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 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 3 parts, ordered from lower level to higher level: Virtual Machine, Assembler, Compiler. 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, there are three major differences. First, the custom VM was coded in C++ instead of C. Secondly, unlike XVM, it was simplified to single-threaded and stand-alone software. Finally, 6 instructions for string concatenation and conditional jumps support in XVM were removed in the custom VM. Instead, 12 new instructions were added to the custom VM.

2.2 Assembler

Like generic assemblers, 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 functional scope.
- Direct support for string type.
- Support for basic string processing.
- Allow instruction to accept a string, variable, and element inside an array (absolute or variable index). Note that type checking is done by the Virtual Machine during the runtime.

However, the limitation of expressivity and tedium of controlling the runtime stack still made assembly language difficult to program with and error-prone. The compiler improves on this issue.

Note that the Assembler design was adapted from the Varanese's XASM[2]. However, there are three major differences compared to XASM. First, the custom assembler was coded in C++ instead of C. Secondly, the lexer for the custom assembler was implemented using regular expression and state machine whereas XASM used a brute force approach. Finally, the parser for the custom assembler resembles more closely to recursive descent more than the brute force parser for XASM.

2.3 Compiler

The custom compiler was designed for C-like custom language, where the goal was to translate source code into assembly code targeted for the custom virtual machine. The compiler provides a subset of functionality that what C compiler can do, including:

- Preprocess line and block comments
- Assignment, arithmetic, relational, and logical operations
- Static scoping using block
- Support for array
- Conditional statements
- Loops and break/continue statements
- User-defined functions
- Pass by value and pass by pointer
- Support for native functions like time, random, print, and exit

On the other hand, the custom compiler also adds additional features:

- Typeless language

- Array holding multiple different types of elements
- Direct support for string type
- Support for basic string processing

The compiler showcased the insight and science behind the introduction of programming languages. As the secret behind the magic black box was revealed, it was incredible to observe how a complex program can be translated into a handful of simple instructions.

Note that the compiler design was adapted from Nystrom JLox interpreter[1]. However, there are four major differences. First, the custom compiler was coded in C++ instead of Java. Secondly, the custom compiler emits assembly code as an output whereas JLox interprets each statement on the spot. Thirdly, the custom compiler supports both variables and arrays whereas JLox only supports variables. Finally, the custom compiler supports both pass-by-value and pass-by-reference whereas JLox only supports pass-by-value.

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.

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 came with a cost. The Virtual Machine itself is software that needs to translate virtual instruction to real instruction during execution. Therefore, multiple real instructions were 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 grammar of the potential instruction with the grammar rule cached inside 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

The compiler was divided into 5 parts in the following order: preprocessor, lexer, parser, semantic analyzer, and code emitter. The compiler reuse most of the lexer design from the assembler, but comment handling was delegated to the preprocessor. On the other hand, parser design needed a different strategy as the complexity of the grammar rules was more demanding. So, unlike the assembler parser, the compiler parser only takes care of syntax analysis and generates an almost ready-to-use Abstract syntax tree (AST) as an output. Then, the semantic analyzer performs semantic analysis and completes the AST. As the last step, the code emitter visits the completed AST in post-order traversal and generates assembly code corresponding to the visiting AST node.

5.1 Preprocessor

To simplify lexer design, the preprocessor was designed to handle line and block comments. For line comments, the preprocessor replaces any character starting from // to a newline character with white space. Similarly, for block comments, the preprocessor replaces any character starting from /* until the end of block comment */.

5.2 Lexer

Like the assembler lexer, it takes a character stream from the source text file and outputs the lexeme/token stream needed for the parser. The same strategy from the assembler lexer, building a state machine where state transitions are described with regular expression, was used for lexical analysis. The difference is that number of unique tokens that the compiler lexer needs to support is much greater than the assembler lexer.

5.3 Parser

There are two main functionalities of the parser: (1) perform syntax analysis and (2) construct AST that represents source code in a more structured form, but not quite ready for code emitter. Like assembler, identifying the first token in the statement was enough to determine the type of statement that the parser was parsing. The following BNF describes the grammar rules of the custom language.

Head		Body
program	→	declaration* EOF
declaration	→	func
		| var
		| statement ;
func	→	"func" function ;
var	→	"var" IDENT ("[" INT "]")? ("=" expression)? ";" ;
statement	→	exprStmt
		| forStmt
		| ifStmt
		| printStmt
		| returnStmt
		| whileStmt
		| gotoStmt
		| block ;
exprStmt	→	expression ";" ;
forStmt	→	"for" "(" (var | exprStmt | ";") expression? ";" expressions? ")" statement ;
ifStmt	→	"if" "(" expression ")" statement
		("else" statement)? ;
printStmt	→	"print" expression ";" ;
returnStmt	→	"return" expression? ";" ;
whileStmt	→	"while" "(" expression ")" statement ;
gotoStmt	→	("break" | "continue") ";" ;
block	→	"{" declaration* "}" ;

However, unlike the assembler parser, now the statements can be formed with expressions instead of all terminals. So, knowing the first token in the statement was no longer enough to disambiguate what remaining tokens must be within the statement. The precedence and associativity also added the complexity of disambiguating expressions to correct grammar rules. As a solution, the parser was designed to use the recursive descent technique. The following BNF describes the expression grammar rules of the custom language.

Head		Body
expression	→	assignment ;
assignment	→	(IDENT | ARRAY) "=" assignment
		| logic-or ;
logic-or	→	logic-and ("| |" logic-and)* ;
logic-and	→	equality ("&&" equality)* ;
equality	→	comparison (("!=" | "==") comparison)* ;
comparison	→	term ((">" | ">=" | "<" | "<=") term)* ;
term	→	factor (("-" | "+") factor)* ;
factor	→	unary (("/" | "*") unary)* ;
unary	→	("!" | "-") unary | call ;
call	→	ref "(" arguments? ")" ;
ref	→	"&" IDENT | primary ;
primary	→	"true" | "false"
		| INT | FLOAT | STRING | IDENT | ARRAY | "(" expression ")" ;

The precedence is determined from the bottom to the top in the above rules since the recursive descent technique recurses down until it reaches the leaf node of the AST. The associativity is determined by the placement of an operator. Placing before the recursive production like unary makes the rule right-associative whereas placing after the recursive production makes the rule left-associative.

The above grammar rules were translated directly to the C++ functions using the following strategy:

Grammar notation	Code representation
Terminal	Token to match
Nonterminal	Function call to corresponding rule
|	**if** or **switch** statement
* or +	**while** or **for** loop
?	**if** statement

When the parser parses through the source file, it simultaneously builds the AST. Each rule adds corresponding node representation to AST, shaping the AST exactly the same as how the file was parsed. In the end, the parser output an AST that describes the source code using the grammar rules shown above, almost ready for use by the code emitter. The semantic analyzer fills in the remaining gap.

5.4 Semantic analyzer

The semantic analyzer also has two main tasks: (1) finishing up AST construction started in the parser and (2) semantic analysis. Like the parser, both actions happen simultaneously. The semantic analyzer used the visitor pattern to walk through the AST statement by statement, visiting AST nodes in the post-order traversal.

First, the semantic analyzer completes the AST construction by taking care of static scope resolution and dereferencing that could not be handled during the parsing phase. Static scope resolution allows custom language to declare the local variable with a name that was already declared from another scope. This was implemented by shadowing the symbol table described in the following steps:

(1) When the semantic analyzer visits the new block, it assigns a new symbol table to be used for the new scope, pushing behind the previous symbol table as a backup.
(2) Then, when the semantic analyzer verifies the scope of a variable, it always checks the innermost symbol table first. Move onto the symbol table next in line if the corresponding variable name was not found.
(3) Semantic analyzer record the variable scope (symbol table ID) inside the AST, which will be used by the code emitter later.
(4) Once the semantic analyzer finishes visiting the block, it removes the assigned symbol table for the block and restores the symbol table to the previous value.

On the other hand, dereferencing was needed to allow the pass-by-reference feature. Similar to C, the star "*" in front of the parameter indicates that the parameter is a pointer instead of a value and is recorded as a reference when added to the symbol table. When the semantic analyzer records the scope of a variable, it also checks if the variable is a reference. Dereferencing flag is set to true if the visiting AST node is the array index. This completes the AST construction to be used for the code emitter.

Now, for the semantic analysis, it is responsible for checking name conflict, validating the number of argument passing and arity of functions, and catching any incorrect usage of variables/arrays/functions. The semantic analyzer was designed to disallow name conflict between functions and variables to minimize the coding mistake caused by misusing the function as a variable or vice versa. The verification for name conflict is simple as whether the name was previously defined as a different type by looking up both function and symbol tables. The other verifications also use table lookup and follow similar steps.

5.5 Code emitter

The code emitter generates assembly code targeted for the custom virtual machine. Like a semantic analyzer, it uses the visitor design pattern to visit each AST node in the post-order traversal. For simpler implementation, the code emitter was designed to follow the stack machine model. That is, it heavily utilizes stack to store intermediate results for each instruction. However, to make the generated code compatible with the custom assembler, three registers, _t0, _t1, and a return value register were used in addition to the stack.

In the case of emitting code for expressions, the code emitter needed explicitly write the corresponding operation in assembly. For example, writing the binary expression takes the following steps:

(1) Pop the operands from the top of the stack to _t0
(2) Pop the operands from the top of the stack to _t1
(3) Perform operation on _t0 with _t1
(4) Push the _t0 to the top of the stack

On the other hand, emitting code for statements resembled more closely to the high-level counterpart. Most of the work was to think about what part of the code should be emitted first and where should be placed, whereas emitting process was simple as letting the code emitter recursively do the work. For example, writing a while statement takes the following steps. Note that the code for continue/break was omitted for a clearer view.

(1) Start label
(2) Let code emitter emit conditional expression
(3) Pop the top of the stack value to _t0
(4) Jump to End label if _t0 is false
(5) Let code emitter emit body statement
(6) Jump to Start label
(7) End label

6 TEST STRATEGY

Testing was divided into two category: unit testing and integration testing.

7 BENCHMARK RESULTS

Program	Input size	Custom language	Python
Fibonacci	25	1.3913 seconds	0.0345 seconds
Insertion sort	1000	2.1393 seconds	0.0258 seconds
Merge sort	10000	2.3585 seconds	0.0394 seconds

8 LIMITATION

There are 3 limitations in the custom langague. First, the compiler cannot be extended to support initializing array with variable. The current VM design

The project was under tight time constraint where new concept was introduced and implemented each week. During the design phase, the limitations By the end of the project, some of the limitations of the custom language were observed.

9 FUTURE WORK

The project leave with many possible extension can be added.

9.1 Extending the supporting langauge features

9.2 Optimization

9.3 Fuzzer

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