University of Glasgow | School of Computing Science

Design and implement your own programming language

Kyle Simpson
Kristiyan Dimitrov
Darren Findlay
David Creigh
Gerard Docherty

Level 3 Project — 27 March 2015

Abstract

The abstract goes here

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

Chapter 2

Language Tutorial

This chapter comprises of a brief introduction of how to use our language.

2.1 Getting Started

In every language, the first program to write is always 'hello world' - where you would print the words "hello world". However, as this language is mainly a graphical language, the equivalent would be 'hello square', and the aim is to draw a square. In MLang, the program required to draw a square is:

```
#EXAMPLE 1.1
Update(){
        Polygon square = ({0,0} + {5,0}) * 4}
        draw(square)}
}
```

to run this program on machines running a UNIX operating system, you must create a file with the the suffix ".m" **Need compiling commands**

Any MLang program you write will have to consist of at least one function, and variables. These functions and variables can be named anything you like. You can call other functions to help carry out the task, only ones that you have written, as there are no libraries provided as such.

In Mlang, implicit semi-colons exist at the end of every line. This means that, you dont have to end each line wih a semi-colon, and whether you do or not will not have an effect on the compilation of the program.

The draw() function is used to pass the shapes you want to draw to the correct function, so it can be written to the cnavas. This will be talked about in detail later.

One way in which you can pass data between functions is by including variables in the calling statement as arguments. However, you can only do this if the function that is being called is expect-

ing the same number and type of variables. An example is:

```
#EXAMPLE 1.2
main(){
        num n = 2;
        takesParams(num n){
                print(n);
        }
        takesParams(n);
}
```

Here, you see that the calling statement - *takesParams(n);* - provides the correct number and type of arguments. If there was another argument included, for example *takesParams(n, 7)*, or the wrong types, then it would not compile. This will be talked about in more detail in 1.7. As you can see in example 1.1, the code inside of a function is enclosed by curly braces . The statement draw takes in one argument - either a Line, Point or Polygon, and draws it to the canvas.

newlines?

2.2 Variables and Arithmetic operators

This next section will use a more complicated program than before, introducing more features, such as comments, loops, and exapnd on previously touched on features, such as variables.

```
#EXAMPLE 1.3
#This Program will draw 3 different shapes - triangle, square and pentagon.
main(){
        num sides = 5;
        num counter = 3;
        Point pt1 = {3,1}
        pt2 = {1,3}
        Line l1 = (pt1 + pt2);
        while(counter <= sides){
                Polygon shape = 1 * counter;
                draw(shape);
                counter++;
        }
}
```

The initial lines indicate how to show comments in your code in MLang. Using a hashsign will be ignored by the compiler, and show that the current line is a comment. These can be used to explain how your programs works, and make it easier to read and understand, for you or other users.

As you can see in example 1.3, there are two ways of declaring variables. A variable declaration must always consist of

implicit/explicit types while loop points lines shapes ++ and arithmetic operators

Chapter 3

Language Reference Manual

Chapter 4

Project Plan

Chapter 5

Language Evolution

Chapter 6

Compiler Architecture

6.1 Overview

Our program module, as part of the role it plays, contains a combination of a compiler, bytecode executor and WebGL interface - written in JavaScript as required by the platform. This section aims to provide a view of the overall architecture and connection between these components, as well a brief explanation of each component's function and design considerations.

The core language module can be divided into 5 main parts:

1. **Lexer and Parser** — The program component charged with tokenising and interpreting MLang programs, generating an abstract syntax tree for use in code generation.
2. **Code Generator** — The program component responsible for conversion of abstract syntax trees into bytecode sequences, for execution by the abstract machine.
3. **Shader Manager** — An active subsystem of the module responsible for management, selection and execution of shader programs for drawing at runtime.
4. **Abstract Machine** — The main active subsystem present in the module, responsible for all runtime code execution. The abstract machine relies on the shader manager to make draw calls, but handles direct WebGL manipulation for certain key functions separately.
5. **Module Facade** — The active component responsible for managing the interconnections and operation of the above components, as well as exposing the module interface to programmers and for use with.

Individual and detailed discussion of each system is provided below. The module's subsystems are then connected as shown in Figure (number).

6.2 Lexer and Parser

6.3 Code Generator

6.4 Shader Manager

In designing our system, it was established that regardless of the chosen compilation and execution pathway all of our library's *shaders* would have to be intelligently managed; both during runtime and for storage purposes inside the module itself.

Shaders are the graphics card level program units used in the conversion from vertex arrays into two-dimensional projections in the screen space, and in the selection of pixel colours to populate the screen space from that projection. In WebGL, in older versions of OpenGL and in its subsets such as GLES, shaders must always come in pairs: a *vertex shader* and *fragment shader* comprise each *shader program*. This may then be accessed, modified and called through the WebGL API.

The unique needs of our module created an interesting set of design requirements for the subsystem - mandating a somewhat in-depth understanding of the WebGL API to manipulate the canvas as required. Upon examination, the shader manager's requirements were found to be:

- **Extensibility** — The language would need to support future extension with various shapes, shaders and other such additions beyond its standard library - most shader programs require writing tailored code in, say, an object's .draw() method.
- **Ease of Use** — Definition of new shader files should be intuitive and an easy process, provided the user has valid shader code. The linking and compilation of shaders and shader programs must be handled internally, and completely sequestered from external control to centralise most of the API manipulation.
- **Configurable** — Shader programs should have variable control methods exposed as an external interface, to provide a simple means of control over the intricacies of the WebGL API.
- **Simplicity** — Interactions with the module should be designed with brevity in mind - being able to execute variable update and draw functions as a single method call makes WebGL usage simpler in the other subsystem implementations.

6.5 Abstract Machine

6.6 Module Facade

Chapter 7

Development Environment

Chapter 8

Test Plan and test suites

Chapter 9

Conclusion

9.1 Contributions

9.1.1 Report

Writing:

- *Kyle Simpson:* Chapter 6.1, 6.4

Editing:

9.1.2 Program

Chapter 10

Appendix A

Includes full source listing of compiler