CS 4974 Independent Study: Scripting language design and implementation

BUG LEE, Virginia Tech, USA

CCS Concepts: • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: Compiler, Assembler, Virtual Machine

[1]

1 INTRODUCTION

Over the years, the complexity of compiler design has immensely increased. At the hardware level, there are more diverse and sophisticated instruction sets to consider. At the software level, complex optimization schemes have been introduced for maximal performance, in addition to different language paradigms. As a result, the complexity of the modern compiler design overshadows the fundamentals behind the language construction as well as making the steps between programming language and execution more mysterious. For this reason, this project focused on two aspects: (1) building a simplified version of each component used in programming language construction, and (2) understanding how each component works together to execute a custom programming language. This report describes the semester-long project of building a C-like (C subset) programming language from scratch.

2 PROJECT OVERVIEW

The project was broken down into 4 parts, ordered from lower level to higher level: Virtual Machine, Assembler, Compiler, and Fuzzer. Each level depends on the level right below it, except for Virtual Machine which is a standalone software.

2.1 Virtual Machine

The Virtual Machine mimics the generic single-cycle hardware processor. The following describes the similarity:

- Load instructions from the executable to the instruction cache.
- Load and store a value into runtime stack and keep track of stack frame using stack pointer and frame pointer.
- Use the return register to access the return value from a function.
- Use the instruction pointer to read and execute the next instruction.
- Execute one instruction in a single cycle.

However, since the Virtual Machine was implemented using C++ instead of transistors, it was flexible for more functionalities that are not available in generic hardware:

- Include the function cache that load all the function information from the executable.
- Perform type resolution/casting/coercion during runtime.

The Virtual Machine support 36 instructions (see appendix). The instruction set for Virtual Machine was designed to follow the Complex Instruction Set Computing (CISC) methodology. This is to make the runtime environment faster: doing as much work as possible in C++ instead of leaving implementation to a slower custom language.

Overall, the role of the Virtual Machine is to set the first point of simplification. Even with the simpler instruction set, however, writing binary executables by hand would be a painful task. Which lead to the implementation of the assembler.

Note that the Virtual Machine design was adapted from the Varanese's XVM[2]. However, unlike XVM, it was simplified to single-threaded and stand-alone software. Also, new instructions were added to support pass-by-reference and print functionality.

2.2 Assembler

Like other assemblers for specific hardware processors, the main functionality is to allow users to use mnemonics and numeric operands, which then get translated to binary executables. However, the custom assembler supports more advanced features and takes advantage of flexibility from the Virtual Machine:

- Differentiate function and label. Use func directive and curly braces to define function. Automatically add ret instruction at the end of the function after assembling.
- Use Var and Param directives to define variables or arrays and automatically reserve space inside the stack.
- Support basic scope resolution.
- Direct support for string type.
- Allow instruction to accept a string, variable, and array (absolute or variable index). Note that type checking is done by a Virtual Machine during the runtime.

However, the limitation of expressivity and tedium of controlling the runtime stack still make assembly language difficult to work with and error-prone. The next subsection, compiler, improve on this issue.

Note that the Assembler design was adapted from the Varanese's XASM[2]. However, there are two major differences compared to XASM. First, the lexer for the custom assembler was implemented using regular expression and state machine whereas XASM used a brute force approach. Secondly, although both the custom assembler and XASM implement a top-down parser, the parser for the custom assembler resembles more closely to recursive descent than XASM.

2.3 Compiler

3 VIRTUAL MACHINE

3.1 Runtime Stack

The Virtual Machine simulates the runtime stack using the linked list. Each stack position can hold any value type (integer, float, stack index, table index, and register code) and increment by 1, resembling word addressable memory. All values were designed to fit inside a single stack position, simplifying stack access/management. String processing instructions are available when a character must be accessed/modified inside the string

3.2 Execution Cycle

The execution cycle begins by first loading the entry point from the executable. Once load the entry point, the Virtual Machine takes the following steps for each cycle, which resemble the execution cycle of the hardware processor:

- Fetch instruction
  Read the instruction that the instruction pointer is pointing at.

- Decode opcode
  The type of instruction is determined by first reading the opcode. Then, the Virtual Machine calls the appropriate function by index into an array of function pointers.
- Resolve operand
  The type of operand must be resolved in this stage. As the goal is to support the typeless language, the source operand gets locally cast to the destination operand type.
- Execute instruction
  Once the operands are resolved locally, the Virtual Machine executes the instruction's logic.
- Write back
  For many instructions, the destination operand (stack index or return value register) gets updated to the result of the execution stage.

3.3 Flexibility VS Performance

The simpler instruction set comes with a cost. The Virtual Machine itself is software that needs to translate virtual instruction to real instruction during execution. Therefore, multiple real instructions are generated under the hood to execute one virtual instruction. This was one tradeoff between flexibility and performance.

4 ASSEMBLER

Although adding the assembler between the compiler and virtual machine was not necessary, the assembler was added for two reasons: (1) to gain more experience in applying compiler theory with a simpler problem domain before implementing the compiler, and (2) to simplify the code generation step for the compiler. The assembler was divided into 3 parts in the following order: lexer, parser, and code generator.

4.1 Lexer

The lexer was the starting point of the assembler implementation, where it transforms the input character stream from the text file to the lexeme/token stream needed for the parser. The lexical analysis process was implemented using a state machine, where the following regular expressions define rules for grouping characters into a lexeme.

4.2 Parser

The parsing process was designed to complete in 2 passes. In the first pass, the parser record labels, identifiers, and function into corresponding tables. Instructions are ignored in the first pass. In the second pass, the parser evaluates the instruction. When an instruction hold operand of type labels, identifiers, or function, the operand gets replaced with the numeric value (either instruction address for labels/function or stack address for identifiers). 2 pass parsing was needed to address forward referencing.

The parser follows the top-down approach, resembling recursive descent parsing. However, due to the grammar structure of the assembly language, identifying the first token on the line was enough to determine the syntax and semantics of the line. When the first token is directive, the parser follows the grammar rules and recurses to the appropriate function. However, for most cases when the first token is an opcode, syntax and semantic analysis are simple as matching the instruction template of the read opcode from the instruction lookup table.

In the end, the parser output a linked list of instructions, string, and function tables that are used by the code generator to output binary executable.

4.3 Code generator

The code generator uses the linked list of instructions, string, and function tables outputted by the parser to generate a binary executable. The format of the binary executable is shown below.

5 COMPILER

6 TEST STRATEGY

7 TEST RESULTS

8 LIMITATION

9 FUTURE WORK

ACKNOWLEDGMENTS

To Dr.Gulzar, for supervising the independent study and providing guidance.

REFERENCES

[1] Robert Nystrom. 2021. *Crafting Interpreters.* Genever Benning.
[2] Alex Varanese. 2002. *Game Scripting Mastery.* Course Technology PTR.

A INSTRUCTION SET

A.1 Memory

Instruction	Opcode	Operand count	Operands
mov	0	2	DESTINATION, SOURCE

A.2 Arithmetic

Instruction	Opcode	Operand count	Operands
add	1	2	DESTINATION, SOURCE
sub	2	2	DESTINATION, SOURCE
mul	3	2	DESTINATION, SOURCE
div	4	2	DESTINATION, SOURCE
mod	5	2	DESTINATION, SOURCE
exp	6	2	DESTINATION, SOURCE
neg	7	1	DESTINATION
inc	8	1	DESTINATION
dec	9	1	DESTINATION

A.3 Bitwise

A.4 String Processing

A.5 Conditional Branching

A.6 Stack Interface

A.7 Function Interface

A.8 Directives