthermore, there is no portability, as we'd only be producing for one architecture type of the multiple that are used today. To get our language on all of them, we'd need to learn all of their instruction sets and write a separate back end for each one.

Bytecode is in the middle of both extremes of these implementations, retaining the portability of a tree-walker and sacrifices some simplicity for an increase in performance. Although bytecode resembles machine code in it's dense, linear sequence of binary instructions, simpler, higher-level instruction set than any real chip. This keeps the overhead low, and takes advantage of the caching feature of a chip.

It's essentially like writing for an idealised fantasy instruction set for some perfect architecture. However, obviously when we develop for such an architecture that doesn't exist, we can't actually run it normally. This means we need to write an emulator – a simulated chip written in software that interprets the bytecode one instruction at a time and simulates what the perfect architecture would do. This is known as a VM.

This does add overhead, which is why using a bytecode implementation is not as fast as writing native machine code, but in return we receive portability to run on basically any hardware we want. Below we outline our implementation process, although we probably won't build each phase in order.

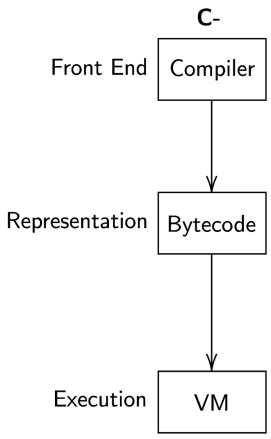


Figure 2: Implementation Process

3.3 LAYING THE FOUNDATION

We'll start with our `main()` function in `main.c`. For now we'll just create an empty function.

```
// main.c

#include "common.h"
```

```
int main(int argc, const char* argv[]) {
  return 0;
}
```

As we can see by the quotation marks, the `common.h` file is not part of the C standard library. Let's create it now:

```
// common.h

#ifndef common_h
#define common_h

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#endif
```

This is where we store the additional C types we'll use throughout the process. For now, we've added `NULL`, the C99 `bool` data type which macros true and false expand to `1` and `0` respectively, and some more specific types of integers such as `uint8_t` (unsigned integers).

In *Crafting Interpreters*, Bob Nystrom refers to chunks as sequences of bytecode, so we draw from that to create the module `chunk.h`. In bytecode, each instruction has a one-byte opcode (operation code), which controls what kind of instruction we're dealing with. We define that here.

```
// chunk.h

#ifndef chunk_h
#define chunk_h

#include "common.h"

typedef enum {
  OP_RETURN,
} OpCode;

#endif
```

The only instruction `OP_RETURN`, will eventually mean "return from the currrent function", but has no use at the moment.

3.3.1 *Dynamic Arrays*

Let's now create the `struct` to hold the data that will be stored alongside the instructions.

```
// chunk.h

typedef struct {
  uint8_t* code;
} Chunk;
```

This just creates a wrapper arround an area of bytes, however because we don't know how big the array of data needs to be before compiling a chunk we need, it needs to be dynamic. Implementing this is simple, we just need to declare an additional two quantities: the number of elements in the array we have allocated (`capacity`) and how many of these are in use (`count`).

```
// chunk.h

typedef struct {
  int count;
  int capacity;
  uint8_t* code;
} Chunk;
```

If `count` is less than `capacity` then adding an element is as simple as allocating the new element to the free space and incrementing count. If there is no spare capacity then our process is as follows:

- Allocate a new array with more capacity and carry over the existing elements from the old array to the new one.
- Store the new capacity and delete the old array
- Update `code` to point to the new array and store the element in the new array
- Increment `count`

Note that through the use of amortized analysis, we can see that as long as we grow the array by a multiple of its current size, when we average out the cost of a sequence of appends, each append only takes $\mathcal{O}(1)$.

Now that our `struct` is ready, we can implement the functions to work with it. Because C doesn't have constructors, we need to declare a function to initialize a new chunk.

```
// chunk.h

void initChunk(Chunk* chunk);
```

Let's implement it by creating a new file, `chunk.c`:

```
// chunk.c

#include <stdlib.h>

#include "chunk.h"

void initChunk(Chunk* chunk) {
  chunk->count = 0;
  chunk->capacity = 0;
  chunk->code = NULL;
}
```

Note that we don't even allocate a raw array to our dynamic array; it starts of completely empty. Let's create a new function to append a byte to the end of a chunk:

```
// chunk.c

void writeChunk(Chunk* chunk, uint8_t byte) {
  if (chunk->capacity < chunk->count + 1) {
    int oldCapacity = chunk->capacity;
    chunk->capacity = GROW_CAPACITY(oldCapacity);
    chunk->code = GROW_ARRAY(uint8_t, chunk->code,
        oldCapacity, chunk->capacity);
  }

  chunk->code[chunk->count] = byte;
  chunk->count++;
}
```

The first thing that we need to do is see if the current array already has the capacity for the new byte, and if it doesn't we grow the array to make room. This will happen every time on the first write, as we've set the original `capacity` to `0`.

To increase the array's size, we first calculate the new capacity and then adjust the array to match it. These lower-level memory operations are now defined in a new module `memory.h`.

```
// memory.h

#ifndef memory_h
#define memory_h

#include "common.h"

#define GROW_CAPACITY(capacity) \
    ((capacity) < 8 ? 8 : (capacity) * 2)

#endif
```

Since we're using it in the process of chunks, we should also include it in `chunk.c` using:

```
// chunk.c

#include "memory.h"
```

This macro calculates a new capacity based on the current capacity, and for optimal performance, it scales relative to the previous size. Typically, we increase it by a factor of two, although 1.5× is another common choice.

We've also accounted for cases where the current capacity is zero. In such instances, we start with eight elements instead of one, reducing unnecessary memory overhead for very small arrays, albeit at the cost of some wasted space. The choice of the number eight was

somewhat arbitrary in this context, as most dynamic array implementations have a similar minimum threshold.

Once the desired capacity is determined, we proceed to create or expand the array to match that size using the `GROW_ARRAY()` function.

```
// memory.h

#define GROW_ARRAY(type, pointer, oldCount, newCount) \
    (type*)reallocate(pointer, sizeof(type) * (oldCount), \
        sizeof(type) * (newCount))

void* reallocate(void* pointer, size_t oldSize, size_t newSize);
```

This macro simplifies a function call to `reallocate()` where the primary work is accomplished. It takes care of determining the size of the array's element type and performing the necessary casting to ensure the resulting `void*` is of the correct type.

The `reallocate()` function serves as the central function for all dynamic memory management within C-. It handles various tasks such as memory allocation, deallocation, and resizing existing allocations. This consolidation of memory operations into a single function becomes crucial later when we introduce a garbage collector responsible for monitoring memory usage.

The two size arguments provided to `reallocate()` dictate the specific operation to be executed:

`oldSize`	`newSize`	Operation
0	Non-zero	Allocate new block
Non-zero	0	Free allocation
Non-zero	Smaller than `oldSize`	Shrink existing allocation
Non-zero	Larger than `oldSize`	Grow existing allocation

Although this looks like a lot of different cases to consider for, the implementation is quite simple. We'll add this to a new file `memory.c`

```
// memory.c

#include <stdlib.h>

#include "memory.h"

void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
  if (newSize == 0) {
    free(pointer);
    return NULL;
  }

  void* result = realloc(pointer, newSize);
  return result;
}
```

When `newSize` is zero, we handle deallocation ourselves by calling `free()`. Otherwise, we rely on the C standard library's `realloc()` function, which conveniently supports the other three aspects of our memory management policy.

The interesting cases arise when both `oldSize` and `newSize` are non-zero. In such instances, `realloc()` is instructed to resize the previously allocated memory block. If the new size is smaller than the existing block, it updates the block's size and returns the same pointer provided.

If the new size is larger, it attempts to expand the existing block, but this is contingent on there being available memory space beyond it. It can only do this if the memory following the block is not already in use. If there isn't enough room to grow the block, `realloc()` takes an alternative approach: it allocates a new memory block of the desired size, copies the old data into it, frees the old block, and then returns a pointer to the new block.

This behavior aligns perfectly with our dynamic array requirements. However, it's crucial to acknowledge that due to the finite nature of computer resources, allocation can fail if there's insufficient memory. In such cases, `realloc()` will return `NULL`, meaning we should add an extra fallback for this case:

```
// memory.c

void* result = realloc(pointer, newSize);
  if (result == NULL) exit(1);
  return result;
```

We still can't do anything very useful with the VM without it getting it the memory it needs, but at least we can detect that and abort the process instead of just returning a `NULL` pointer and it causing problems later.

Now we've found a way to create new chunks and write instructions to them. Because we're creating this in C, memory management is important and we need to find a way to free chunks as well. We declare it first:

```
// memory.h

void freeChunk(Chunk* chunk);
```

And implement it like so:

```
// memory.c

void freeChunk(Chunk* chunk) {
  FREE_ARRAY(uint8_t, chunk->code, chunk->capacity);
  initChunk(chunk);
}
```

This works by deallocating all the memory and then calling `initChunk()` to zero out the fields leaving the chunk in a (well defined) empty

space. To free the memory completely, we need to add another macro to the header file.

```
// memory.h

#define FREE_ARRAY(type, pointer, oldCount) \
    reallocate(pointer, sizeof(type) * (oldCount), 0)
```

Like the other macro, `GROW_ARRAY()`, this is a wrapper around a `reallocate()` call which frees the memory by passing in zero for the new size.

3.3.2 *Disassembling Chunks*

Let's try our work by trying to hand-building a sample chunk. We'll test it through `main.c`.

```
// main.c

#include "common.h"
#include "chunk.h"

int main(int argc, const char* argv[]) {
  Chunk chunk;
  initChunk(&chunk);
  writeChunk(&chunk, OP_RETURN);
  freeChunk(&chunk);
  return 0;
```

We can run this and it doesn't return any errors, but we can't exactly confirm if our tests worked. This makes sense since all we've done it shift around bytes in memory and there's no human friendly way to see what we've created. To resolve this we're going to create a disassembler.

An assembler, in its traditional form, is a program designed to process a file containing human-readable mnemonic names for CPU instructions, such as `ADD` and `MULT`, and convert them into their corresponding binary machine code representations. Conversely, a disassembler performs the reverse operation: when provided with a chunk of machine code, it generates a textual listing of the instructions contained within.

Let's pass the chunk in `main.c` through the disassembler after creation:

```
// main.c

disassembleChunk(&chunk, "test chunk");
```

We should create a debug header for this as well:

```
// debug.h

#ifndef debug_h
```

```
#define debug_h

#include "chunk.h"

void disassembleChunk(Chunk* chunk, const char* name);
int disassembleInstruction(Chunk* chunk, int offset);

#endif
```

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/