# The BigDuck Programming Language

Jair Antonio Bautista Loranca

November 21, 2021

# Contents

# 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. |

| Date | Title | Observations |
|---|---|---|
| Nov 19 | Arrays implemented | Arrays were perhaps one of the most *"challenging"* features to implement, not because of IR code generation, rather the implementation of indirection was not clear to do on the virtual machine. However after some thought I was able to come with a simple but efective solution to the problem. |
| Nov 19 | Tensors implemented | However $n$-dimensional arrays, or as I call them *tensors*, were actually super easy to implement after the experience gained with the implementation of arrays. |
| Nov 20 | Special scalar procedures implemented | The implementation of this functions really helps with the user experience while using the language, since commonly used functions are no longer needed to be implemented on every program and also they are more eficient, since they are just a library call. |

# 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. |
| Tensor dimension must be constant | This occurs when a tensor is declared with variable dimension. |
| Tensor dimension must be greater than 0 | This occurs when a tensor is declared with a not valid dimension. |

| Error message | Description |
|---|---|
| Index value must be of type int | This occurs when the index for a tensor does not resolve into an integer value. |
| Tensor access does not match with dimensions | This occurs when the number of indexes for a tensor does not match with the declared dimensions. |
| Scalar value cannot be indexed | This occurs when it is attempted to index a scalar variable. |

### 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. |
| Unexpected operator | This occurs when an the virtual machine cannot recognized a given operator. |
| 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

**Note** The following grammar is not a one-to-one description of the grammar used in the compiler, this is because there are addtional rules just to have some breakpoints on the grammar required for compilation.

```
        program → vars_decl procs_decl

      vars_decl → var_decl var_decl
                | ϵ
       var_decl → VAR ID next_var var_type “ ; ” next_var_decl
       next_var → “ , ” ID next_var
                | ϵ
  next_var_decl → var_decl next_var_decl
                | ϵ

       var_type → scalar | tensor

         scalar → INT | FLOAT | BOOL

         tensor → dimension scalar
      dimension → “ [ ” num_expr “ ] next_dimension ”
 next_dimension → dimension next_dimension
                | ϵ

     procs_decl → proc_decl procs_decl
                | ϵ

      proc_decl → PROC ID proc_args ret_type local_decl block

      proc_args → “ ( ” “ ) ”
                | “ ( ” ID next_args scalar next_types “ ) ”
      next_args → “ , ” ID next_args
                | ϵ
     next_types → “ ; ” ID next_args scalar next_types
                | ϵ
```

```
 ret_type → “ -> ” scalar
          | ϵ

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
          | variable
          | 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
           | variable
           | proc_call

    variable| ID (dimension | ϵ )

 proc_call → ID “ ( ” (param | ϵ ) “ ) ”
     param → param_term next_param
param_term → bool_expr
           | num_expr
next_param → “ , ” param
     block → “ { ” stmts “ } ”

     stmts → stmt stmts
           | ϵ
      stmt → assigment “ ; ”
           | condition
           | loop_stmt
           | ctrl_flow “ ; ”
           | ret_stmt “ ; ”
           | proc_call “ ; ”

assignment → variable “ <- ” (num_expr | bool_expr)

 condition → IF bool_expr block (alter | ϵ )

     alter → IF bool_expr block (alter | ϵ )

      loop → LOOP (for_style | while_style | infinite) block
 for_style →  (assignment | ϵ ) “ ; ” bool_expr “ ; ” assignment
while_style → bool_expr
  infinite → ϵ
```

## 3.4 IR Code and Semantic Analisis

### 3.4.1 Operation Code

For this project the operation code can be considered as an instruction set, since each of this operation indicates an action to be perform by the virtual machine in order to achieve some computation. The operator code names were chosen to be like mnemonics to facilitate some developement tasks.

| **Operator** | **Description** |
|---|---|
| NOP | Null operator, mainly used as null value for compilation checks. |
| ASG | Assignation. |
| OR | Logical or. |
| AND | Logical and. |
| NOT | Logical not. |
| EQ | Value equality comparison. |
| NEQ | Value inequality comparison. |
| LES | Less than comparison. |
| GRE | Greater than comparison. |
| LEQ | Less than or equal comparison. |
| GEQ | Greater than or equal comparison. |
| SUB | Arithmetic substraction. |
| ADD | Arithmetic addition. |
| DIV | Arithmetic division. |
| MUL | Arithmetic multiplication. |
| GOPROC | Indicates change to a procedure. |
| GOPROC | Indicates change to a procedure. |
| ERA | Indicates the framesizes for new memory to be allocated. |
| PARAM | Indicates value of the parameter to be passed to a procedure. |
| RETURN | Indicates the value to be returned by a procedure. |
| ENDPROC | Clears the procedure context and restores program execution |

| Operator | Description |
|---|---|
| ASSERT | Run-time check for tensor index correctness. |
| PRINT | Prints value to STDOUT. |
| PRINTLN | Prints value and newline char to STDOUT. |
| READ | Reads value form STDIN. |
| SET | Initializes global address with givenn values. |
| PROGRAM | Indicates program starting point on executable. |

### 3.4.2 Virtual Addresses

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. An additional flag was need to add to have support indirection and consequently pointers.

**Memory map**

```
1 addressing mode bit
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 map 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 most efective to implement.

### 3.4.3 Syntax Diagrams

### 3.4.4 Semantic and IR Generation Actions

### 3.4.5 Semantic Consideration Table

## 3.5 Memory Management

**Tree listener**

| Element | Description |
|---|---|
| Filename | Keeps the name of the source code to produce the executable. |
| Valid | Flag to indicate whether an error has been found or not. |
| Debug | Flag to indicate whether debug mode is toggled. |
| Symbol table | Table to keep track of the variable symbols and procedures used in source code. |

# 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.

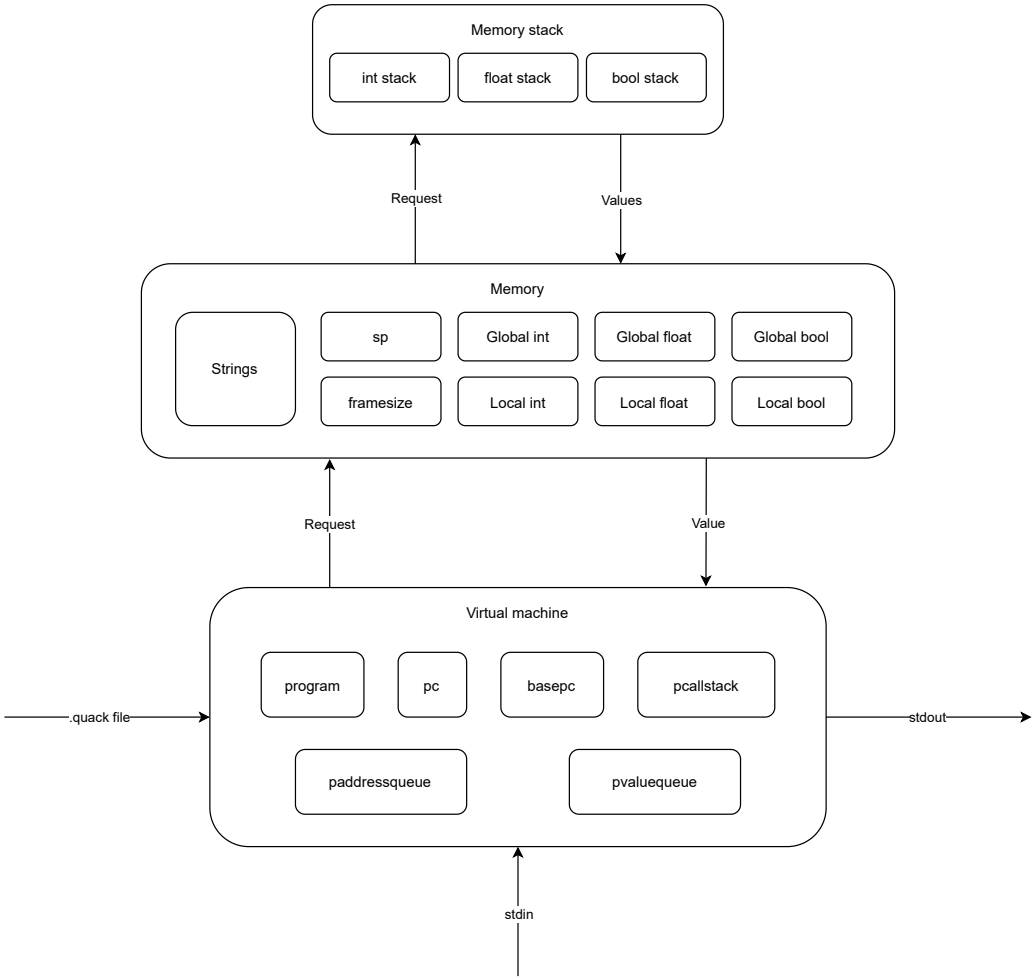
## 4.2 Memory Management

### 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<br>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

Since the memory map is really simple, the virtual address translation is really simple. The following functions take the information embedded on the virtual address to determine; scope, type, address, and addressing mode.

```
func GetScope(address int) int {
    return address & (0x1 << 31) >> 31
}

func GetType(address int) int {
    return address & (0x7 << 28) >> 28
}

func GetAddress(address int) int {
    return address & 0x0fffffff
}

func IsPointer(address int) bool {
    return address & (0x1 << 32) != 0
}
```

# Chapter 5

# Code Documentation

## 5.1 Modules Description

# Part II

# User Manual

# Chapter 6

# Quick Reference

## 6.1 Enviroment Setup

Welcome to the BigDuck programming language reference. Through this chapter it is going to be presented all the syntax and features present on this programming language.

Once downloaded the codebase, on any UNIX-like environment (like macOS or Linux) you can use Make to build the compiler. Just be sure you have installed ANTLR 4.9 on its usual directory `/usr/local/lib/`. However if you are on macOS Monterey, it is almost certain that you can run the `duck` executable like any other executable from the terminal.

After getting the compiler, create a new text file with the `.duck` file extension, and type the following text.

```
proc main() {
    print("Hello, World!");
}
```

Every BigDuck program starts by the last procedure declared (procedures will be explained in more detail further in the chapter). The print command displays on screen the text inside the quotation marks.

Run this program with the following commands.

```
duck hello.duck
duck run hello.quack
```

The first command compiles the source code and creates a new file, called executable, with the same name of the source code file just with the extension changed to `.quack`. The second command will read the file and execute it.

## 6.2 Variables

To work with values it is necessary to store them in variables. Variables can be thought of containers for values in memory, therefore, you can used them to make any desired computation.

Look at the following example for variable declaration.

```
proc main()
    var a, b, c int;
    var x, y float;
    var condition bool;
{
    print(a, b, c);     #| prints: 0 0 0 |#
    print(x, y);        #| prints: 0 0   |#
    print(condition);   #| prints: false |#
}
```

As you can see you have to start with the keyword `var` followed by a list of names separated by commas, and closed by type keyword. This tells to the language that every name on the list will be of the same type.

On the BigDuck language there are 3 primitive types; `int`, `float`, and `bool`. Which are enough for any kind of numeric and logic operation.

The text that is enclosed by `#|` and `|#` is ignored by the compiler, this are called comments and are used to clarify a section of code. In this case they show the output of performing such instructions.

On the BigDuck language all variables are initialize to their respective zero value, for ints and floats is `0`, and for bools is `false`. The next section we will discuss on how to change this values and work with variables.

## 6.3 Statements

On any computational language exists the notion of *sequencing*, this could be for instructions, operations, functions, etc. This sequencing mechanism allows us to indicate the order and steps to be taken by an algorithm.

### 6.3.1 Assignments

After the declaration of a variable, the assignment operator `<-` allows to indicate a value to be hold by the variable. It will remain this value untill another asignment is performed.

### 6.3.2 Arithmetic Expressions

In order to perform operations on values or variables, there are several operators that can be used for different purposes. For example take a look at the following program.

```
proc main()
    var a, b float;
{
    a <- 1;
    b <- 1;
    print(a + b);   #| prints: 2 |#
    a <- 1 + b;     #| now a holds the value: 2 |#
    b <- 5 * b;     #| now b holds the value: 5 |#
    print(a / b);   #| prints: 2 / 5 = 0.4 |#
}
```

### 6.3.3 Operator Precedence and Associativity

As in mathematics, the order of operations is important for certain operations, thus it is advice to have into consideration the following table. The earlier the operator appear on the table, the higher is its precedence. All Operator are left to right associative to provide a natural left to right reading.

| Operator | Usage |
|---|---|
| `()` | `(expression)` |
| `*`, `/` | `a * b`, `a / b` |
| `+`, `-` | `a + b`, `a - b` |
| `=`, `/=`, `<`, `>`, `<=`, `<=`, | `a <relation> b` |
| `not` | `not a` |
| `and` | `a and b` |
| `or` | `a or b` |
| `<-` | `a <- b` |

As you can see multiplication and division, addition and substraction, or relational operators have the same precedence. The order of evaluation is resolved by giving priority to one that was read first.

Therefore the expression `a + b - c` is equal to `(a + b) - c`, and is **not equal** to `a + (b - c)`.

## 6.4 Conditional Statements

On any computational language exists the notion of *decisions*. The decision mechanism is use to perform certain instructions under certain conditions.

The BigDuck language allows for decisions to be taken during program execution. For an example take a look at the following program.

```
proc main()
    var a, b int;
{
    a <- read("Type a value for a");
    b <- read("Type a value for b");

    if a = b {
        print("a equals b");
    } else {
        print("a does not equals b");
    }
}
```

The first two instructions are a especial syntax to indicate that the user can give a value and assign it to a variable. Despite these looking like the value obtained by read is assigned to the variable, the whole line is the read + assigment, therefore no operation can be inmediately applied to a read value. This desicion was taken to enforce legibility.

Whether the given values for a and b are equal or not, the program will print a diffent message. The first print is performed if clause holds true, otherwise the else clause will be perfomed.

Else clauses can be omitted like here.

```
proc main()
    var a, b int;
{
    a <- read("Type a value for a");
    b <- read("Type a value for b");

    if a = b {
        print("a equals b");
    }
}
```

And you can stack if else clauses for multiple cases.

```
proc main()
    var a, b int;
{
    a <- read("Type a value for a");
    b <- read("Type a value for b");

    if a < b {
        print("a is less than b");

    } else if a > b {
        print("a is greater than b");

    } else {
        print("a equals b");
    }
}
```

## 6.5 Loop Statements

### 6.5.1 Infinite Loop

### 6.5.2 While Loop

### 6.5.3 For Loop

### 6.5.4 Do While Loop

### 6.5.5 Control Flow Statements

## 6.6 Procedures

## 6.7 Tensorial Types

## 6.8 Built-in Procedures