The BigDuck Programming Language

Jair Antonio Bautista Loranca

November 17, 2021

Contents

I Description and Technical Documentation **3**

1 Project **4**
1.1 Introduction 4
1.1.1 Purpose 4
1.1.2 Scope 4
1.2 Software Requirements 5
1.2.1 Analysis 5
1.2.2 Test Cases 6
1.3 Software Developement Process 6
1.3.1 Developement Process Description 6
1.3.2 Weekly Log 6
1.3.3 Git Commitments 7

2 Language **10**
2.1 General Overview 10
2.1.1 Language Name 10
2.1.2 Main Features Description 10
2.2 Language Errors 11
2.2.1 Compile-Time Errors 11
2.2.2 Run-Time Errors 12

3 Compiler **13**
3.1 Development Environment 13
3.2 Lexical Analisis 13
3.3 Syntactical Analisis 14
3.4 IR Code and Semantic Analisis 16
3.4.1 Operation Code 16
3.4.2 Virtual Addresses 16
3.4.3 Syntax Diagrams 16
3.4.4 Semantic and IR Generation Actions 16
3.4.5 Semantic Consideration Table 16
3.5 Memory Management 16

4 Virtual Machine **17**
4.1 Development Environment 17
4.2 Memory Management 17
4.2.1 Architecture 17
4.2.2 Data Structures 19
4.2.3 Virtual Address Translation 20

5 Execution Evidence **21**
5.1 Test Cases 21
5.1.1 Test Implementation 21
5.1.2 IR Code Output 21
5.1.3 Execution Output 21

6 Code Documentation **22**
6.1 Modules Description 22

II User Manual **23**

7 Quick Reference **24**
7.1 Enviroment 24
7.2 Variables 24
7.3 Statements 24
7.3.1 Assignments 24
7.3.2 Arithmetic Expressions 24
7.3.3 Operator Hierarchy 24
7.4 Conditional Statements 24
7.5 Loop Statements 24
7.5.1 Infinite Loop 24
7.5.2 While Loop 24
7.5.3 For Loop 24
7.5.4 Control Flow Statements 24
7.6 Procedures 24
7.7 Tensorial Types 24
7.8 Built-in Procedures 24

Part I

Description and Technical Documentation

Chapter 1

Project

1.1 Introduction

1.1.1 Purpose

This document describes the software developement process, technical documentation, and user manual, for the final project of the Compiler Design course. Which consists on the design and implementation of a programming language and a virtual machine.

1.1.2 Scope

The programming language developed is specified to be a compiled imperative, with support of modules and structured types. Additionaly it is required to develop a virtual machine capable to execute the output code generated by the compiler.

1.2 Software Requirements

1.2.1 Analysis

Based on the specifications and recomendations given by the teachers, the following requirements were defined as necessary for the successful development of this project.

Functional requirements

1. The programming language must aim to solve a domain specific problem.
2. The compiler must support scoped and global variables.
3. The compiler must support numeric data types.
4. The compiler must support conditional statements.
5. The compiler must support loop statements.
6. The compiler must support modules.
7. The compiler must support recursion.
8. The compiler must support structured types.
9. The compiler must report compile-time errors.
10. The compiler must generate intermidiate code.
11. The virtual machine must execute generated code.
12. The virtual machine must manage program memory.
13. The virtual machine report run-time errors.

Non-Functional requirements

1. The language grammar must be non-ambiguous.
2. The compiler shall use a scanner and parser generation tool.
3. The compiler must be efficient in time and memory.
4. The virtual machine must be efficient in time and memory.

1.2.2 Test Cases

1.3 Software Developement Process

1.3.1 Developement Process Description

The project was developed throught weekly sprints, were each sprint consisted in developing a major feature needed for the programming language compilation or execution. It must be said that despite having an suggested schedule, the reality is that the project went a little bit different from this schedule. This is because some features were prioritize to be implemented first.

1.3.2 Weekly Log

Date	**Description**
Sep 20	Proposal Developement
Sep 27	Lexic and syntax analysis
Oct 4	Symbol table and sematic cube
Oct 11	Expressions compilation
Oct 18	Conditionals compilation
Oct 25	Loops compilation
Nov 1	Procedures compilation
Nov 8	Semantic analysis, memory layout, and virtual machine
Nov 15	Structured types compilation, and application specific code

1.3.3 Git Commitments

Note Not all commits are included in this table because some of them are not directly related with the project (like `README.md` or `.gitignore` updates).

Date	Title	Observations
Sep 27	First Commit	The work done throught the previous HWs were really useful to getting to know ANTLR and make easier the developement.
Sep 29	Parser and Lexer working	It was necessary to make some changes on the grammar to make easier and more concise the implementation.
Oct 5	Advance in semantic analysis	The implementation of a symbol table can be somewhat easy if the concept of symbol is well defined.
Oct 8	Semantic for variables	Despite having a symbol table it is necessary to have additional flags to keep track the context of variables in order to have correct identification of variables.
Oct 9	Semantic for procedures and arguments	Some changes in the syntax were done, since it was not clear enough for the compiler and also for myself.
Oct 9	Variable and expression semantic done	Despite having a symbol table it is necessary to have additional flags to keep track the context of variables in order to have correct identification of variables.
Oct 14	TAC generation	Three address code can be simple to be generated by hand, however to implement it has to be done carefully.
Oct 16	Bugs corrected	There were associativity problems, this was due that all performed actions were done one level deeper on the syntax tree generation, thus as seen in class the associativity was done right to left.

Date	Title	Observations
Oct 25	Conditional and infinite loops implemented	It is important to keep track of loop jumps, otherwise there can be infinte loops on execution.
Oct 28	For style loop implemented	Looking at the generated code I can tell there are optimizations that can be done, however they are probably not done while generating the IR code, since sometimes context is needed to perform such optimizations.
Oct 28	Skip and break implemented	Despite having a jump stack and similar structures to handle the nesting of conditions and loops, some other structures like queues are necesarry to solve this control flow statements.
Nov 6	Procedures implemented	Procedure calls are fairly easy to understand however the details required for them to work in a virtual machine are still needed to be solved.
Nov 7	Semantics for expressions implemented	Despite having worked on the semantic cube, it was not used since it was not a priority. However now that I am seeking to working on the memory layout, having this validation will make it much more easy and reliable to implement.
Nov 8	Semantics for procedures implemented	The memory mapper is great stategy to create the context independant variables for each procedure.
Nov 9	Parameters semantics implemented	Way back to the implementation of the symbol table, I had already thought on having information regarding parameter types, thus the compiler had almost implemented this check.
Nov 9	Return type semantics implemented	Way back to the implementation of the symbol table, I had already thought on having information regarding return types, thus the compiler had almost implemented this check.

Date	Title	Observations
Nov 10	Variable-Address mapping implemented	Implementing the variable-address mapping right on expression compilation has the benefit to be more memory efficient, since only used variables are included on the memory count.
Nov 11	`.quack` file generation	Once the IR code was generated it was only necessary to append it to some instructions to initialize global memory.
Nov 14	Reading of `.quack` files and global memory initialization	The `.quack` files were change to only be a single line string of opcodes and addresses to simplify the reading. This files are meant for computer readability, not for humans.
Nov 14	Era, Goproc, Bool Operators implemented	I decided to implement this operators first because they are somewhat direct.
Nov 15	Basic language features implemented	The implementation of arithmetic operations on the virtual machine is long, boring, and repetitive, but easy. On the other side it is really interesting to see code execution.
Nov 15	Print implemented	Nothing to be said, just the implementation of the print instruction on the virtual machine.
Nov 15	Procedure calls implemented	This was a really interesting problem to solve since it involve on the implementation of a memory stack to handle procedure calls.
Nov 16	Procedure fully working in vm	I was originally stuck since there was no direct solution for this, however, after some thought I was able to implement a parameter buffer to then assign the correspondant values on the context of the function.

Chapter 2

Language

2.1 General Overview

2.1.1 Language Name

The for the programming language was given as a small joke, one of the homeworks on the semester was to develop a scanner and parser for a small language called LittleDuck. Therefore BigDuck could be considered as the next step for the previous mentioned language, even though there is no similarities but the name between these languages.

Additionaly to this, I really like birds and use them as a naming scheme for my devices, thus the decision seemed natural and adecuate.

2.1.2 Main Features Description

BigDuck is language aimed for the developement of mathematical models commonly used in Machine-Learning and Data Science. Therefore this language includes integer and floating point arithmetic, vector and matrix operations, and some basic utilities for reading and writing `.csv` iles. All this to make it easier for the user to work within the Machine-Learning and Data Science fields.

2.2 Language Errors

2.2.1 Compile-Time Errors

BigDuck is language aimed for the developement of mathematical models commonly used in Machine-Learning and Data Science. Therefore this language includes integer and floating point arithmetic, vector and matrix operations, and some basic utilities for reading and writing `.csv` iles. All this to make it easier for the user to work within the Machine-Learning and Data Science fields.

Error message	**Description**
Duplicate symbol	This occurs when the current symbol is already used on the declared scope.
Variable was not declared	This occurs when the current variable has not been previously declared.
Procedure was not declared	This occurs when current procedure has not been previously declared.
Void procedure used in expression	This occurs when there is no return value on the procedure call in an expression.
Procedure expected n arguments, given m	This occurs when the procedure call was given m arguments but it does not match with expectected n atributes specified on declaration.
Type error mismatch	This occurs when an operation can not be performed with the given operands.
Expected boolean expression	This occurs when an expression inside a condition (if's or loop's) does not evaluated to a boolean value.
Parameter expected to be a, given b	This occurs when the parameter of type b does not match with type a expected by the procedure.
Return type different from procedure sign	This occurs when the returned value does not match with the return type expected.

2.2.2 Run-Time Errors

Error message	**Description**
Local address used in data segment	This occurs when it is attempted to initialize a local address before having a local context setup.
Invalid address used in data segment	This occurs when it is attempted to initialize an address that does not conform to with address specification.
Unexpected operator at data segment	This occurs when an operator is used on a segment it was not supposed to be.
Type error mismatch	This occurs when an operation can not be performed with the given operands.

Chapter 3

Compiler

3.1 Development Environment

The BigDuck compiler will be developed using the Go programming language. Antlr4 will be used as lexer and parser generator. And it will be developed on MacOS, any other system support is not considered. Nevertheless with access to a Go compiler and ANTLR, it should be possible to run the BigDuck compiler however this has not been tested.

3.2 Lexical Analisis

Reserved Keywords

`proc`	`return`	`if`	`else`
`loop`	`break`	`skip`	`and`
`or`	`not`	`var`	`int`
`float`	`bool`	`true`	`false`

Tokens

```
     DIGIT → [0-9]
    DIGITS → digit+
    LETTER → [A-Za-z]
      SIGN → ‘‘ - ’’
   CTE_INT → sign?  digits
 CTE_FLOAT → sign?  digits (\. digits)?
        ID → letter (letter | digit | ‘‘ _ ’’)*
   COMMENT → ‘‘ #| ’’ .*?  ‘‘ |# ’’ *
```

3.3 Syntactical Analisis

$$
\begin{aligned}
\texttt{program} &\rightarrow \texttt{vars\_decl procs\_decl} \\
\\
\texttt{vars\_decl} &\rightarrow \texttt{var\_decl var\_decl} \\
&\mid \epsilon \\
\texttt{var\_decl} &\rightarrow \texttt{VAR ID next\_var var\_type `` ; '' next\_var\_decl} \\
\texttt{next\_var} &\rightarrow \texttt{`` , '' ID next\_var} \\
&\mid \epsilon \\
\texttt{next\_var\_decl} &\rightarrow \texttt{var\_decl next\_var\_decl} \\
&\mid \epsilon \\
\\
\texttt{var\_type} &\rightarrow \texttt{scalar | tensor} \\
\\
\texttt{scalar} &\rightarrow \texttt{INT | FLOAT | BOOL} \\
\\
\texttt{tensor} &\rightarrow \texttt{dimension scalar} \\
\texttt{dimension} &\rightarrow \texttt{`` [ '' num\_expr ``] next\_dimension ''} \\
\texttt{next\_dimension} &\rightarrow \texttt{dimension next\_dimension} \\
&\mid \epsilon \\
\\
\texttt{procs\_decl} &\rightarrow \texttt{proc\_decl procs\_decl} \\
&\mid \epsilon \\
\\
\texttt{proc\_decl} &\rightarrow \texttt{PROC ID proc\_args ret\_type local\_decl block} \\
\\
\texttt{proc\_args} &\rightarrow \texttt{`` ( '' ``) ''} \\
&\mid \texttt{`` ( '' ID next\_args scalar next\_types ``) ''} \\
\texttt{next\_args} &\rightarrow \texttt{`` , '' ID next\_args} \\
&\mid \epsilon \\
\texttt{next\_types} &\rightarrow \texttt{`` ; '' ID next\_args scalar next\_types} \\
&\mid \epsilon \\
\\
\texttt{ret\_type} &\rightarrow \texttt{`` -> '' scalar}
\end{aligned}
$$

```
bool_expr → and_expr next_bool
next_bool → OR bool_expr
          | ε

 and_expr → not_expr next_and
 next_and → AND bool_expr
          | ε

 not_expr → (NOT | ε ) bool_term
bool_term → “ ( ” bool_expr “ ) ”
          | rel_expr
          | TRUE
          | FALSE
          | ID (dimension | ε )
          | proc_call

 rel_expr → num_expr rel_op num_expr
   rel_op → “ = ”
          | “ /= ”
          | “ < ”
          | “ > ”
          | “ >= ”
          | “ <= ”

 num_expr → prod_expr next_sum
 next_sum → ( “ + ” | “ - ”) num_expr
          | ε

prod_expr → factor next_prod
next_prod → (“ * ” | “ / ”) prod_expr
          | ε
```

factor → " (" num_expr ") "
| CTE_INT
| CTE_FLOAT
| ID (dimension | ϵ)
| proc_call

3.4 IR Code and Semantic Analisis

3.4.1 Operation Code

3.4.2 Virtual Addresses

3.4.3 Syntax Diagrams

3.4.4 Semantic and IR Generation Actions

3.4.5 Semantic Consideration Table

3.5 Memory Management

Chapter 4

Virtual Machine

4.1 Development Environment

The BigDuck compiler will be developed using the Go programming language. Antlr4 will be used as lexer and parser generator. And it will be developed on MacOS, any other system support is not considered. Nevertheless with access to a Go compiler and ANTLR, it should be possible to run the BigDuck compiler however this has not been tested.

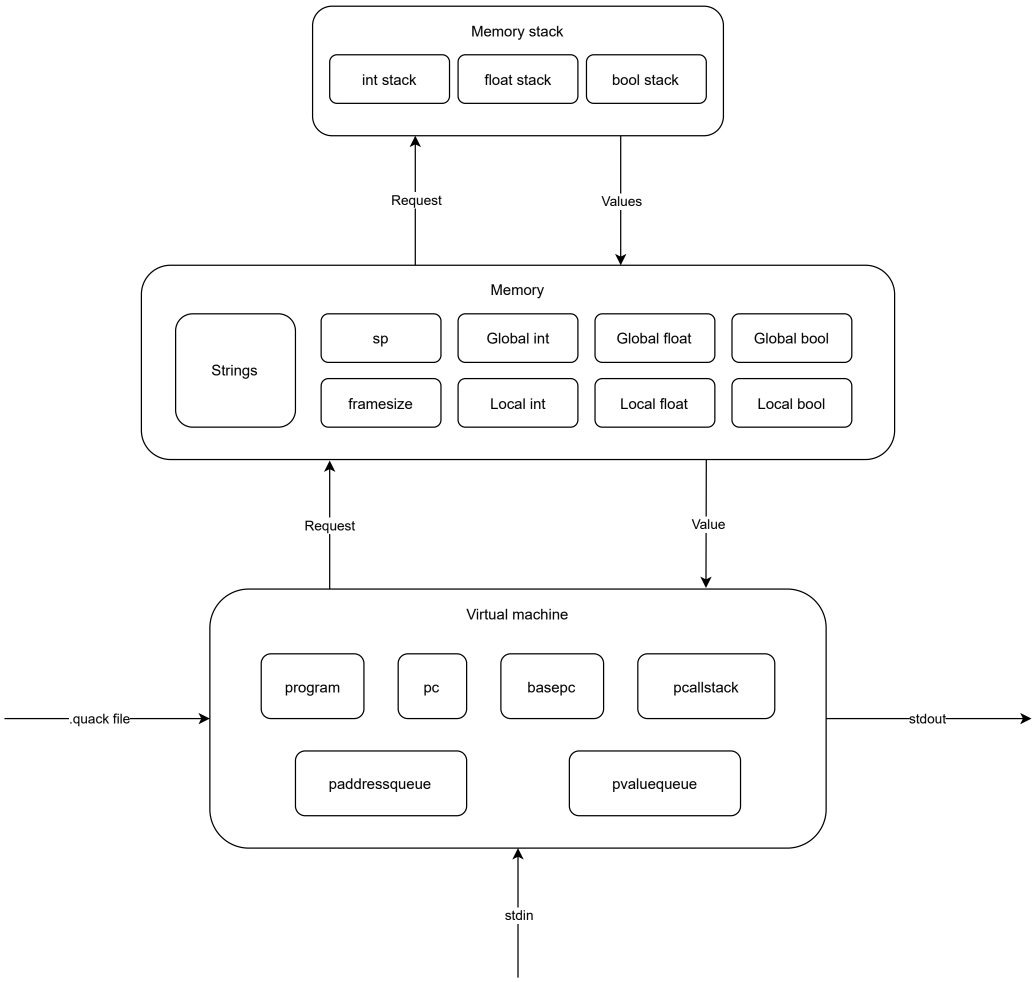
4.2 Memory Management

4.2.1 Architecture

The BigDuck virtual machine is influenced by the architecture used by the MOS 6502 8-bit microprocesor (mainly because this was the one we study in depth on the Computer Organization course). The features taken directly from this processor are the usage of stack pointers to handle function calls and recursion, and the usage of a program counter to keep track program execution.

There are three components, the *virtual* machine which is the one that manages program execution, the *memory* which stores all values and has stack pointers to handle contexts, and the *memory stack* which stores the frame sizes used by each context.

Figure 4.1: Architecture Diagram



4.2.2 Data Structures

Virtual machine

Element	Description
program	Array of three-address code structures which contains program instructions.
pc	Program counter, keeps track of the current instruction to execute.
basepc	Base program counter, points to the beginning of the program segment on the executable.
pcallstack	Procedure call stack, stores the next pc value before jumping to a procedure's code.
paddressqueue	Parameter address queue, stores the addresses of the parameters given to a procedure.
pvaluequeue	Parameter address queue, stores the value of the parameters given to a procedure.

Memory

Element	Description
strings	Stores string values.
sp	Stack pointer, points to the beginning of the frame on each memory pool.
framesize	Stores the size of the current frame.
Local and global pools	For each type, there are 2 pools to maintain the values used in each scope.

Memory Stack

Element	Description
Type stack	Stores the framesizes used by each procedure call.

4.2.3 Virtual Address Translation

The following enumerations were already used throughout compilation.

Scope enumeration

```
0 local
1 global
```

Type enumeration

```
2 0010 int
3 0011 float
4 0100 bool
5 0101 string
```

Therefore it seemed natural to used it as flags on a bit mask in order to assign the virtual addresses.

Memory map

```
1 scope bit
3 type bits
7 address nibbles
```

Examples

```
0010 ... 0000 0000 → local int at address 0
1011 ... 0000 1010 → global float at address 10
0100 ... 0001 0110 → local bool at address 22
1101 ... 0000 1011 → string at address 11
```

Note All strings are global since they cannot be assigned to variables.

This virtual address translations can hold up to $2^{28} = 268,435,456$ addresses per each data type, which means that it can hold around 750 MB of data on a single program. I acknowledge that this is not the most memory efficient mapping however might be the simplest and more eficcient to implement.

Chapter 5

Execution Evidence

5.1 Test Cases

5.1.1 Test Implementation

5.1.2 IR Code Output

5.1.3 Execution Output

Chapter 6

Code Documentation

6.1 Modules Description

Part II

User Manual

Chapter 7

Quick Reference

7.1 Enviroment

7.2 Variables

7.3 Statements

7.3.1 Assignments

7.3.2 Arithmetic Expressions

7.3.3 Operator Hierarchy

7.4 Conditional Statements

7.5 Loop Statements

7.5.1 Infinite Loop

7.5.2 While Loop

7.5.3 For Loop

7.5.4 Control Flow Statements

7.6 Procedures

7.7 Tensorial Types

7.8 Built-in Procedures