CS 4974 Scripting Language Design and Implementation

Dr. Muhammad Ali Gulzar
Office: KW II, Suite 2224, 2202 Kraft Drive, Blacksburg, VA 24060
Email: gulzar@cs.vt.edu
Phone: (540) 231-0851

Course Syllabus

1. Credit Hours: 3

2. Description:

Provide an in-depth introduction to principles and implementation of scripting language. The objective is to design and implement a C-like scripting language by the end. Emphasis is given to topics of Assembler, Virtual Machine, and Compiler thorough hands-on implementation. Also, give the student opportunity to dig deeper on a specific topic on VM/Compiler of his choice through an open-ended project.

3. Format:

The course will be entirely project-based, where the student will work on a semester-long project to build a working C-like (C subset) scripting language from scratch. The project is broken down into 4 parts: Assembler, Virtual Machine, Compiler, and Open-ended project. The Assembler, Virtual Machine, and Compiler will be written in C.
In the first several weeks, the student will work on the low-level scripting environment. The student will design an assembly language that mimics a subset of intel 80x86 assembly, implement an assembler for generating bytecode, then create a Virtual Machine that mimics a generic hardware processor to execute the bytecode. After that, the student will implement a compiler for the next several weeks. Finally, for the remaining weeks, the student will select and work on a specific topic that interestingly extends the project (See the open-ended project below).

4. Learning outcomes:

- Obtain a full overview of scripting language construction through studying the interaction/relationship between source code, compiler, assembler, and virtual machine.
- Gain a deeper understanding of computer architecture and operating systems through hands-on implementation of an assembler that generates 80x86-like instructions and a virtual machine that can run it.
- Gain a deeper understanding of compiler theory and the inner working of programming language.
- Understand the advantages and disadvantages of scripting language. Know why execution of scripting language has a lower performance over programming languages that compile and run directly on specific hardware.
- Improve C programming and organization/management of a large-scale software project.

- More exposure to test-driven development, writing own test cases, and regression testing. Learn to write meaningful test cases that catch bugs and ensure the correctness of software.

5. Textbooks:

1. Crafting interpreters by Robert Nystrom
2. Modern Compiler Implementation in ML by Andrew Appel
3. Game scripting mastery by Alex Varanese

6. Assignments:

- Reading assigned chapters from the textbooks (See the tentative schedule below)
- Project report for each project
- Project 1: Assembler
- Project 2: Virtual Machine
- Project 3: Compiler
- **Open-ended final project:**
  - The open-ended project should extend the scripting system in an interesting way. Examples of possible projects include:
    - Implementing Symbolic execution.
    - Implementing fuzzing to test the compiler.
    - Integrating the scripting system into a game engine or embedded application.
    - Implementing priority-based multithreading features into VM.
    - Optimizing compiler output with better stack utilization and use of additional virtual registers.
    - Implementing popular language features such as classes, inheritance, garbage collection, etc.

7. Grading:

Grading is based on correctness and coverage of test cases for implemented features, project reports, and successful demonstration of working scripting language at the end of the semester.

Test cases:

Employ test-driven development by writing a large set of small programs demonstrating one aspect of the assembler, virtual machine, and compiler. These will include both programs that should assemble/compile successfully and ones that should output errors.

Project reports:

Document the justification and trade-offs made in each project. This should demonstrate a strong understanding of the systems that the student has built throughout the semester.

End of semester Demo:

Since this course aims to build a working scripting language, the student should be able to demonstrate a program that is entirely run using the scripting language he has built. The program should demonstrate the capability, versatility, and robustness of the scripting language.

8. Tentative schedule:

Phase 1: Assembler and Virtual Machine

Week #	Topic	Assignment	Reading
1	Instruction set and Assembly framework	Project 1 start	Varanese Chapter 8
2	Assembler for VM		Varanese Chapter 9
3	Bytecode and Executable format	Project 1 due Project 2 start	
4	VM structure interfaces		Varanese Chapter 10
5	VM execution cycle		Varanese Chapter 11

Phase 2: Compiler

Week #	Topic	Assignment	Reading
6	Loader and Preprocessor	Project 2 due Project 3 start	Varanese Chapter 12
7	Lexical Analysis		Appel Chapter 2 Nystrom Chapter 4-5, 16
8	Parsing		Appel Chapter 3, 4 Nystrom Chapter 6-8, 17
9	Semantic Analysis, I-Code		Appel Chapter 5, 7 Nystrom Chapter 11
10	Code Generation		Appel Chapter 8-9

Phase 3: Open-ended project

Week #	Topic	Assignment	Reading
11	Topic for open-ended project	Project 3 due Open-ended final project start	TBD
12	Topic for open-ended project		Fuzzing Book, Sec II and III/ Prelim Symbolic Execution
13	Topic for open-ended project		Fuzzing Book, Sec II and III/Prelim Symbolic Execution
14	Thanksgiving		

	break		
15	Topic for open-ended project		Fuzzing Book, Sec II and III/Prelim Symbolic Execution
16	Wrap-up and Demo	Open-ended final project due	

Instruction Set

The instruction set will follow Complex Instruction Set Computing (CISC) methodology, similar to Intel 80x86. This is to make the scripting system's runtime environment faster: doing as much work as possible in C instead of leaving implementation to a slower scripting language.

Memory

```
mov     Destination, Source
```

Arithmetic

```
add     Destination, Source
sub     Destination, Source
mul     Destination, Source
div     Destination, Source
mod     Destination, Source
exp     Destination, power
neg     Destination
Inc     Destination
Dec     Destination
```

Bitwise

```
and     Destination, Source
or      Destination, Source
xor     Destination, Source
not     Destination
shl     Destination, ShiftCount
shr     Destination, ShiftCount
```

String processing

```
concat      String0, String1
getChar     Destination, Source, Index
SetChar     Index, Destination, Source
```

Conditional Branching

```
jmp     Label
je      Op0, Op1, Label
jne     Op0, Op1, Label
jg      Op0, Op1, Label
jl      Op0, Op1, Label
jge     Op0, Op1, Label
jle     Op0, Op1, Label
```

Stack Interface

```
push    Source
pop     Destination
```

Function Interface

```
call        FunctionName
ret
```

Miscellaneous

```
pause   Duration
exit    Code
```

Directives

```
setStackSize    Size
var             Identifier
func            FunctionName
```

Supporting Operand types

- Integer
- Float
- String
- Variable
- Array with literal index
- Array with variable index
- Line label
- Function name

Scripting language Minimum Design Goal

Data Structures

Typeless variable: support Boolean, integer, floating-point, and string.

```
var myBool = true;
var myInt = 33;
var myFloat = 1.1234;
var myString = "hello world!";
```

Array

```
var myArray [10];
myArray[1] = "hello world!";
```

Operators

Arithmetic

Operator	Description
+	Addition (Binary)
-	Subtraction (Binary)
$	String Concatenation (Binary)
*	Multiplication (Binary)
/	Division (Binary)
%	Modulus (Binary)
^	Exponent (Binary)
++	Increment (Unary)
--	Decrement (Unary)
=	Assignment (Binary)
+=	Addition assignment (Binary)
-=	Subtraction assignment (Binary)
*=	Multiplication assignment (Binary)
/=	Division assignment (Binary)
%=	Modulus assignment (Binary)
^=	Exponent assignment (Binary)

Bitwise

Operator	Description
&	And (Binary)
|	Or (Binary)
#	Xor (Binary)
~	Not (Unary)
<<	Shift left (Binary)
>>	Shift right (Binary)

Logical and Relational

Operator	Description
&&	And (Binary)
||	Or (Binary)
!	Not (Unary)
==	Equal (Binary)
!=	Not Equal (Binary)
<	Less than (Binary)
>	Greater than (Binary)
<=	Less than or equal (Binary)
>=	Greater than or equal (Binary)

Control Structures

```
if (Expression)
{
    // True
}
else
{
    // false
}

while (Expression)
{
    // body
    // break or continue can be used
}

for (Initializer; Terminating condition; Iterator)
{
    // body
```

```
    // break or continue can be used
}
```

Functions

```
func FunctionName (...)
{
    // body
    // (optional) return ...
}
```