



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

1	Introduction	4
1.1	Project Outline	4
1.2	Motivation	4
1.3	Aim	4
1.4	WebGL Background	5
1.5	Outline of report	5
2	Language Tutorial	6
2.1	Getting Started	6
2.2	Variables and Arithmetic operators	7
2.3	If statement and comparative operators	9
2.4	Modifying Operators	10
2.5	Functions and Scope	10
3	Language Reference Manual	12
4	Project Plan	13
5	Language Evolution	14
6	Compiler Architecture	15
6.1	Overview	15
6.2	Lexer and Parser	16

6.2.1	Introduction	16
6.2.2	Initial steps	16
6.2.3	Design	16
6.3	Code Generator	17
6.4	Shader Manager	17
6.4.1	Introduction	17
6.4.2	Definitions	17
6.4.3	Requirements	17
6.4.4	Usage	18
6.4.5	Features	19
6.5	Abstract Machine	19
6.6	Module Facade	19
7	Development Environment	20
8	Test Plan and test suites	21
9	Conclusion	22
9.1	Contributions	22
9.1.1	Report	22
9.1.2	Program	22
10	Appendix A	23

Chapter 1

Introduction

1.1 Project Outline

For our project we chose project number 3924; design and implement your own programming language. This project involves designing a programming language, by creating a BNF, system diagram, and the syntax and semantics of the language, among others and then implementing it, using a parser, compiler, and abstract machine.

1.2 Motivation

Our Motivation behind the choosing of this particular project is that, especially in the past few years, computing has come to the forefront of teaching. Many parents now want their children to learn how to code to some degree, with computing science now being a much more prevalent subject in primary and secondary schools. Now in our third year of university, we have all taken part in a fair amount of coding, however, none of us have ever attempted to write our own language to code in. This project seemed challenging, but achievable, and very rewarding not to mention interesting.

1.3 Aim

The aim for this project is to create a graphical programming language that is simple to use and read, and can be picked up by nearly any novice user, but is also powerful, and can be used to create drawings, or edit pictures. Our idea for a simple language came from Processing, a graphical language created by Casey Reas, and the basis for the language is going to be WebGL a browser version of OpenGL. With this project, we want to create a language that follows what we think are useful and intuitive features in the design of most programming languages, for example using `for` functions and loops and keeping the familiar structure of the `for` and `while` loops, and making sure our language follows them, while finding what we believe to be confusing or annoying common features of programming languages, like the double equals for the comparison of two objects, and replace them with what we find suitable and more

1.4 WebGL Background

Web Graphics Library (WebGL) is a JavaScript API, that makes use of the `<canvas>` element available in HTML to render interactive 2d/3d graphics. It is a branch of the Open Graphics Library Embedded Systems 2.0 (OpenGL ES 2.0), and is supported by most modern browsers, like Google Chrome and Mozilla Firefox, with no necessary plugin (as it makes use of `<canvas>`). A WebGL program consists of control code written in JavaScript, and function code (shader code) that is executed by a computer's GPU.

1.5 Outline of report

The remaining sections of this report will cover the following topics:

- Language Tutorial :- This will comprise of a brief introduction to our programming language, which we have named **Sketch**
- Language Reference Manual :- This will contain the grammar, and describe the language
- Project Plan :- This will cover a timeline and basic structure of our project, and what we aimed to achieve from it. **I think so anyway**
- Language Evolution :- This will describe the decisions we made for the language syntax and domain, and why we made them.
- Compiler Architecture :- This section will cover the 5 different sections of the compiler:
 - Lexer and Parser
 - Code Generator
 - Shader Manager
 - Abstract Machine
 - Module Facade
- Development Environment:- The environment(s) that we used to help structure and create this programming language
- Test Plan and Test Suites :- An overview on how we tested the language, using what, and any bugs we discovered
- Conclusions :- This will cover what, as a team, we learned and discovered during this project, any problems we encountered, and anything we would try differently.

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 Sketch, the program required to draw a square is:

```
function init() {  
  Polygon square = ({0,0} + {5,0}) * 4;  
  draw square;  
}
```

Listing 2.1: Hello Square

To compile any Sketch program, insert the code into the browser page for our project and click "Render". The result of the program will then be shown on the canvas.

Any Sketch 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 in the current version of the language.

In Sketch, implicit semi-colons exist at the end of every line. This means that, you don't have to end each line with 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 canvas. This will be discussed in detail later.

One way in which you can pass data between functions is by including variables in the calling state-

ment as arguments. However, you can only do this if the function that is being called is expecting the same number and type of variables. An example is:

```
function init() {  
    int n = 2;  
    function takesParams(int n) {  
        print(n);  
    }  
    takesParams(n);  
}
```

Listing 2.2: Function Parameters

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 Chapter 2.5. As you can see in Listing 2.1, the code inside of a function is enclosed by curly braces {}. The draw keyword is used to draw any shapes that you have constructed to the canvas. It will take one parameter, either the point, line or polygon that you have constructed. This is a predefined keyword in the language, so you don't have to write it, but you cannot name any of your functions *draw()* as a result.

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 expand on previously touched on features, such as variables.

```
#This Program will draw 3 different shapes - triangle, square and pentagon.  
//Both of these lines are ignored by the compiler.  
  
function init(){  
    int sides;  
    sides = 5;  
    int counter = 3;  
  
    Point pt1 = {3,1}  
    pt2 = {1,3}  
  
    Line l1 = (pt1 + pt2);  
  
    #this will loop for the number of sides  
    while(counter <= sides){  
        clear;  
        Polygon shape = l1 * counter;  
        draw shape ;  
        counter++;  
    }  
}
```

Listing 2.3: Comments and Point Addition

The initial lines indicate how to show comments in your code in Sketch. Using a hash sign (#) or a double slash (//) will cause the compiler to ignore all characters until the next newline, 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. The 'clear' function in this example

is a reserved keyword, and is used to clear the canvas. A reserved keyword is a word that cannot be used otherwise, for variable names, of function calls - 'draw' is one such example we have shown.

As you can see in Listing 2.3, there are two ways of declaring variables, explicitly and implicitly. Each variable declaration must include the variable name, which is an alphanumeric string that always begins with a letter. To explicitly define your variable, each variable name must be preceded by the intended type. For example, the variables 'sides' and 'counter' in Listing 2.3 are declared explicitly. Explicit declaration allows for the ability to not initialise the variable immediately. Implicit declaration is when the type is not defined by the user, but instead taken from the context. The type is inferred from the initialization of the variable afterwards. For example, 'pt2' in Listing 2.3 is implicitly defined, and from the context, the compiler will recognise that it is of type Point. However, with implicit types, you must immediately initialise the variable so its type can be inferred.

The while loop in Sketch is of standard format and standard functionality. This means that the condition contained in the parentheses is tested, and if it returns true, then the body of the loop, and whatever statements it may contain, are executed. Then the original condition is retested. If true, it again executes the body. This continues until the condition is false, at which point, the body of the loop is not executed, and the next command outside the loop is executed and the program continues. A while loop has the format:

```
while(condition) {  
    #insert body here  
}
```

Listing 2.4: The "While" Loop

Sketch has three main types for drawing shapes, lines and so forth. These are Point, Line and Polygon. Let's start with Point; Point is the basis of all possible shapes. Points make the vertices on which the shapes will be drawn about. A point is defined in the following format {*x* Coordinate, *y* Coordinate}. An example of both implicit and explicit declaration is shown in Listing 2.3 with 'pt1' and 'pt2'. These points can be added together to create a Line. Lines are a collection of two points joined together. The format for defining a line, shown by 'l1' in Listing 2.3, is Line = (Point1 + Point2). Lines can be extended from the centre point by multiplying it by a number. For example, multiplying a line by 4 will make it 4 times the size. Finally, variables of type Polygon are a collection of lines, displayed as a closed circuit. Polygons can be defined by either adding multiple lines together, but they must all join, or by defining one line, and multiplying it by the number of sides you would like. This would create a shape of the stated number of lines, closed off, with the origin being the first co-ordinate of the line. The format of this would be either Polygon = (Line1 + Line2 ... + LineN) or Polygon = Line * N. In these examples, N is the number of sides desired.

The line 'counter++' increments the counter, using the modifying operator '++'. We will explain modifying operators later in this tutorial.

Sketch also has one more type of loop - the "for" loop. The for loop has a completely different structure to the while loop, but has, more or less, the same functionality. The choice between these two types is based on the context, which one would make more sense, for example, incrementations should be used in for loops, while boolean variables should be the control in while loops. The following is the while loop from Listing 2.3, changed to a for loop:

```

for(int counter = 3; counter < 5; counter ++) {
    ...
}

```

Listing 2.5: The "For" Loop

The for loop statement has three parts. Firstly, the control variable is initialised (it also does not need to be declared beforehand), then the statement that will be checked after every execute of the body, and finally, the incrementation of the control variable. The general functionality is the same as the while loop.

2.3 If statement and comparative operators

This section will cover how to perform comparing statements in Sketch. The following is an if statement which compares to check if a variable is equal to something else:

```

function init() {
    int n = 5;
    if(n == 5) {
        print(n);
    }
    else if(n > 5) {
        print("less than 5");
    }
    else if(n < 5) {
        print("greater than 5");
    }
}

```

Listing 2.6: If-Else Conditionals

If statements are a way of executing a series of commands, but only if a particular statement returns true. To compare to elements, a comparative operator is needed. In Sketch, there exists a series of six comparative operators:

- `==` will return true if left hand side is equal to right hand side.
- `>` will return true if left hand side is greater than right hand side.
- `<` will return true if left hand side is less than right hand side.
- `!=` will return true if left hand side does not equal right hand side.
- `!>` will return true if left hand side is not greater than right hand side (ie is less than or equal to).
- `!<` will return true if left hand side is not less than right hand side (ie is greater than or equal to).

An exclamation mark can be inserted in front of the check statement in the if, to receive the inverse of what it returns.

2.4 Modifying Operators

This section will cover how you can make use of sketch's modifying operators. The following example will outline them:

```
function init() {  
  int n = 1; #line 1  
  n += 2; #line 2  
  n *= 2; #line 3  
  n /= 2; #line 4  
  n -= 2; #line 5  
  n %= 2; #line 6  
  n++; #line 7  
  n--; #line 8  
  n = (n + 3) * 5; #line 9  
}
```

Listing 2.7: Modifying Operators in Action

Listing 2.7 outlines many modifying operators in Sketch. To explain each one, we will walk through the program itself. As previously mentioned, the first line will declare `n` to be of type `int`, with a value of 1. The next line will change the value of `n` by adding one to it. The general syntax would be **variable to be changed* += *how many to change it by**. The next three lines follow the same syntax and semantics, with the exception that, line 3 would multiply the variable by the second value and assign it, line 4 would divide it by the second value and assign it, and line 5 would subtract by the second value and assign it, and finally line 6 would change the variable to equal the value of it modulo the second variable. This means that it would change it to the remainder after being divided by that number. So, after line 2, `n` would be equal to 3, after line 3, `n` would equal 6, after line 4, `n` would equal 3, after line 5 `n` would equal 1 and after line 6, `n` would still equal 1, as the remainder of `1 / 2` is still 1. These operators are functionally a shorter way of writing `n = n + *value*`, but obviously changed respective to the context (division, multiplication, subtraction and modulo).

Line 6 is different. This operator will increment the variable by only 1. In the context of this program, `n` will now equal 2. The following line following the same structure, but instead subtracts one, meaning `n` is back to equal 1. These can be useful in loops for example, to increment or decrement your controlling variable as necessary. The final line outlines how parenthesis work in regards to performing mathematic actions. The flow it would take, is to first, add to `n`, making it equal 4, and then multiply by 5, hence `n` equals 20. The brackets would specify what task has to be carried out first, as if they were not here on this occasion, by the laws of maths, 3 would be multiplied by 5 and then added to `n`, giving a different outcome. All of these modifying operators, except on line 9, are functionally just shorthand, but can be very useful and time saving while writing a program.

2.5 Functions and Scope

In Sketch, functions, as mentioned before, taken certain types and number of parameters dependant on what you specify, it can range from 0 to `n`. Furthermore, these functions can be called whatever you decide, and can either have a return type or not, dependant on your needs. Example 1.6 will show these in action:

```

function init(){
    int x = 1;
    int i = 5;
    function foo(int n) -> int{
        n++;
        return n;
    }
    function foo2(int n){
        n++;
    }
    i = foo(int i);
    foo2(int x);
}

```

Listing 2.8: Functions and Scope

In Listing 2.8, you can see different properties of defining and calling a function. If a function is not returning anything, then you can see that the void is implicit. Furthermore, if a function has a return type, then you can assign a variable to equal its returned value, if it is of the same type. In regards to scope, functions have local scope, meaning that they would alter outside variables, or variables in other functions. This means that, as shown here, you can call variables the same name in two different functions, and there will be no crossover or confusion of values or any conflicts.

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 Sketch 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

6.2.1 Introduction

The lexer and the parser are both part of the syntactic analysis of the compiler. During this stage, a source program is parsed in order to determine if it is well-formed, and to determine its phrase structure, in accordance with the syntax of our programming language. If the source program complies with the syntax and grammar of Sketch, an abstract syntax tree (AST) will be returned from the parser. An AST is a way to represent a source program's phrase structure.

6.2.2 Initial steps

At the beginning of implementing parser and lexer for our grammar, an appropriate parser generator for JavaScript had to be found. We found three possible parser generators, each of which would help us with the desired result. ANTLR, JISON and PEG.js were considered as possible parser generators.

ANTLR uses LL(*) parsing and it can generate lexers, parsers, tree parsers and lexer-parsers. Parsers produced by it are able to automatically generate an AST, which was necessary for our design. However, the most recent release of the software (ANTLR 4), does not support JavaScript as a target source, even though that previous release does.

JISON and PEG.js do not encounter the same problem - they successfully support JavaScript as a target and both could have been appropriate choices for our parser. PEG.js is based on parsing expression grammar formalism, which is more powerful than traditional LL(k) and LR(k) parsers. On the other hand, Jison can recognize languages described by LALR(1) grammars and provides additional features (i.e. the ability to define operator associations and precedence). Because of these additional features, JISON was our choice for parser generator.

6.2.3 Design

JISON provided us with the opportunity to specify both tokenizing rules and language grammar in the same file. This turned out to be very helpful in later stages when the communication between the parser and code generator had to happen. The whole design process is comprised of three stages: lexical analysis, specification of the grammar and specification of the abstract syntax tree in JSON format.

6.2.3.1 Lexer

6.2.3.2 Grammar

6.2.3.3 Abstract Syntax Tree (AST)

6.3 Code Generator

6.4 Shader Manager

6.4.1 Introduction

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. While an extensive system like Palette may seem too broad in scope for Sketch's module in its current form, at the language's inception we wanted custom shapes and shaders to be a defining feature of our platform. As such, Palette allows us to one day realise our initial ambitious vision of Sketch due to the level of extensibility and portability it provides.

6.4.2 Definitions

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*.

6.4.3 Requirements

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** — Sketch needed to support future extension with various shapes, shaders and other such additions beyond its standard library - most shader programs require writing specially tailored code in an object's `.draw()` method. A method of allowing
- **Portability** — Such tailored draw code cannot be imported without putting users at significant security risk by requiring them to execute arbitrary code. Some way to infer the attribute update process from the shader itself is a necessity.

- **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.

6.4.4 Usage

While JSDoc automatically generates the API documentation for Palette, construction of Shader JSON files is not yet documented. As these are an essential part of Sketch's operation, and shader source code will not suffice on its own, a light explanation is provided.

Shader JSON objects may fall into three types:

1. **Vertex Shaders** — Objects following the form of Listing 6.1, with type set to 0 and at least two attributes: a *buffer* named "vertexBuffer" with default value set to the number of components per vertex, and a *vertexAttrib* with a name matching an attribute within the shader and default value set to "vertexBuffer". Further attributes are defined in the same format as fragment shaders.
2. **Fragment Shaders** — Objects defined similarly to Listing 6.2, with type set to 1 and no restriction on what attributes must be predefined.
3. **Shader Lists** — Objects following the structure of Listing 6.3, with type set to 2 and the content tag corresponding to an array of further shader objects of any type.

All integer and vector types are supported, along with their array equivalents. These are denoted as `int`, `int[]`, `float`, `float[]` with vectors taking the form `ivec n` , `ivec n []`, `vec n` and `vec n []` for $n \in [2, 4]$. Float matrices may be defined by `mat n` , for n within the same domain.

```
{
  "type": 0,
  "name": "squareVtx",
  "src": "#version 100; attribute vec3 aVertexAttrib;//...",
  "attrs": [
    ["aVertexAttrib",
     "vertexAttrib",
     "vertexBuffer"],

    ["vertexBuffer",
     "buffer",
     3]
  ]
}
```

Listing 6.1: Vertex Shader JSON Structure

```
{
  "type": 1,
  "name": "squareFrag",
  "src": "//source",
  "attrs": [
    "name",
    "type",
    "default"
  ]
}
```

Listing 6.2: Fragment Shader JSON Structure

```
{
  "type": 2,
  "content": [
    {
      "type": 0
      "name": "...",
      "src": "...",
      "attrs": [...]
    },
    {
      "type": 1
      "name": "...",
      "src": "...",
      "attrs": [...]
    }
  ]
}
```

Listing 6.3: Shader List JSON Structure

6.4.5 Features

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