

The BigDuck Programming Language

Jair Antonio Bautista Loranca

November 23, 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
1.3	Software Development Process	6
1.3.1	Development Process Description	6
1.3.2	Weekly Log	6
1.3.3	Git Commitments	7
2	Language	11
2.1	General Overview	11
2.1.1	Language Name	11
2.1.2	Main Features Description	11
2.2	Language Errors	12
2.2.1	Compile-Time Errors	12
2.2.2	Run-Time Errors	14
3	Compiler	15
3.1	Development Environment	15
3.2	Lexical Analysis	15
3.3	Syntactical Analysis	16
3.4	IR Code and Semantic Analysis	20
3.4.1	Operation Code	20
3.4.2	Virtual Addresses	22
3.4.3	Syntax Diagrams with actions	24
3.4.4	Semantic Consideration Table	42
3.5	Memory Management	53
4	Virtual Machine	56
4.1	Development Environment	56
4.2	Memory Management	56

4.2.1	Architecture	56
4.2.2	Data Structures	58
4.2.3	Virtual Address Translation	59
5	Execution Evidence	60
5.1	Test Cases	60
5.1.1	Factorial	60
5.1.2	Fibonacci	62
5.1.3	Find	65
5.1.4	Sort	67
5.1.5	Matrix multiplication	69
5.1.6	Built-in procedures	71
6	Code Documentation	74
6.1	General Code Overview	74
6.2	Modules Description	75
6.3	Code excerpts	76
6.3.1	GenerateOpTac	76
6.3.2	Memory management methods	79
II	User Manual	81
7	A Quick Tour	82
7.1	Environment Setup	82
7.2	Variables	83
7.3	Statements	83
7.3.1	Assignments	83
7.3.2	Arithmetic Expressions	84
7.3.3	Operator Precedence and Associativity	84
7.4	Conditional Statements	85
7.5	Loop Statements	86
7.5.1	Infinite Loop	86
7.5.2	While Loop	87
7.5.3	For Loop	87
7.5.4	Do While Loop	88
7.5.5	Control Flow Statements	88
7.6	Procedures	89
7.7	Tensorial Types	91
7.8	Built-in Procedures	92

Part I

Description and Technical Documentation

Chapter 1

Project

1.1 Introduction

1.1.1 Purpose

This document describes the software development 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. Additionally 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 recommendations 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 intermediate 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.

1.3 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`, `.gitignore` updates or documentation advancements).

Date	Title	Observations
Sep 27	First Commit	The work done through the previous HWs were really useful to getting to know ANTLR and make easier the development.
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 infinite 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 necessary 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 strategy to create the context independent 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 “ <i>challenging</i> ” 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 effective solution to the problem.
Nov 19	Tensors implemented	n -dimensional arrays, or as I call them <i>tensors</i> , 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 efficient, since they are just a library call.
Nov 21	Special vectorial procedures implemented	Similar to scalar procedures, these were fairly easy to implement. It just required rebuilding the vectors inside the virtual machine and perform operations.
Nov 22	Procedure call nesting error message added	Throughout testing I noticed that it was not possible currently to handle procedure call nesting. The solution is relatively easy (it requires stacks), however due to the remaining time, I decided to leave it out of the language for the delivery.

Chapter 2

Language

2.1 General Overview

2.1.1 Language Name

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 mathematical and scientific computations, numerical methods, and some basic statistics. Therefore this language includes integer and floating point arithmetic, trigonometric and trascendental functions, other commonly used math operations, and vector operations to support some statistical functions.

2.2 Language Errors

2.2.1 Compile-Time Errors

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 expected n attributes 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 evaluate 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.
Cannot use control flow statements outside of loops	This occurs when a control flow statement was used outside of a loop.
Procedure call nesting is not supported for user procedures	This occurs when it is attempted to call a function inside a user procedure parameter.
Cannot perform operation on scalar values	This occurs when it is attempted to use a vectorial function with an scalar value as a parameter.
Cannot perform operation on a boolean vector	This occurs when it is attempted to use a vectorial function for numeric types with an boolean vector as a parameter.
Cannot perform operation on higher dimensions	This occurs when it is attempted to use a vectorial function over a higher dimensional tensor.

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 Analysis

Reserved Keywords

proc	return	if	else	loop	break	skip	and
or	not	var	int	float	bool	true	false
print	read	sin	asin	cos	acos	tan	atan
atan2	exp	exp	ln	sqrt	pow	mod	abs
ceil	floor	mean	median	mode			

Tokens

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


3.3 Syntactical Analysis

Note The following grammar is not a one-to-one description of the grammar used in the compiler, this is because there are additional 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
        | functions

variable | ID (dimension |  $\epsilon$  )

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

stmts → stmt stmts
        |  $\epsilon$ 
stmt → assignment “ ; ”
        | condition
        | loop_stmt
        | ctrl_flow “ ; ”
        | ret_stmt “ ; ”
        | proc_call “ ; ”
        | built_in “ ; ”

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

condition → IF bool_expr block (alter |  $\epsilon$  )

alter → IF bool_expr block (alter |  $\epsilon$  )

loop → LOOP (for_style | while_style | infinite) block
for_style → (assignment |  $\epsilon$  ) “ ; ” bool_expr “ ; ” assignment
while_style → bool_expr
infinite →  $\epsilon$ 

```

```

built_in → print
          | read

functions → u_func
           | bin_func
           | vec_func

print → PRINT “ ( ” print_param “ ) ”
print_param → print_term print_next_param
print_term → bool_expr
            | num_expr
            | CTE_STRING
print_next_param → “ , ” print_param
                  | ε

u_func → u_funcs “ ( ” num_expr “ ) ”
u_funcs → SIN | ASIN | COS | ACOS | TAN | ATAN
         | EXP | LN | SQRT | ABS | CEIL | FLOOR

bin_func → bin_funcs “ ( ” num_expr “ , ” num_expr “ ) ”
bin_funcs → ATAN2 | POW | LOG | MOD

vec_func → vec_funcs “ ( ” variable “ ) ”
vec_funcs → MEAN | MEDIAN | MODE

```

3.4 IR Code and Semantic Analysis

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	Assigation.
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.
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
ASSERT	Run-time check for tensor index correctness.

Operator	Description
PRINT	Prints value to STDOUT.
PRINTLN	Prints value and newline char to STDOUT.
READ	Reads value form STDIN, and assigns it to a value.
SIN	Trigonometric sin function.
ASIN	Trigonometric arcsin function.
COS	Trigonometric cos function.
ACOS	Trigonometric arccos function.
TAN	Trigonometric tan function.
ATAN	Trigonometric atan function.
ATAN2	Trigonometric atan2 function.
EXP	Exponential function.
LN	Natural logarithm function.
SQRT	Square root function.
POW	Raise number x to the y power.
LOG	Logarithm base b of x .
MOD	Modulus base b of n .
ABS	Absolute value function.
CEIL	Ceiling function.
FLOOR	Floor function.
MEAN	Mean of a vector.
MEDIAN	Meadian of a vector.
MODE	Mode of a vector.
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 use it as flags on a bit mask in order to assign the virtual addresses. An additional flag was needed to add support for indirection and consequently pointers.

Memory map

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

Examples

```
0 0010 ... 0000 0000 → local int at address 0
0 1011 ... 0000 1010 → global float at address 10
0 0100 ... 0001 0110 → local bool at address 22
0 1101 ... 0000 1011 → string at address 11
1 1010 ... 0000 0101 → local pointer at address 5
```

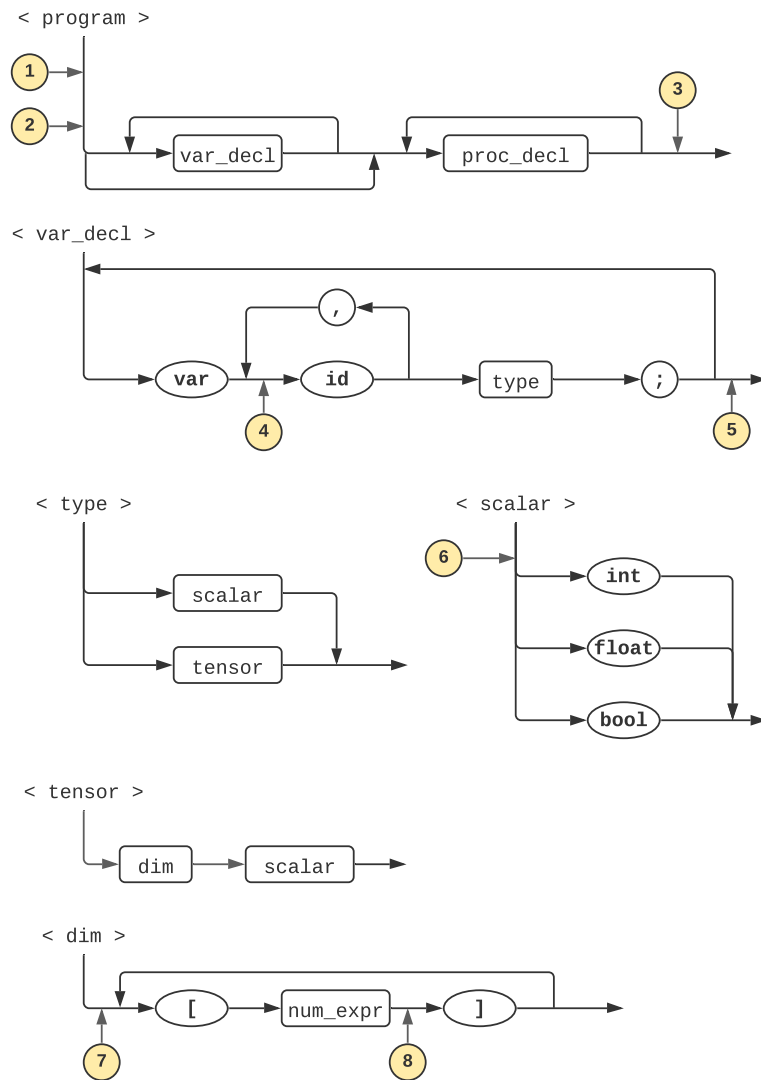
Note 1 All strings are global since they cannot be assigned to variables.

Note 2 All pointers are `int` since they hold integer values.

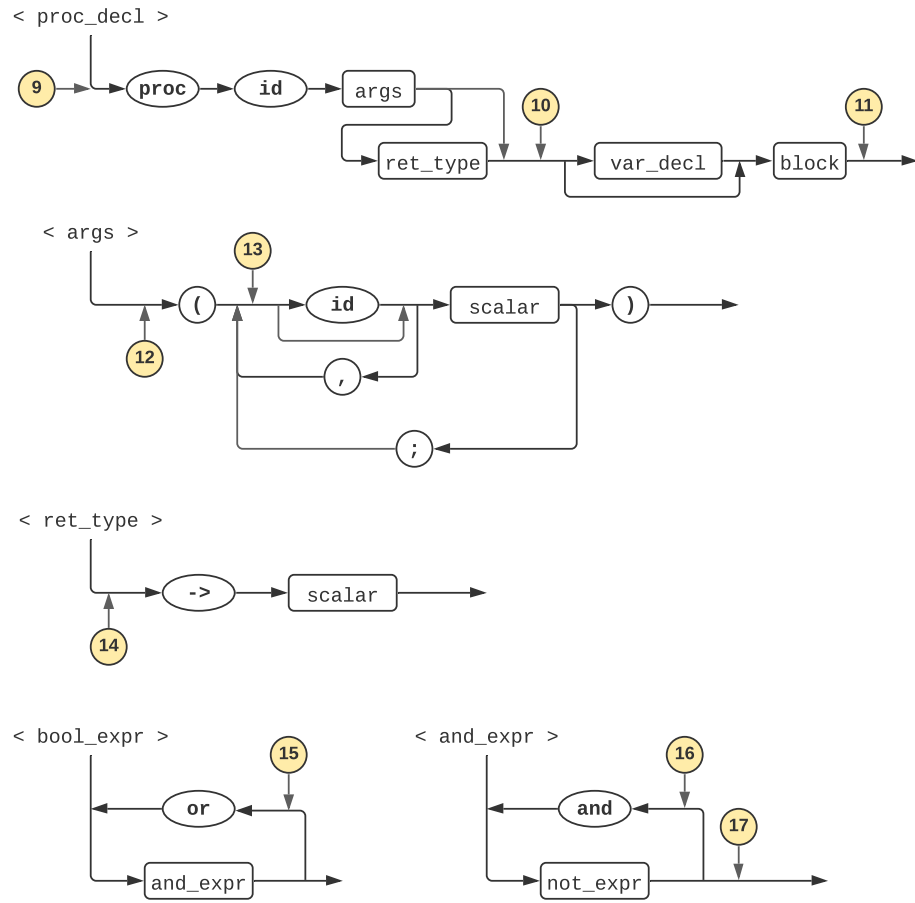
This virtual address map can hold up to $2^{28} - 1 = 268,435,455$ addresses per each data type, which means that it can hold around 750 MB of data on a single program. I acknowledge that this is not the most memory efficient mapping however might be the simplest and most effective to implement.

This page was left empty on purpose For easier read of the syntax diagrams with its actions, it is advised to view the pdf by 2 pages. In such way that on left page are the diagrams and the right page are the actions description.

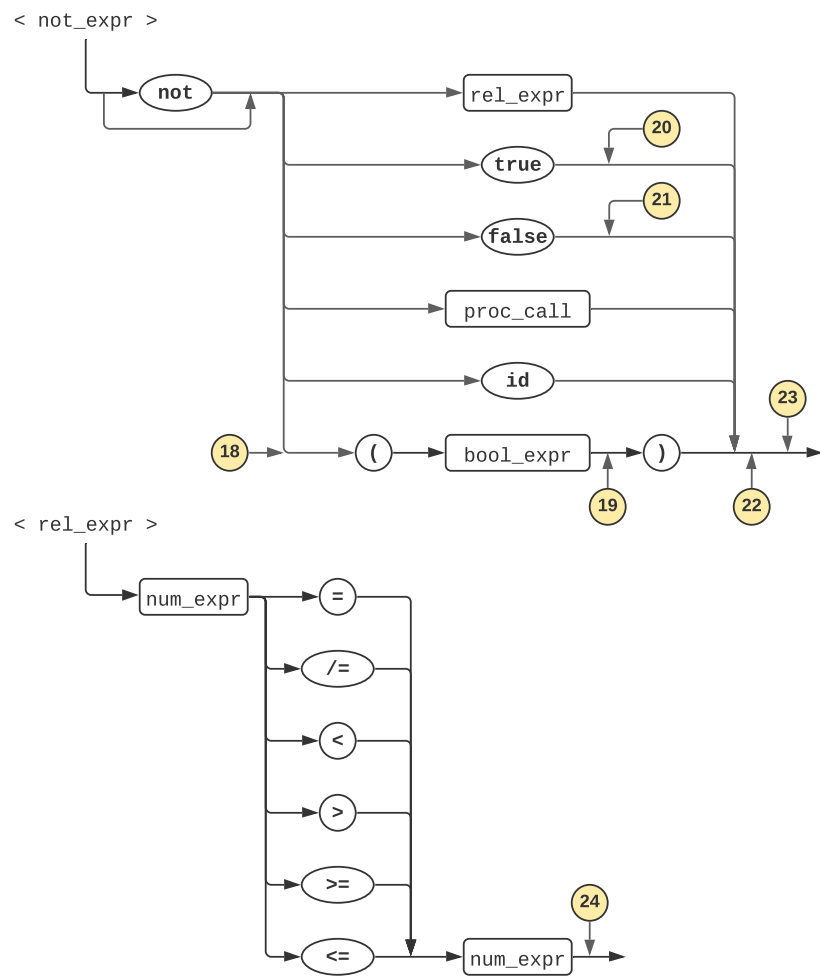
3.4.3 Syntax Diagrams with actions



No.	Description
1	Compiler initialization
2	Generate era and goproc for starting procedure
3	If valid program the create obj file
4	If valid program the create obj file
5	Turn on in_decl flag
6	Turn off in_decl flag
7	Turn off in_decl flag
8	Use dimqueue and symqueue to create symbols and add each to symbol table, if symbol has been read then raise error

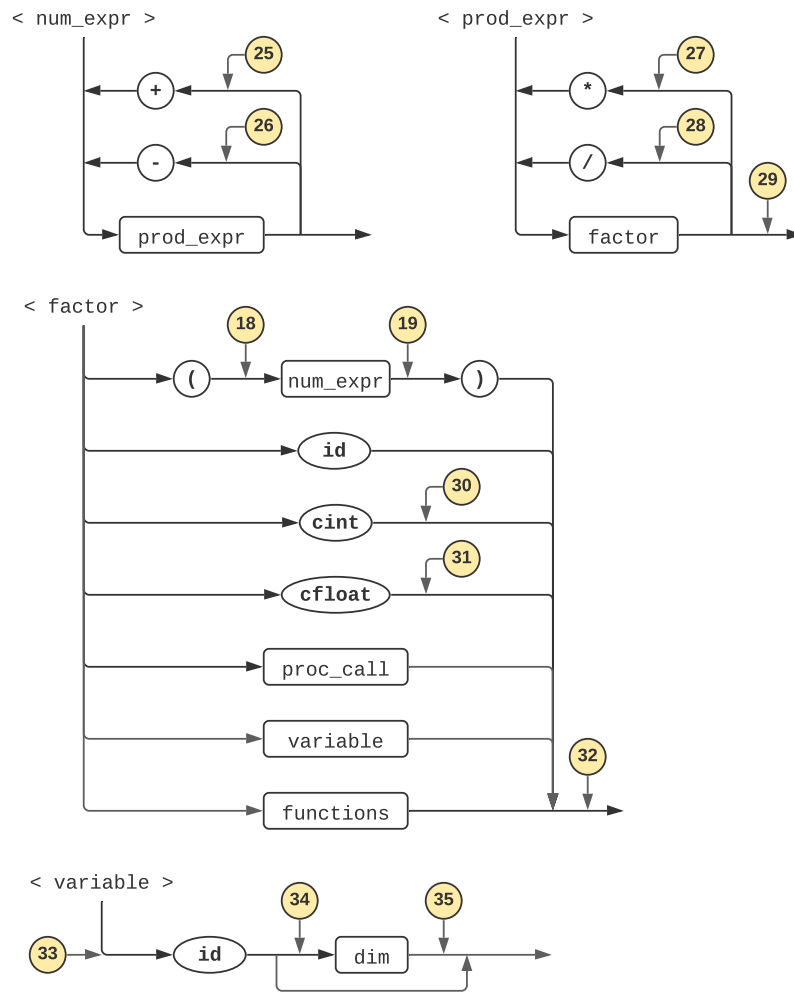


No.	Description
9	Add id to symbol table, if symbol has been read then raise error
10	Update procedure's parameter information, argc and ret_type
11	Generate endproc, update procedure's type counters, resolve recursive calls, and clear scope
12	Turn on in_decl and in_args flags
13	Push id to symqueue
14	Register return type
15	Push or to opstack
16	Push and to opstack
17	If or at top of opstack then GenerateOpTac



No.	Description
18	Push LPAREN to opstack
19	Push RPAREN to opstack
20	Push #t to argstack and push bool_t to typestack
21	Push #f to argstack and push bool_t to typestack
22	If and at top of opstack then GenerateOpTac
23	If not at top of opstack then GenerateOpTac
24	GenerateOpTac

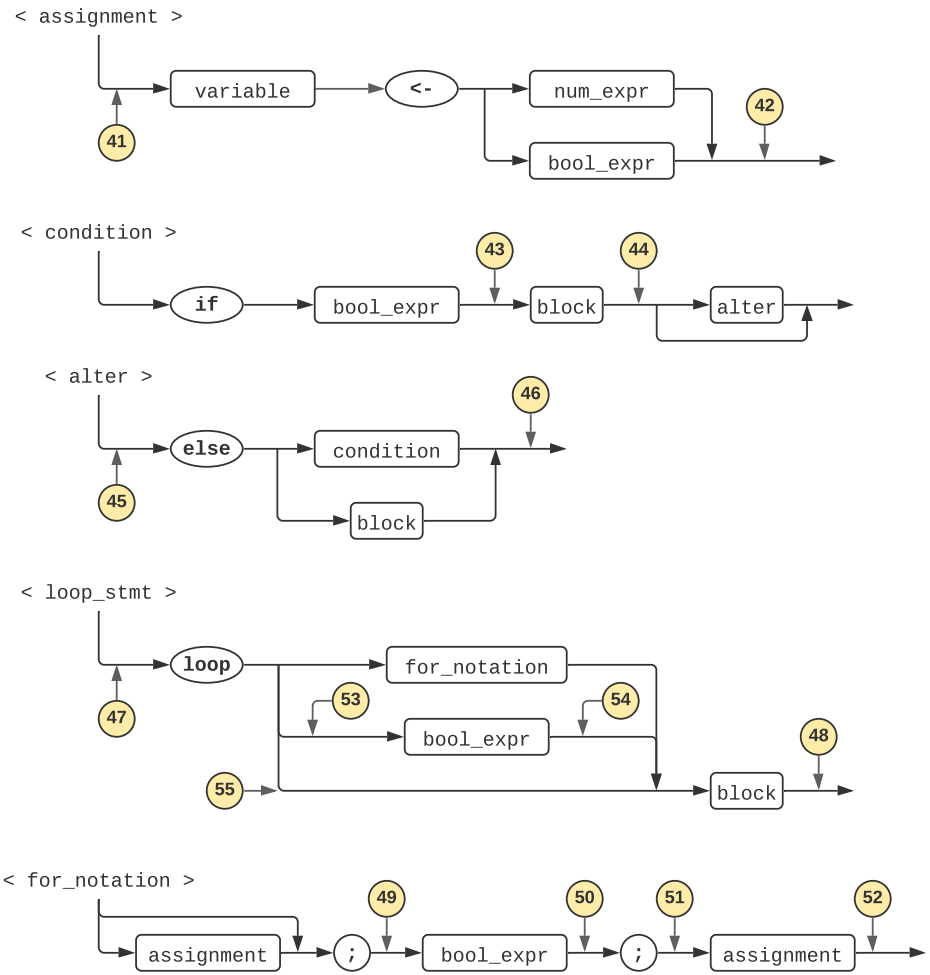
Note Keep in mind actions 18 and 19, since they are reused.



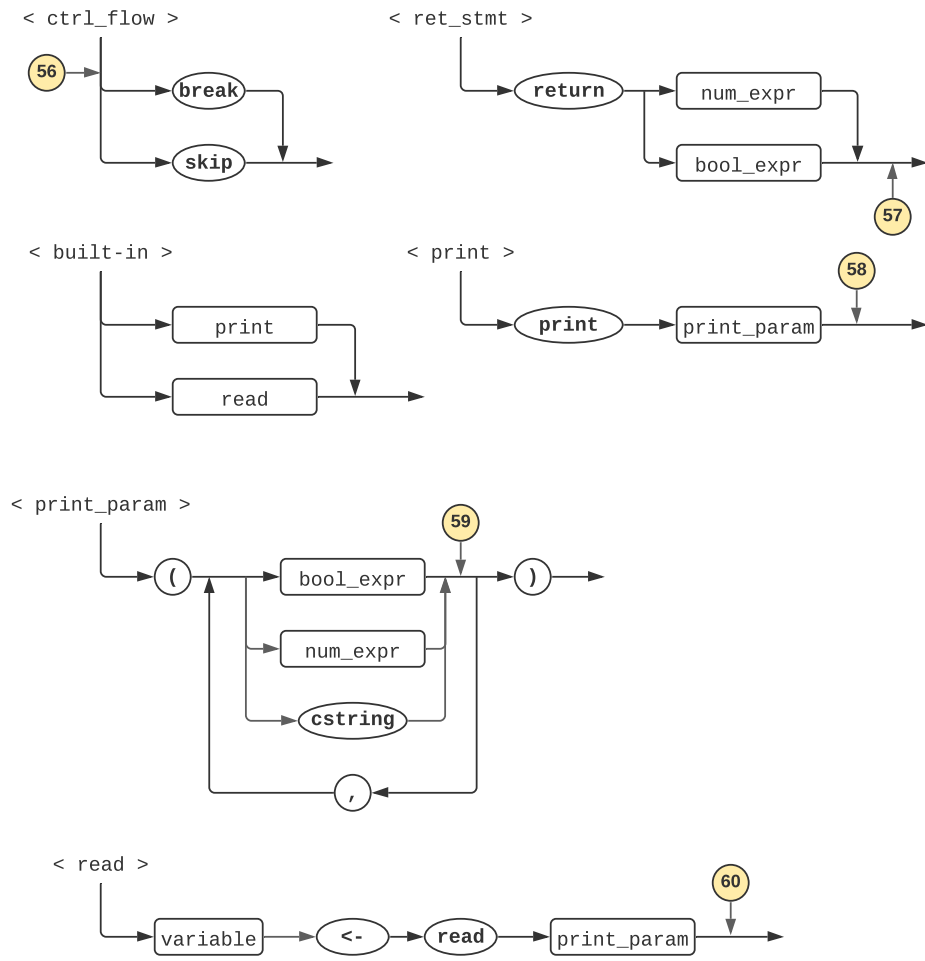
No.	Description
25	Push + to opstack
26	Push — to opstack
27	Push * to opstack
28	Push / to opstack
29	If + or — at top of opstack then GenerateOpTac
30	Push int literal to argstack push int_t to typestack
31	Push float literal to argstack push float_t to typestack
32	If * or / at top of opstack then GenerateOpTac
33	Push id to argstack and its type to typestack
34	If id dimension is 0 then raise error else push LPAREN
35	Push RPAREN to opstack, if curr_dim is different from expected dimension then raise error



No.	Description
36	Init paramc, if curr_pcall is empty then assign id to curr_pcall else raise error, if exists in syntable the generate era else raise error
37	Push RPAREN to opstack, GenerateParamTac
38	if has return value then GenereteReturnTac else if in_stmt Generate goproc, if paramc different from procedures argc then raise error
39	Turn on in_stmt flag
40	Turn off in_stmt flag

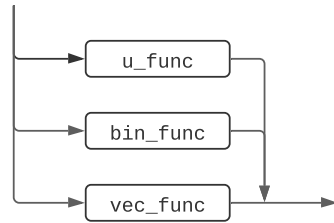


No.	Description
41	Push \leftarrow to opstack
42	GenerateOpTac
43	Push pc to jmpstack and GenerateJumpTac (JMF)
44	Pop jmpstack and use value fo FillJumpTac
45	Push pc to jmpstack and FillJumpTac
46	Pop jmpstack and use value fo FillJumpTac
47	Increment loop_nest
48	Fill unresolved jumps according loop style, fill skips and breaks, decrement loop_nest
49	Push pc to jmpstack and set loopstyle to ForStyle
50	Push pc to jmpstack, GenerateJumpTac (JMT), push pc to jmpstack and GenerateJumpTac (JMP)
51	Push pc to jmpstack
52	GenerateJumpTac (JMP), handle jumps for condition when true, false, and return from control variable assignment
53	Push pc to jmpstack and set loopstyle to WhileStyle
54	Push pc to jmpstack and GenerateJumpTac (JMF)
55	Push pc to jmpstack and set loopstyle to InfLoop

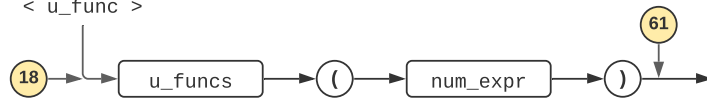


No.	Description
56	Push break or skip to its appropriate queue, GenerateJmpTac (JMP)
57	GenerateRetTac
58	Change last PRINT to PRINTLN
59	GeneratePrintTac
60	Pop argstack and typestack and use values to generate read TAC

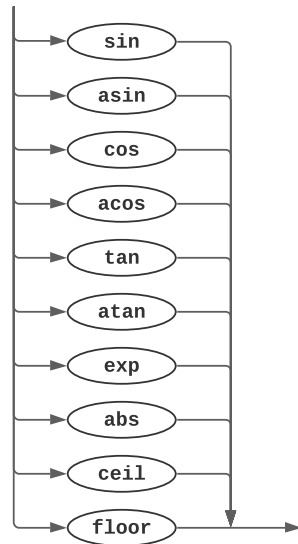
< functions >



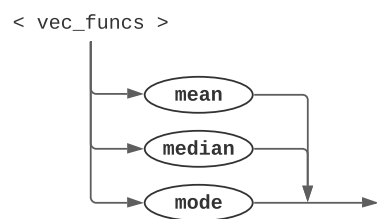
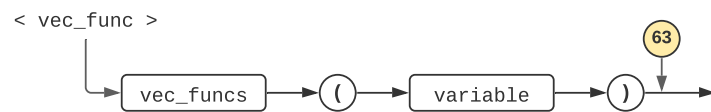
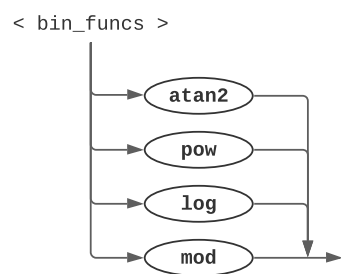
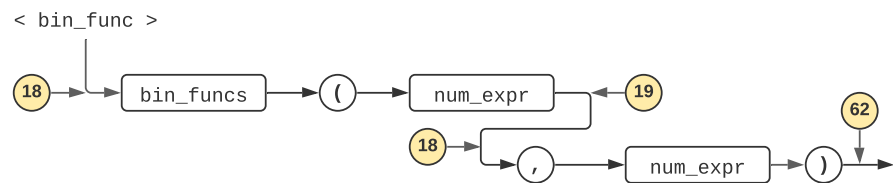
< u_func >



< u_funcs >



No.	Description
61	Push unary function to opstack, GenerateOpTac and push RPAREN to opstack



No.	Description
62	Push binary function to opstack, GenerateOpTac and push RPAREN to opstack
63	If variable exists and has appropriate dimensions then push vec function to opstack, GenerateOp-Stack

3.4.4 Semantic Consideration Table

Operator	Type 1	Type2	Result type
ASG	Int_t	Int_t	Int_t
ASG	Int_t	Float_t	Int_t
ASG	Int_t	Bool_t	Error_t
ASG	Float_t	Int_t	Float_t
ASG	Float_t	Float_t	Float_t
ASG	Float_t	Bool_t	Error_t
ASG	Bool_t	Int_t	Error_t
ASG	Bool_t	Float_t	Error_t
ASG	Bool_t	Bool_t	Bool_t
ADD	Int_t	Int_t	Int_t
ADD	Int_t	Float_t	Float_t
ADD	Int_t	Bool_t	Error_t
ADD	Float_t	Int_t	Float_t
ADD	Float_t	Float_t	Float_t
ADD	Float_t	Bool_t	Error_t
ADD	Bool_t	Int_t	Error_t
ADD	Bool_t	Float_t	Error_t
ADD	Bool_t	Bool_t	Error_t
SUB	Int_t	Int_t	Int_t
SUB	Int_t	Float_t	Float_t
SUB	Int_t	Bool_t	Error_t
SUB	Float_t	Int_t	Float_t
SUB	Float_t	Float_t	Float_t
SUB	Float_t	Bool_t	Error_t
SUB	Bool_t	Int_t	Error_t
SUB	Bool_t	Float_t	Error_t
SUB	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
MUL	Int_t	Int_t	Int_t
MUL	Int_t	Float_t	Float_t
MUL	Int_t	Bool_t	Error_t
MUL	Float_t	Int_t	Float_t
MUL	Float_t	Float_t	Float_t
MUL	Float_t	Bool_t	Error_t
MUL	Bool_t	Int_t	Error_t
MUL	Bool_t	Float_t	Error_t
MUL	Bool_t	Bool_t	Error_t
DIV	Int_t	Int_t	Int_t
DIV	Int_t	Float_t	Float_t
DIV	Int_t	Bool_t	Error_t
DIV	Float_t	Int_t	Float_t
DIV	Float_t	Float_t	Float_t
DIV	Float_t	Bool_t	Error_t
DIV	Bool_t	Int_t	Error_t
DIV	Bool_t	Float_t	Error_t
DIV	Bool_t	Bool_t	Error_t
AND	Int_t	Int_t	Error_t
AND	Int_t	Float_t	Error_t
AND	Int_t	Bool_t	Error_t
AND	Float_t	Int_t	Error_t
AND	Float_t	Float_t	Error_t
AND	Float_t	Bool_t	Error_t
AND	Bool_t	Int_t	Error_t
AND	Bool_t	Float_t	Error_t
AND	Bool_t	Bool_t	Bool_t

Operator	Type 1	Type2	Result type
OR	Int_t	Int_t	Error_t
OR	Int_t	Float_t	Error_t
OR	Int_t	Bool_t	Error_t
OR	Float_t	Int_t	Error_t
OR	Float_t	Float_t	Error_t
OR	Float_t	Bool_t	Error_t
OR	Bool_t	Int_t	Error_t
OR	Bool_t	Float_t	Error_t
OR	Bool_t	Bool_t	Bool_t
NOT	Int_t	Int_t	Error_t
NOT	Int_t	Float_t	Error_t
NOT	Int_t	Bool_t	Error_t
NOT	Float_t	Int_t	Error_t
NOT	Float_t	Float_t	Error_t
NOT	Float_t	Bool_t	Error_t
NOT	Bool_t	Int_t	Error_t
NOT	Bool_t	Float_t	Error_t
NOT	Bool_t	Bool_t	Bool_t
EQ	Int_t	Int_t	Bool_t
EQ	Int_t	Float_t	Bool_t
EQ	Int_t	Bool_t	Error_t
EQ	Float_t	Int_t	Bool_t
EQ	Float_t	Float_t	Bool_t
EQ	Float_t	Bool_t	Error_t
EQ	Bool_t	Int_t	Error_t
EQ	Bool_t	Float_t	Error_t
EQ	Bool_t	Bool_t	Bool_t

Operator	Type 1	Type2	Result type
NEQ	Int_t	Int_t	Bool_t
NEQ	Int_t	Float_t	Bool_t
NEQ	Int_t	Bool_t	Error_t
NEQ	Float_t	Int_t	Bool_t
NEQ	Float_t	Float_t	Bool_t
NEQ	Float_t	Bool_t	Error_t
NEQ	Bool_t	Int_t	Error_t
NEQ	Bool_t	Float_t	Error_t
NEQ	Bool_t	Bool_t	Bool_t
LES	Int_t	Int_t	Bool_t
LES	Int_t	Float_t	Bool_t
LES	Int_t	Bool_t	Error_t
LES	Float_t	Int_t	Bool_t
LES	Float_t	Float_t	Bool_t
LES	Float_t	Bool_t	Error_t
LES	Bool_t	Int_t	Error_t
LES	Bool_t	Float_t	Error_t
LES	Bool_t	Bool_t	Error_t
GRE	Int_t	Int_t	Bool_t
GRE	Int_t	Float_t	Bool_t
GRE	Int_t	Bool_t	Error_t
GRE	Float_t	Int_t	Bool_t
GRE	Float_t	Float_t	Bool_t
GRE	Float_t	Bool_t	Error_t
GRE	Bool_t	Int_t	Error_t
GRE	Bool_t	Float_t	Error_t
GRE	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
LEQ	Int_t	Int_t	Bool_t
LEQ	Int_t	Float_t	Bool_t
LEQ	Int_t	Bool_t	Error_t
LEQ	Float_t	Int_t	Bool_t
LEQ	Float_t	Float_t	Bool_t
LEQ	Float_t	Bool_t	Error_t
LEQ	Bool_t	Int_t	Error_t
LEQ	Bool_t	Float_t	Error_t
LEQ	Bool_t	Bool_t	Error_t
GEQ	Int_t	Int_t	Bool_t
GEQ	Int_t	Float_t	Bool_t
GEQ	Int_t	Bool_t	Error_t
GEQ	Float_t	Int_t	Bool_t
GEQ	Float_t	Float_t	Bool_t
GEQ	Float_t	Bool_t	Error_t
GEQ	Bool_t	Int_t	Error_t
GEQ	Bool_t	Float_t	Error_t
GEQ	Bool_t	Bool_t	Error_t
SIN	Int_t	Int_t	Float_t
SIN	Int_t	Float_t	Error_t
SIN	Int_t	Bool_t	Error_t
SIN	Float_t	Int_t	Error_t
SIN	Float_t	Float_t	Float_t
SIN	Float_t	Bool_t	Error_t
SIN	Bool_t	Int_t	Error_t
SIN	Bool_t	Float_t	Error_t
SIN	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
ASIN	Int_t	Int_t	Float_t
ASIN	Int_t	Float_t	Error_t
ASIN	Int_t	Bool_t	Error_t
ASIN	Float_t	Int_t	Error_t
ASIN	Float_t	Float_t	Float_t
ASIN	Float_t	Bool_t	Error_t
ASIN	Bool_t	Int_t	Error_t
ASIN	Bool_t	Float_t	Error_t
ASIN	Bool_t	Bool_t	Error_t
COS	Int_t	Int_t	Float_t
COS	Int_t	Float_t	Error_t
COS	Int_t	Bool_t	Error_t
COS	Float_t	Int_t	Error_t
COS	Float_t	Float_t	Float_t
COS	Float_t	Bool_t	Error_t
COS	Bool_t	Int_t	Error_t
COS	Bool_t	Float_t	Error_t
COS	Bool_t	Bool_t	Error_t
ACOS	Int_t	Int_t	Float_t
ACOS	Int_t	Float_t	Error_t
ACOS	Int_t	Bool_t	Error_t
ACOS	Float_t	Int_t	Error_t
ACOS	Float_t	Float_t	Float_t
ACOS	Float_t	Bool_t	Error_t
ACOS	Bool_t	Int_t	Error_t
ACOS	Bool_t	Float_t	Error_t
ACOS	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
TAN	Int_t	Int_t	Float_t
TAN	Int_t	Float_t	Error_t
TAN	Int_t	Bool_t	Error_t
TAN	Float_t	Int_t	Error_t
TAN	Float_t	Float_t	Float_t
TAN	Float_t	Bool_t	Error_t
TAN	Bool_t	Int_t	Error_t
TAN	Bool_t	Float_t	Error_t
TAN	Bool_t	Bool_t	Error_t
ATAN	Int_t	Int_t	Float_t
ATAN	Int_t	Float_t	Error_t
ATAN	Int_t	Bool_t	Error_t
ATAN	Float_t	Int_t	Error_t
ATAN	Float_t	Float_t	Float_t
ATAN	Float_t	Bool_t	Error_t
ATAN	Bool_t	Int_t	Error_t
ATAN	Bool_t	Float_t	Error_t
ATAN	Bool_t	Bool_t	Error_t
ATAN2	Int_t	Int_t	Float_t
ATAN2	Int_t	Float_t	Error_t
ATAN2	Int_t	Bool_t	Error_t
ATAN2	Float_t	Int_t	Error_t
ATAN2	Float_t	Float_t	Float_t
ATAN2	Float_t	Bool_t	Error_t
ATAN2	Bool_t	Int_t	Error_t
ATAN2	Bool_t	Float_t	Error_t
ATAN2	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
EXP	Int_t	Int_t	Float_t
EXP	Int_t	Float_t	Float_t
EXP	Int_t	Bool_t	Error_t
EXP	Float_t	Int_t	Float_t
EXP	Float_t	Float_t	Float_t
EXP	Float_t	Bool_t	Error_t
EXP	Bool_t	Int_t	Error_t
EXP	Bool_t	Float_t	Error_t
EXP	Bool_t	Bool_t	Error_t
LN	Int_t	Int_t	Float_t
LN	Int_t	Float_t	Error_t
LN	Int_t	Bool_t	Error_t
LN	Float_t	Int_t	Error_t
LN	Float_t	Float_t	Float_t
LN	Float_t	Bool_t	Error_t
LN	Bool_t	Int_t	Error_t
LN	Bool_t	Float_t	Error_t
LN	Bool_t	Bool_t	Error_t
SQRT	Int_t	Int_t	Float_t
SQRT	Int_t	Float_t	Float_t
SQRT	Int_t	Bool_t	Error_t
SQRT	Float_t	Int_t	Float_t
SQRT	Float_t	Float_t	Float_t
SQRT	Float_t	Bool_t	Error_t
SQRT	Bool_t	Int_t	Error_t
SQRT	Bool_t	Float_t	Error_t
SQRT	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
POW	Int_t	Int_t	Float_t
POW	Int_t	Float_t	Float_t
POW	Int_t	Bool_t	Error_t
POW	Float_t	Int_t	Float_t
POW	Float_t	Float_t	Float_t
POW	Float_t	Bool_t	Error_t
POW	Bool_t	Int_t	Error_t
POW	Bool_t	Float_t	Error_t
POW	Bool_t	Bool_t	Error_t
LOG	Int_t	Int_t	Float_t
LOG	Int_t	Float_t	Float_t
LOG	Int_t	Bool_t	Error_t
LOG	Float_t	Int_t	Float_t
LOG	Float_t	Float_t	Float_t
LOG	Float_t	Bool_t	Error_t
LOG	Bool_t	Int_t	Error_t
LOG	Bool_t	Float_t	Error_t
LOG	Bool_t	Bool_t	Error_t
MOD	Int_t	Int_t	Float_t
MOD	Int_t	Float_t	Float_t
MOD	Int_t	Bool_t	Error_t
MOD	Float_t	Int_t	Float_t
MOD	Float_t	Float_t	Float_t
MOD	Float_t	Bool_t	Error_t
MOD	Bool_t	Int_t	Error_t
MOD	Bool_t	Float_t	Error_t
MOD	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
ABS	Int_t	Int_t	Float_t
ABS	Int_t	Float_t	Error_t
ABS	Int_t	Bool_t	Error_t
ABS	Float_t	Int_t	Error_t
ABS	Float_t	Float_t	Float_t
ABS	Float_t	Bool_t	Error_t
ABS	Bool_t	Int_t	Error_t
ABS	Bool_t	Float_t	Error_t
ABS	Bool_t	Bool_t	Error_t
CEIL	Int_t	Int_t	Float_t
CEIL	Int_t	Float_t	Float_t
CEIL	Int_t	Bool_t	Error_t
CEIL	Float_t	Int_t	Float_t
CEIL	Float_t	Float_t	Float_t
CEIL	Float_t	Bool_t	Error_t
CEIL	Bool_t	Int_t	Error_t
CEIL	Bool_t	Float_t	Error_t
CEIL	Bool_t	Bool_t	Error_t
FLOOR	Int_t	Int_t	Float_t
FLOOR	Int_t	Float_t	Error_t
FLOOR	Int_t	Bool_t	Error_t
FLOOR	Float_t	Int_t	Error_t
FLOOR	Float_t	Float_t	Float_t
FLOOR	Float_t	Bool_t	Error_t
FLOOR	Bool_t	Int_t	Error_t
FLOOR	Bool_t	Float_t	Error_t
FLOOR	Bool_t	Bool_t	Error_t

Operator	Type 1	Type2	Result type
MEAN	Int_t	Int_t	Float_t
MEAN	Int_t	Float_t	Float_t
MEAN	Int_t	Bool_t	Error_t
MEAN	Float_t	Int_t	Float_t
MEAN	Float_t	Float_t	Float_t
MEAN	Float_t	Bool_t	Error_t
MEAN	Bool_t	Int_t	Error_t
MEAN	Bool_t	Float_t	Error_t
MEAN	Bool_t	Bool_t	Error_t
MEDIAN	Int_t	Int_t	Float_t
MEDIAN	Int_t	Float_t	Float_t
MEDIAN	Int_t	Bool_t	Error_t
MEDIAN	Float_t	Int_t	Float_t
MEDIAN	Float_t	Float_t	Float_t
MEDIAN	Float_t	Bool_t	Error_t
MEDIAN	Bool_t	Int_t	Error_t
MEDIAN	Bool_t	Float_t	Error_t
MEDIAN	Bool_t	Bool_t	Error_t
MODE	Int_t	Int_t	Float_t
MODE	Int_t	Float_t	Float_t
MODE	Int_t	Bool_t	Error_t
MODE	Float_t	Int_t	Float_t
MODE	Float_t	Float_t	Float_t
MODE	Float_t	Bool_t	Error_t
MODE	Bool_t	Int_t	Error_t
MODE	Bool_t	Float_t	Error_t
MODE	Bool_t	Bool_t	Error_t

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.
symtable	Table to keep track of the variable symbols and procedures used in source code.
symqueue	Queue to keep track of which symbols are under the same declaration list.
typequeue	Queue to keep track of which symbols are under the same declaration line.
paramqueue	Queue to keep track of which symbols are on the procedures parameters.
dimqueue	Queue to keep track of which the dimensions used by a tensor.
in_decl	Flag to know if it is reading a variable declaration.
in_args	Flag to know if it is reading the arguments of a procedure.
in_stmt	Flag to know if it is reading a statement.
scope	Flag to know the current scope.
argc	Counter for arguments used in a procedure call.
loop_nest	Counter to keep track of loop nesting.
ret_type	Variable to keep track of current procedure's return type.
curr_proc	Variable to keep track of the name of current procedure.

Element	Description
ir_code	Array of TACs (three-address code) generated by the compiler.
data_seg	Array of TACs used for data segment generated by the compiler.
op_stack	Stack of operators for expression compilation.
arg_stack	Stack of arguments for operators for expression compilation.
type_stack	Stack of type variables for expression compilation.
pc	Program counter to keep track of generated TACs.
tmpe	Temporal counter to keep track of generated temporal variables.
tmpe	Temporal counter to keep track of generated temporal variables.
paramc	Parameter counter to keep track of number of parameters given to a procedure.
loopstyle	Variable to keep track of which kind of loop syntax is used.
startpoint	Variable to keep track of where to start the program.
startproc	Variable to keep track of the name of initial procedure.
curr_line	Variable to keep track of current line.
curr_column	Variable to keep track of current column.
curr_pcall	Variable to keep track of the name of current procedure call.
curr_tensor	Variable to keep track of the name of current tensor in use.
curr_dim	Variable to keep track of the name of current dimension index used in tensor indexing.
memmap	Memory mapper, to assign a memory address to a used variable.

Symbol

Element	Description
Stype	Enumeration to indicate the type of a variable.
Dim	Array to keep track of the dimensions of a variable.
Baddress	Base address, to keep track of base address for tensorial types.
Argc	Argument count, to keep track of the arguments required for procedure.
TypeArgs	Array to keep track of the number of arguments of a procedure.
RetType	Variable to keep track of the return type of an argument.
Startpoint	Variable to keep track of the starting point of a procedure.
Paddress	Parameter address, array to keep track of the reserved addresses for the parameter.
Type count	Counters for the amount of variables of each type required.

Symbol Table

Element	Description
table	An array of size 2 of hash tables that take strings as keys and Symbols as values.

Memory mapper

Element	Description
memcache	Hash table of strings as keys and int as value, to store every symbol with a memory address.
Typecount	An array of size 2 of hash table that takes strings as keys and ints as values, to store a counter of used memory per scope and type.

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.

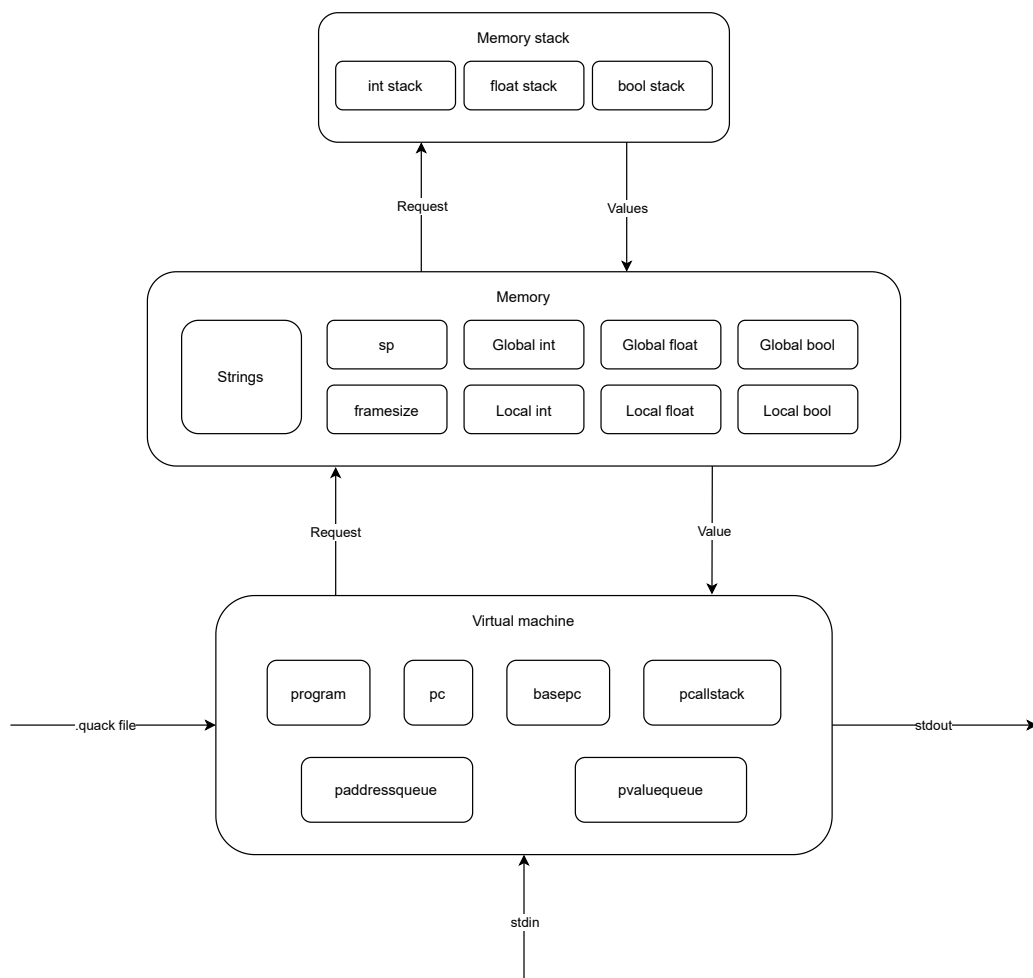
4.2 Memory Management

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

Since the memory map is really simple, the virtual address translation is really simple. The following functions take the information embedded in 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 & 0xffffffff
}

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

Chapter 5

Execution Evidence

5.1 Test Cases

5.1.1 Factorial

```
#| Iterative factorial implementation |#
proc fact_iter(n int) -> int
  var acc int;
{
  loop acc <- 1; n > 0; n <- n - 1 {
    acc <- n * acc;
  }

  return acc;
}

#| Recursive factorial implementation |#
proc fact_rec(n int) -> int {
  if n <= 0 {
    return 1;
  } else {
    return n * fact_rec(n - 1);
  }
}
```

```

#| Main procedure |#
proc main()
  var f1, f2, i int;
{
  loop {
    print("=== Factorial ===");

    i <- read("Type value calculate factorial");

    f1 <- fact_iter(i);
    f2 <- fact_rec(i);

    print("Iterative answer", f1);
    print("Recursive answer", f2);
  }
}

```

Figure 5.1: Generated IR code

```

A duck fact.duck
0  ERA      4      0      0      4      0      0
1  SET      _fact_iter      a0000002      0      0      0
2  SET      "=== Factorial ==="      d0000000      0      0      0
3  SET      5      a0000000      0      0      0
4  SET      0      a0000001      0      0      0
5  SET      _fact_rec      a0000002      0      0      0
6  SET      "Type value calculate factorial"      d0000001      0      0      0
7  SET      "Iterative answer"      d0000002      0      0      0
8  SET      "Recursive answer"      d0000003      0      0      0

0  ERA      5      0      0      5      0      0
1  GOPROC   0      0      0      0      0      0
2  ASD      1      0      0      0      0      0
3  GRE      n      0      0      0      0      0
4  JMT      t0      9      0      0      0      0
5  JMP      12      0      0      0      0      0
6  SUB      n      1      t1      20000000      a0000000      20000002
7  ASD      t1      0      0      0      0      0
8  JMP      3      0      0      0      0      0
9  MUL      n      acc      12      20000000      20000001      20000003
10 ASD      t2      acc      20000003      0      20000001
11 JMP      0      0      0      0      0      0
12 RETURN   acc      _fact_iter      20000001      0      a0000002
13 ENDPROC
14 LTO      n      0      0      0      0      0
15 JMT      t0      18      0      0      0      0
16 RETURN   1      _fact_rec      a0000000      0      20000003
17 JMP      25      0      0      0      0      0
18 ERA      4      0      0      4      0      0
19 SUB      n      1      t1      20000000      a0000000      20000001
20 PARAM   t1      20000000      20000001      0      20000000
21 GOPROC   _fact_rec      t2      0      0      0
22 ASD      0      0      0      0      0      0
23 MUL      n      t2      13      20000000      20000002      20000003
24 RETURN   t3      _fact_rec      20000003      0      a0000003
25 ENDPROC
26 PRINTLN  "=== Factorial ==="      0      0      0      0      0
27 PRINTLN  "Type value calculate factorial"      0      0      0      0      0
28 READ      1      0      0      0      0      0
29 ERA      4      0      0      4      0      0
30 PARAM   1      20000000      20000000      0      20000003
31 GOPROC   _fact_iter      fact_iter      0      0      0
32 ASD      t0      18      0      0      0      0
33 ASD      f1      20000001      0      20000002
34 ERA      4      0      0      4      0      0
35 PARAM   1      20000000      20000000      0      20000000
36 GOPROC   _fact_rec      fact_rec      0      0      0
37 ASD      t1      a0000003      0      20000003
38 ASD      t2      20000003      0      20000004
39 PRINT    "Iterative answer"      0      0      0      0      0
40 PRINTLN  f1      0      0      0      0      0
41 PRINT    "Recursive answer"      0      0      0      0      0
42 PRINTLN  t2      0      0      0      0      0
43 JMP      26      0      0      0      0      0
44 ENDPROC

```

Program is valid

Figure 5.2: Program output

```
λ duck run fact-quick
== factorial ==
Type value calculate factorial
4
Iterative answer 1
Recursive answer 1
== factorial ==
Type value calculate factorial
1
Iterative answer 1
Recursive answer 1
== factorial ==
Type value calculate factorial
2
Iterative answer 2
Recursive answer 2
== factorial ==
Type value calculate factorial
3
Iterative answer 6
Recursive answer 6
== factorial ==
Type value calculate factorial
4
Iterative answer 24
Recursive answer 24
== factorial ==
Type value calculate factorial
5
Iterative answer 120
Recursive answer 120
== factorial ==
Type value calculate factorial
6
Iterative answer 720
Recursive answer 720
== factorial ==
Type value calculate factorial
7
Iterative answer 5040
Recursive answer 5040
== factorial ==
Type value calculate factorial
8
Iterative answer 40320
Recursive answer 40320
== factorial ==
Type value calculate factorial
9
Iterative answer 362880
Recursive answer 362880
== factorial ==
Type value calculate factorial
10
Iterative answer 3628800
Recursive answer 3628800
== factorial ==
Type value calculate factorial
11
λ
```

5.1.2 Fibonacci

```
#| Iterative fibonacci implementation |#
proc fib_iter(n int) -> int
  var f0, f1, tmp int;
{
  f0 <- 0;
  f1 <- 1;

  loop ; n > 0; n <- n - 1 {
    tmp <- f1;
    f1 <- f0 + f1;
    f0 <- tmp;
  }

  return f0;
}
```

```

#| Recursive fibonacci implementation |#
proc fib_rec(n int) -> int {
  if n <= 0 {
    return 0;
  } else if n = 1 {
    return 1;
  } else {
    return fib_rec(n - 1) + fib_rec(n - 2);
  }
}

#| Main procedure |#
proc main()
  var f1, f2, i int;
{
  loop {
    print("=== Fibonacci ===");

    i <- read("Type value of the nth fibonacci to calculate");

    f1 <- fib_iter(i);
    f2 <- fib_rec(i);

    print("Iterative answer", f1);
    print("Recursive answer", f2);
  }
}

```


Figure 5.3: Generated IR code

```

0  ERA      5      0      0      5      0      0
1  GOPROC   0      0      0      0      0      26
2  ASO      0      0      0      0      0      0
3  ASO      1      1      1      0      0      0
4  GRE      n      0      0      18      0      0
5  JMT      t0     0      0      0      0      0
6  JMP      15     0      0      0      0      0
7  SUB      n      1      1      1      0      0
8  ASO      t1     0      0      0      0      0
9  JMP      4      0      0      0      0      0
10 ASO      f1     0      0      0      0      0
11 ADD      f0     f1      0      0      0      0
12 ASO      t2     1      1      0      0      0
13 ASO      tmp    f0     0      0      0      0
14 JMP      7      0      0      0      0      0
15 RETURN   f0     _fib_iter 0      0      0
16 ENDPROC
17 LEO      n      0      0      18      0      0
18 JMT      t0     0      0      0      0      0
19 RETURN   0      _fib_rec 0      0      0
20 JMP      37     0      0      0      0      0
21 EQ      n      1      1      0      0      0
22 JMT      t1     0      0      0      0      0
23 RETURN   1      _fib_rec 0      0      0
24 JMP      37     0      0      0      0      0
25 ERA      6      0      2      0      0      0
26 SUB      n      1      1      0      0      0
27 PARAM    t2     20000000 20000001 0      0      0
28 GOROC    _fib_rec 0      0      0      0      0
29 ASO      _fib_rec t2     0      0      0      0
30 ERA      6      0      2      0      0      0
31 SUB      n      2      1      0      0      0
32 PARAM    t4     20000000 20000003 0      0      0
33 GOROC    _fib_rec t4     0      0      0      0
34 ASO      _fib_rec t5     0      0      0      0
35 ADD      t0     t5     0      0      0
36 RETURN   t0     _fib_rec 0      0      0
37 ENDPROC
38 PRINTLN  "=== Fibonacci ===" | 0      0      0
39 PRINTLN  "Type value of the nth fibonacci to calculate" | 0      0      0
40 READ
41 ERA      6      0      1      0      0      0
42 PARAM    t1     20000000 20000001 0      0      0
43 GOROC    _fib_iter t1     0      0      0      0
44 ASO      _fib_iter t0     0      0      0      0
45 ASO      t0     t1     0      0      0
46 ERA      6      0      2      0      0      0
47 PARAM    t1     20000000 20000003 0      0      0
48 GOROC    _fib_rec t1     0      0      0      0
49 ASO      _fib_rec t2     0      0      0      0
50 ASO      t1     t2     0      0      0
51 PRINT
52 PRINTLN  "Iterative answer" | 0      0      0
53 PRINT
54 PRINTLN  "Recursive answer" | 0      0      0
55 JMP      38     0      0      0
56 ENDPROC
Program is valid

```

Figure 5.4: Program output

```

λ duck run fib.quick
== Fibonacci ==
Type value of the nth fibonacci to calculate
0
Iterative answer 0
Recursive answer 0
== Fibonacci ==
Type value of the nth fibonacci to calculate
1
Iterative answer 1
Recursive answer 1
== Fibonacci ==
Type value of the nth fibonacci to calculate
2
Iterative answer 1
Recursive answer 2
== Fibonacci ==
Type value of the nth fibonacci to calculate
3
Iterative answer 2
Recursive answer 3
== Fibonacci ==
Type value of the nth fibonacci to calculate
4
Iterative answer 3
Recursive answer 5
== Fibonacci ==
Type value of the nth fibonacci to calculate
5
Iterative answer 5
Recursive answer 8
== Fibonacci ==
Type value of the nth fibonacci to calculate
7
Iterative answer 13
Recursive answer 21
== Fibonacci ==
Type value of the nth fibonacci to calculate
9
Iterative answer 34
Recursive answer 55
== Fibonacci ==
Type value of the nth fibonacci to calculate
10
Iterative answer 55
Recursive answer 89
== Fibonacci ==
Type value of the nth fibonacci to calculate
10
λ

```

5.1.3 Find

```
#| Main procedure |#
proc main()
  var array [5]int;
  var x, i, size int;
  var exists bool;
{
  size <- 5;

  print("=== Array ===");

  loop i <- 0; i < size; i <- i + 1 {
    array[i] <- read("Type value at position", i);
  }

  print("=== Value to find ===");

  x <- read("Type value to find");

  loop i <- 0; i < size; i <- i + 1 {
    if x = array[i] {
      exists <- true;
      break;
    }
  }

  print("=== Result ===");

  if exists {
    print("Value", x, "was found in array at position", i);
  } else {
    print("Value", x, "was not found in array");
  }
}
```

Figure 5.5: Generated IR code

```

3  SET "was not found in array" | 00000007 | 0 0 0 0
4  SET "=== Array ===" | 00000000 | 0 0 0 0
5  SET "Type value to find" | 00000003 | 0 0 0 0
6  SET 0 | 00000000 | 0 0 0 0
7  SET "was found in array at position" | 00000000 | 0 0 0 0
8  SET 536870912 | 00000002 | 0 0 0 0
9  SET "Type value at position" | 00000002 | 0 0 0 0
10 SET #t | 00000000 | 0 0 0 0
11 SET 1 | 00000002 | 0 0 0 0
12 SET 0 | 00000001 | 0 0 0 0
13 SET "Value" | 00000005 | 0 0 0 0

0  EDA 12 0 4 | c 0 4
1  GOPROC 0 | main | 0 2
2  ASG 5 | size | 00000000 0 20000000
3  PRINTLN "=== Array ===" | 0 00000000
4  ASG 0 | 1 | 00000001 0 20000000
5  LES 1 | size | 20000000 20000000 40000000
6  JMT t0 | 11 | 40000000 0 5
7  JMP 17 | 0 11
8  ADD i | 1 | t1 | 20000000 00000002 20000007
9  ASG t1 | 1 | 20000007 0 20000000
10 JMP 18 | 0 5
11 ASSERT i | 0 5 | 20000000 0 00000000
12 ADD 1 | 536870912 | 12 | 20000000 00000003 12000000
13 PRINT "Type value at position" | 0 0 00000004
14 PRINTLN 1 | 0 0 20000000
15 READ t2 | 0 0 12000000
16 JMP 18 | 0 0
17 PRINTLN "=== Value to find ===" | 0 0 00000002
18 PRINTLN "Type value to find" | 0 0 00000003
19 READ x | 0 20000007
20 ASG 0 | 1 | 00000001 0 20000000
21 LES 1 | size | 20000000 20000000 40000001
22 JMT t3 | 27 | 40000001 0 15
23 JMP 34 | 0 22
24 ADD i | 1 | t4 | 20000000 00000002 20000004
25 ASG t4 | 1 | 20000004 0 20000000
26 JMP 27 | 0 15
27 ASSERT i | 0 5 | 20000000 0 00000000
28 ADD 1 | 536870912 | 15 | 20000000 00000003 12000000
29 EQ x | t5 | t6 | 20000007 12000000 40000002
30 JMP t6 | 33 | 40000002 0 21
31 ASG #t | exists | 00000000 0 40000003
32 JMP 34 | 0 22
33 JMP 24 | 0 15
34 PRINTLN "=== Result ===" | 0 0 00000004
35 JNE exists | 41 | 40000003 0 25
36 PRINT "Value" | 0 0 00000005
37 PRINT x | 0 20000007
38 PRINT "was found in array at position" | 0 0 00000006
39 PRINTLN 1 | 0 20000000
40 JMP 44 | 0 20
41 PRINT "Value" | 0 0 00000005
42 PRINT 1 | 0 20000007
43 PRINTLN "was not found in array" | 0 0 0 00000007
44 ENDPROC 0 | 0 0

Program is valid

```

Figure 5.6: Program output

```

λ duck run find.quick
=== Array ===
Type value at position 0
403 Type value at position 1
402 Type value at position 2
82 Type value at position 3
92 Type value at position 4
3
=== Value to find ===
Type value to find
0
=== Result ===
Value 0 was not found in array
λ duck run find.quick
=== Array ===
Type value at position 0
234 Type value at position 1
82 Type value at position 2
42 Type value at position 3
22 Type value at position 4
83
=== Value to find ===
Type value to find
22
=== Result ===
Value 22 was found in array at position 3
λ

```

5.1.4 Sort

```
#| Main procedure |#
proc main()
  var array [5]int;
  var i, j, tmp, size int;
  var is_sorted bool;
{
  size <- 5;

  print("=== Fill array ===");

  loop i <- 0; i < size; i <- i + 1 {
    array[i] <- read("Type value at position", i);
  }

  #| Bubble sort |#
  loop i <- 0; i < size - 1; i <- i + 1 {
    is_sorted <- true;

    loop j <- i + 1; j < size; j <- j + 1 {
      if array[i] > array[j] {
        tmp <- array[i];
        array[i] <- array[j];
        array[j] <- tmp;
        is_sorted <- false;
      }
    }

    if is_sorted {
      break;
    }
  }

  print("=== Sorted array ===");
  print(array);
}
```

Figure 5.7: Generated IR code

```

4  JMT      t0      11 | 40000000 0 0
7  JMP      17      | 0 0 11
8  ADD      1 1 11 | 20000000 a0000002 20000007
9  ASG      t1 1 1 | 20000007 0 20000006
10 JMP      8 0 5 | 0 0 5
11 ASSERT   1 0 5 | 20000006 a0000003 120000003
12 ADD      1 536878912 t2 | 20000006 a0000003 120000003
13 PRINT
14 PRINTLN  "Type value at position" | 0 0 d0000001
15 READ
16 JMP      12 0 8 | 0 0 120000008
17 ASG      0 1 1 | a0000001 0 20000006
18 SUB      size 1 t3 | 20000005 a0000002 20000009
19 LES      1 t3 14 | 20000006 20000007 40000001
20 JMP      t4 25 | 40000001 0 12
21 JMP      14 0 38 | 0 0 38
22 ADD      1 1 15 | 20000006 a0000002 20000008
23 ASG      t5 1 1 | 20000006 0 20000006
24 JMP      18 0 12 | 0 0 12
25 ASG      #t is_sorted c0000000 | c0000000 0 40000002
26 ADD      1 1 16 | 20000006 a0000002 20000008
27 ASG      t6 1 1 | 20000006 0 20000006
28 LES      1 size 17 | 20000006 20000005 40000003
29 JMT      t7 34 | 40000003 0 22
30 JMP      18 0 38 | 0 0 38
31 ADD      1 1 18 | 20000006 a0000002 20000008
32 ASG      t8 1 1 | 20000006 0 20000006
33 JMP      28 0 16 | 0 0 16
34 ASSERT   1 0 5 | 20000006 a0000003 120000003
35 ADD      1 536878912 t9 | 20000006 a0000003 120000003
36 ASSERT   1 0 5 | 20000006 a0000003 120000007
37 ADD      1 536878912 t10 | 20000006 a0000003 120000007
38 GRE      t9 t10 t11 | 120000006 120000007 40000004
39 JNE      t11 52 0 34 | 40000004 0 34
40 ASSERT   1 0 5 | 20000006 0 a0000000
41 ADD      1 536878912 t12 | 20000006 a0000003 120000009
42 ASG      t12 tmp 120000010 | 0 20000011
43 ASSERT   1 0 5 | 20000006 0 a0000000
44 ADD      1 536878912 t13 | 20000006 a0000003 120000012
45 ASSERT   1 0 5 | 20000006 0 a0000000
46 ADD      1 536878912 t14 | 20000006 a0000003 120000013
47 ASG      t14 t13 120000013 | 0 120000012
48 ASSERT   1 0 5 | 20000006 0 a0000000
49 ADD      1 536878912 t15 | 20000006 a0000003 120000014
50 ASG      tmp t15 20000011 | 0 12000014
51 ASG      #t is_sorted c0000001 | 0 40000002
52 JMP      31 0 17 | 0 0 17
53 JNE      is_sorted 55 0 37 | 0 0 37
54 JMP      22 0 16 | 0 0 16
55 JMP      22 0 16 | 0 0 16
56 PRINTLN  "=== Sorted array ===" | 0 0 0 d0000002
57 PRINT   array | 0 0 20000000
58 PRINT   array | 0 0 20000001
59 PRINT   array | 0 0 20000002
60 PRINT   array | 0 0 20000003
61 PRINTLN  array | 0 0 20000004
62 ENDPROC
Program is valid

```

Figure 5.8: Program output

```

λ duck run sort_quick
=== fill array ===
Type value at position 0
0
Type value at position 1
53
Type value at position 2
56
Type value at position 3
2
Type value at position 4
9
=== Sorted array ===
2 9 24 53 56
λ duck run sort_quick
=== fill array ===
Type value at position 0
29
Type value at position 1
40
Type value at position 2
2
Type value at position 3
7
Type value at position 4
44
=== Sorted array ===
2 7 29 40 44
λ duck run sort_quick
=== fill array ===
Type value at position 0
10
Type value at position 1
2
Type value at position 2
14
Type value at position 3
8
Type value at position 4
8
=== Sorted array ===
-2 0 8 10 14
λ

```

5.1.5 Matrix multiplication

```
#| Main procedure |#

proc main()
  var a [2][2]int;
  var b [2][2]int;
  var c [2][2]int;
  var i, j, k, I, J, K int;
{
  I <- 2;
  J <- 2;
  K <- 2;

  print("=== Fill matrices ===");

  loop i <- 0; i < I; i <- i + 1 {
    loop k <- 0; k < K; k <- k + 1 {
      a[i][k] <- read("Type value at a[" , i, "][" , k, " ");
    }
  }

  loop k <- 0; k < K; k <- k + 1 {
    loop j <- 0; j < J; j <- j + 1 {
      b[k][j] <- read("Type value at b[" , k, "][" , j, " ");
    }
  }

  loop i <- 0; i < I; i <- i + 1 {
    loop j <- 0; j < J; j <- j + 1 {
      loop k <- 0; k < K; k <- k + 1 {
        c[i][j] <- c[i][j] + a[i][k] * b[k][j];
      }
    }
  }

  print("=== Result ===");
  print(c);
}
```

Figure 5.9: Generated IR code

```

56 PRINTLN      "j" | 0 | 0 | 20000003
57 READ        | 113 | 0 | 0 | 12000001b
58 JMP         | 44 | 0 | 0 | 2c
59 JMP         | 37 | 0 | 0 | 25
60 ASG         0 | 1 | 1 | a0000001 | 0 | 20000007
61 LES         i | 154 | 20000007 | 2000000c | 40000004
62 JMT        114 | 67 | 40000004 | 0 | 43
63 JMP         | 189 | 0 | 0 | 4b
64 ADD         i | 115 | 20000007 | a0000002 | 2000001c
65 ASG        115 | 1 | 2000001c | 0 | 20000007
66 JMP         | 61 | 0 | 0 | 5d
67 ASG         0 | 3 | 3 | a0000001 | 0 | 20000017
68 LES         j | 116 | 20000017 | 2000000d | 40000005
69 JMT        110 | 74 | 40000005 | 0 | 4a
70 JMP         | 306 | 0 | 0 | 5e
71 ADD         j | 117 | 20000017 | a0000002 | 2000001d
72 ASG        117 | 3 | 2000001d | 0 | 20000017
73 JMP         | 68 | 0 | 0 | 44
74 ASG         0 | k | k | a0000001 | 0 | 20000011
75 LES         k | 118 | 20000011 | 2000000a | 40000006
76 JMT        110 | 81 | 40000006 | 0 | 51
77 JMP         | 308 | 0 | 0 | 59
78 ADD         k | 119 | 20000011 | a0000002 | 2000001a
79 ASG        119 | k | 2000001a | 0 | 20000011
80 JMP         | 78 | 0 | 0 | 4b
81 ASSERT      i | 0 | 2 | 20000007 | 0 | a0000000
82 MUL         i | 2 | 120 | 20000007 | a0000000 | 2000001f
83 ASSERT      j | 0 | 2 | 20000017 | 0 | a0000000
84 ADD        120 | 121 | 2000001f | 20000017 | 20000020
85 ADD        121 536870920 | 122 | 20000020 | a0000005 | 120000021
86 ASSERT      i | 0 | 2 | 20000007 | 0 | a0000000
87 MUL         i | 2 | 123 | 20000007 | a0000000 | 20000022
88 ASSERT      j | 0 | 2 | 20000017 | 0 | a0000000
89 ADD        123 | 124 | 20000022 | 20000017 | 20000023
90 ADD        124 536870920 | 125 | 20000023 | a0000005 | 120000024
91 ASSERT      i | 0 | 2 | 20000007 | 0 | a0000000
92 MUL         i | 0 | 126 | 20000007 | a0000000 | 20000025
93 ASSERT      k | 0 | 2 | 20000011 | 0 | a0000000
94 ADD        126 | k | 127 | 20000025 | 20000011 | 20000026
95 ADD        127 536870912 | 128 | 20000026 | a0000003 | 120000027
96 ASSERT      k | 0 | 2 | 20000011 | 0 | a0000000
97 MUL         k | 2 | 129 | 20000011 | a0000000 | 20000028
98 ASSERT      i | 0 | 2 | 20000017 | 0 | a0000000
99 ADD        129 | j | 130 | 20000028 | 20000017 | 20000029
100 ADD        130 536870916 | 131 | 20000029 | a0000004 | 12000002a
101 MUL        131 | 132 | 120000027 | 12000002a | 2000002b
102 ADD        125 | 132 | 133 | 12000002a | 2000002b | 2000002c
103 ASG        133 | 132 | 132 | 2000002c | 0 | 120000021
104 JMP         | 78 | 0 | 0 | 4a
105 JMP         | 71 | 0 | 0 | 47
106 JMP         | 64 | 0 | 0 | 4b
107 PRINTLN      "=== Result ===" | 0 | 0 | 0 | 20000005
108 PRINT        c | 5 | 0 | 0 | 20000005
109 PRINTLN      c | 0 | 0 | 0 | 20000007
110 PRINT        c | 0 | 0 | 0 | 2000000a
111 PRINTLN      c | 0 | 0 | 0 | 2000000b
112 ENDPROC
Program is valid

```

Figure 5.10: Program output

```

λ duck run mat_mul.quick
=== fill matrices ===
Type value at a[ 0 ][ 0 ]
0
Type value at a[ 0 ][ 1 ]
0
Type value at a[ 1 ][ 0 ]
0
Type value at a[ 1 ][ 1 ]
1
Type value at b[ 0 ][ 0 ]
0
Type value at b[ 0 ][ 1 ]
0
Type value at b[ 1 ][ 0 ]
1
Type value at b[ 1 ][ 1 ]
1
Type value at c[ 1 ][ 1 ]
0
=== Result ===
5
3
2
λ duck run mat_mul.quick
=== fill matrices ===
Type value at a[ 0 ][ 0 ]
0
Type value at a[ 0 ][ 1 ]
0
Type value at a[ 1 ][ 1 ]
1
Type value at b[ 0 ][ 0 ]
0
Type value at b[ 0 ][ 1 ]
1
Type value at b[ 1 ][ 0 ]
0
Type value at b[ 1 ][ 1 ]
1
Type value at c[ 1 ][ 1 ]
0
22 36
17 30
λ

```

5.1.6 Built-in procedures

```
proc main()
  var data [5]float;
  var pi, ans1, ans2, ans3 float;
{
  pi <- 3.14159;

  print("=== Trigonometric functions ===");

  ans1 <- sin(pi / 2);
  ans2 <- asin(ans1);
  print("sin(pi / 2) =", ans1, "asin(sin(pi / 2) =", ans2);

  ans1 <- cos(0);
  ans2 <- acos(ans1);
  print("cos(0) =", ans1, "acos(cos(0) =", ans2);

  ans1 <- tan(pi / 4);
  ans2 <- atan(ans1);
  ans3 <- atan2(1, 1);
  print("tan(pi / 4) =", ans1);
  print("atan(tan(pi / 4)) =", ans2, "atan2(1, 1) =", ans3);

  print("");
  print("=== Transcendental functions ===");

  ans1 <- exp(1);
  print(ans1);

  ans1 <- ln(ans1);
  print("ln(exp(1) =", ans1);

  ans1 <- log(125, 5);
  print("log(125, 5) =", ans1);

  print("");
  print("=== Other functions ===");

  ans1 <- pow(3, 5);
  print("pow(3, 5) =", ans1);

  ans1 <- sqrt(2);
  print("sqrt(2) =", ans1);

  ans1 <- mod(41, 3);
```



```

print("mod(41, 3) =", ans1);

ans1 <- abs(-12);
print("abs(-12) =", ans1);

ans1 <- ceil(1.5);
print("ceil(1.5) =", ans1);

ans1 <- floor(1.5);
print("floor(1.5) =", ans1);

print("");
print("=== Vectorial functions ===");

data[0] <- pi;
data[1] <- exp(1);
data[2] <- sqrt(2);
data[3] <- (1 + sqrt(5)) / 2;
data[4] <- 2;

print("vector = [", data, "]");

ans1 <- mean(data);
ans2 <- median(data);
ans3 <- mode(data);

print("mean =", ans1);
print("median =", ans2);
print("mode =", ans3);
}

```

Figure 5.11: Generated IR code

```

88 ASG t14 ans1 | 20000000 0 0 30000000
89 PRINT "mod(41, 3) =" | 0 0 0000000f
90 PRINTLN ans1 | 0 0 30000000
91 ASG -12 t15 | a0000000 0 0 30000017
92 ASG t15 ans1 | 30000017 0 0 30000000
93 PRINT "abs(-12) =" | 0 0 00000010
94 PRINTLN ans1 | 0 0 30000000
95 CELL t16 | b0000001 0 30000010
96 ASG t16 ans1 | 30000010 0 30000000
97 PRINT "ceil(1.5) =" | 0 0 00000011
98 PRINTLN ans1 | 0 0 30000000
99 FLOOR 1.5 t17 | b0000001 0 30000010
100 ASG t17 ans1 | 30000010 0 30000000
101 PRINT "floor(1.5) =" | 0 0 00000012
102 PRINTLN ans1 | 0 0 30000000
103 PRINTLN "" | 0 0 00000000
104 PRINTLN "" | 0 0 00000013
105 ASSET "" Vectorial functions "" | 0 0 00000000
106 ADD 0 0 5 | a0000001 0 a0000005
107 ADD 0 885386368 t18 | a0000001 a0000007 120000001
108 ASG t18 | 30000005 0 120000001
109 ASSET 1 0 5 | a0000003 0 a0000005
110 ADD 1 885386368 t19 | a0000003 a0000007 120000002
111 EXP t19 | 30000003 0 30000010
112 ASG t20 | 30000010 0 120000002
113 ASSET 2 0 5 | a0000000 0 a0000005
114 ADD 2 885386368 t21 | a0000000 a0000007 120000003
115 SQRT t21 | a0000000 0 30000010
116 ASG t22 | 30000010 0 120000003
117 ASSET 3 0 5 | a0000000 0 a0000005
118 ADD 3 885386368 t23 | a0000000 a0000007 120000004
119 SQRT t23 | a0000000 0 30000010
120 ASG t24 | a0000000 0 30000010
121 ADD 1 174 t25 | a0000003 30000010 30000010
122 DIV t25 2 t26 | 30000010 a0000000 30000010
123 ASG t26 | 30000010 0 120000004
124 ASSET 4 0 5 | a0000002 0 a0000005
125 ADD 4 885386368 t27 | a0000002 a0000007 120000005
126 ASG t27 | a0000002 0 120000005
127 PRINTLN "" | 0 0 00000014
128 PRINT data | 0 0 30000000
129 PRINT data | 0 0 30000001
130 PRINT data | 0 0 30000002
131 PRINT data | 0 0 30000003
132 PRINT data | 0 0 30000004
133 PRINTLN "" | 0 0 00000015
134 MEAN data 5 t28 | 30000000 a0000005 30000017
135 ASG t28 ans1 | 30000017 0 30000005
136 MEDIAN data 5 t29 | 30000000 a0000005 30000020
137 ASG t29 ans2 | 30000020 0 30000000
138 MODE data 5 t30 | 30000000 a0000005 30000021
139 ASG t30 ans3 | 30000021 0 30000011
140 PRINT "mean =" | 0 0 00000016
141 PRINTLN ans1 | 0 0 30000000
142 PRINT "median =" | 0 0 00000017
143 PRINTLN ans2 | 0 0 30000000
144 PRINT "mode =" | 0 0 00000018
145 PRINTLN ans3 | 0 0 30000011
146 ENDPROC | 0 0 0

```

Program is valid
 1

Figure 5.12: Program output

```

λ duck run built-in.quick
== Trigonometric functions ==
sin(pi / 2) = 0.99999999999999991198 asin(sin(pi / 2)) = 1.5707956800000579
cos(0) = 1 acos(cos(0)) = 0
tan(pi / 4) = 0.9999986732809835 atan(tan(pi / 4)) = 0.7853981633974483
== Transcendental functions ==
2.718281828459045 ln(exp(1)) = 1
log(120, 6) = 3.8880000000000004
== Other functions ==
pow(5, 5) = 243
sqrt(2) = 1.4142135623730951
mod(41, 3) = 2
abs(-12) = 12
ceil(1.5) = 2
floor(1.5) = 1
== Vectorial functions ==
vector = [ 3.14159 2.718281828459045 1.4142135623730951 1.618033988749895 2 ]
mean = 2.174652879916407
median = 2
mode = 3.14159
λ

```

Chapter 6

Code Documentation

6.1 General Code Overview

The codebase, ommiting doc, examples and vim, has the following structure.

```
(main directory)
- BigDuck.g4
- Makefile
- README.md
- compiler.go
- main.go
- parser*
- run.go
- structs
  - enums.go
  - linked_list.go
  - memory.go
  - memory_mapper.go
  - memory_stack.go
  - queue.go
  - semantic_cube.go
  - stack.go
  - stack_pointer.go
  - symbol_table.go
  - tac.go
  - virtual_machine.go
- tac_gen.go
- tree_listener.go
```

Note The parser directory is generated by ANTLR, thus it will not be talk about, however all the details for lexical an syntax analyzer are there.

6.2 Modules Description

The order of files is given by their usage on an execution.

File name	Description
main.go	Is the starting point of the program, according to the given arguments it will determine to run the compiler or the virtual machine using the filename given.
compiler.go	Makes an instance of the tree listener and runs the ANTLR lexer and parser.
tree_listener.go	Makes an implementation of the grammar listener interface provided by ANTLR, it contains a method for entering and exiting a grammar rule. All compilation actions are embedded in this methods.
tac_gen	For any non-trivial or repetitive action for generation of TAC, there is an implementation of methods, for tree_listener, in this file.
run.go	Parsers the .quack file and builds a TAC array to be given to the virtual machine, and then runs it.
structs	Contains all the data structures used for compilation and execution. I will not go through each file since they are self explanatory.

6.3 Code excerpts

On the following section, there some excerpts of code with some descriptions.

6.3.1 GenerateOpTac

```
func (l *BigDuckListener) GenerateOpTAC(pointer int) {
```

The signature of this method is read as a method for a pointer to a BigDuckListener, this means that there is mutability on the struct. The pointer int argument indicates if there is an allocation for a new pointer and which argument of the operation is the pointer.

```
    var args [3]string
    var types [3]int
    var i, argc int

    op := l.PopOp()

    is_unary, _ := structs.IsUnaryOp[op]

    if is_unary {
        argc = 0
    } else {
        argc = 1
    }
```

Retrieving basic information for TAC generation; operator, argument count, if it is a unary operator.

```
    for i = argc; i >= 0; i-- {
        item, _ := l.argstack.Pop()
        args[i], _ = item.(string)
        item, _ = l.typestack.Pop()
        types[i], _ = item.(int)
    }
```

Getting arguments and types from argstack and typestack.

```

if structs.Cube[op][types[0]][types[argc]] == structs.Error_t {
    l.valid = false;
    fmt.Printf("line %d:%d type error mismatch\n", l.curr_line, l.curr_col)
} else if op != structs.ASG {
    types[2] = structs.Cube[op][types[0]][types[argc]]
    l.typestack.Push(types[2])
}

```

Check with semantic cube, and if operator not assignment (meaning there is a temporal variable generated) the push the type of the result into typestack.

```

if op == structs.ASG {
    args[2] = args[0]
    args[0] = args[1]
    args[1] = ""
    types[2] = types[0]
    types[0] = types[1]
    types[1] = 0
} else {
    args[2] = "t" + strconv.Itoa(l.tmpc)
    l.argstack.Push(args[2])
    l.tmpc++
}

```

Because the target address on assignment is on the retrieved variables, then it necessary to make some adjustments to the order of the arguments. Otherwise create a new temporal and push it into argstack.

```

var address [3]int

for i = 0; i < 3; i++ {
    scope, _, exists := l.symtable.Lookup(args[i])

    if exists || (len(args[i]) > 0 && args[i][0] == 't') {
        if i + 1 == pointer {
            l.memmap.RegisterPointer(scope, args[i])
        }

        address[i] = l.memmap.GetAddress(scope, args[i], types[i])

    } else if len(args[i]) > 0 {
        address[i] = l.memmap.GetAddress(structs.Global, args[i], types[i])
    }
}

```

For each argument, test if it is variable or temporal then, if it matches with pointer allocate the pointer and lastly get its address, else register as global variable and get address.

```

l.ir_code = append(
    l.ir_code,
    structs.Tac{
        Op: op,
        Args: args,
        Address: address})
l.pc++
}

```

Lastly create a new TAC, append it to the array of TACs, and increment program counter.

6.3.2 Memory management methods

```
func (m *memory) InitGlobal(ic, fc, bc int) {
    m.Strings = make(map[int]string, ic)
    m.MemI[Global] = make([]int, ic)
    m.MemF[Global] = make([]float64, fc)
    m.MemB[Global] = make([]bool, bc)
    m.framesize = make(map[int]int)
    m.Sp = make(map[int]int)
}
```

Create slices (Go's name for dynamic arrays) for each global memory pool per type, and initialize framesizes and stack pointers.

```
func (m *memory) InitLocal(ic, fc, bc int) {
    m.MemI[Local] = append(m.MemI[Local], make([]int, ic)...)
    m.MemF[Local] = append(m.MemF[Local], make([]float64, fc)...)
    m.MemB[Local] = append(m.MemB[Local], make([]bool, bc)...)

    m.memstack.Push(ic, fc, bc)
}
```

Increment memory pools' capacity and push allocated memory into memory stack to keep track of memory used in procedure calls.

```
func (m *memory) PushContext() {
    m.Sp[Int_t] += m.framesize[Int_t]
    m.Sp[Float_t] += m.framesize[Float_t]
    m.Sp[Bool_t] += m.framesize[Bool_t]

    ic, fc, bc := m.memstack.Top()
    m.framesize[Int_t] = ic
    m.framesize[Float_t] = fc
    m.framesize[Bool_t] = bc
}
```

Increment stack pointers by current frame size, and update framesizes.


```

func (m *memory) PopContext() {
    ic, fc, bc := m.memstack.Pop()

    m.MemI[Local] = append(
        []int(nil), m.MemI[Local][:len(m.MemI[Local]) - ic]...)
    m.MemF[Local] = append(
        []float64(nil), m.MemF[Local][:len(m.MemF[Local]) - fc]...)
    m.MemB[Local] = append(
        []bool(nil), m.MemB[Local][:len(m.MemB[Local]) - bc]...)

    ic, fc, bc = m.memstack.Top()

    m.framesize[Int_t]      = ic
    m.framesize[Float_t]    = fc
    m.framesize[Bool_t]     = bc

    m.Sp[Int_t]             -= m.framesize[Int_t]
    m.Sp[Float_t]           -= m.framesize[Float_t]
    m.Sp[Bool_t]            -= m.framesize[Bool_t]
}

```

Shrink memory pools, and restore framesizes and stack pointers.

Part II

User Manual

Chapter 7

A Quick Tour

7.1 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.

7.2 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 use 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.

7.3 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.

7.3.1 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 until another assignment is performed.

7.3.2 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 |#
}
```

7.3.3 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 subtraction, 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 it is **not** equal to `a + (b - c)`.

7.4 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 and assignment, therefore no operation can be immediately 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 when the if clause condition 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");
  }
}
```

7.5 Loop Statements

On any computational language exists the notion of *loops*. The looping mechanism allow us to concisely tell the computer to perform some operations n amount of times. Otherwise we would have to sequence n amount of times the same instruction and this would not be managable on the long run (also some computations explicitly require a looping mechanism).

7.5.1 Infinite Loop

The easiest way to loop instructions is shown by the next example.

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

If you run this program you will see that it never ends and it is going to fill the console with “Hello, World!”, to stop the program type `ctrl + c`. The infinite loop is useful for interactive programs, where it is up to the user when to end the program.

There are other mechanisms to stop an infinite loops, they will be covered in depth on Subsection 7.5.5.

7.5.2 While Loop

Most of the times it is desired to end a loop when a condition is met, thus the BigDuck language provide syntax for handling this kind of loops. A while loop is a type of loop that continues until the loop condition is false.

```
proc main()
  var i int;
{
  i <- 1;

  loop i < 10 {
    print("i value is", i);
    i <- i + 1;
  }
}
```

The previous program prints all numbers starting from 1 up to 9, 10 will not be printed because the condition `i < 10` is no longer true.

7.5.3 For Loop

Many times when looping, there is going to be a control variable, for example on the previous program it was `i`. Since this is really common there is and special syntax for this kind of loop. The following program is equivalent to the previous one, however is using the for loop syntax.

```
proc main()
  var i int;
{
  loop i <- 1; i < 10; i <- i + 1 {
    print("i value is", i);
  }
}
```

As you can it is much more compact and explicit on how the loop will behave. The first statements is an assignment to initialize the control variable (is only performed once), the second statement is the loop condition, and the third statement is an assignment statement to modify the control variable.

At this point you may think that the while syntax is redundant against the for syntax. However this is not true, the recommendation is to use the while syntax when you are not sure how many iterations is going to take the loop. On the other side, for style syntax is best when you know beforehand how is the loop going to behave.

7.5.4 Do While Loop

The next type of loop does not have a dedicated syntax, however it is also a common type of loop present on many programming languages. Therefore here is provided an idiom to achieve the same goal.

```
proc main()
  var i int;
  var ok bool;
{
  loop ok <- false; not ok; ok <- i = 2 {
    i <- read("Type 2 to exit the loop");
  }
}
```

Do while loops are at least run one time, by assuming that it is not ok to exit the loop the program will enter inside the loop. And at every iteration it is tested whether the condition to exit is met.

7.5.5 Control Flow Statements

Throughout loops sometimes it is required to exit earlier or go to the next iteration. This could be achieved by using logic, however it can be messy and obfuscate the meaning of the code. Take a look at the following program.

```
proc main() {
  loop #| Exit condition |# {
    if #| Condition skip iterations|# {
      skip;
    }

    if #| Condition to break from loop |# {
      break;
    }
  }
}
```

When a skip statement is reached, the following code on the loop will be ignored and the next iteration is going to be reached. However the break statement exits the loop. These are useful to avoid unnecessary operations or to make the code more readable.

7.6 Procedures

At this point everything has been done inside the main procedure, this means that all code written in main belongs to the same context. With context it is meant that all variables and flow of the program is self contained in the procedure. This may be good enough for small programs, however as the complexity increases you start seeing code repetition or related code to perform an action.

Many programming languages, including BigDuck, are able to provide different contexts through; functions, procedures, methods, etc. On this language it is done through procedure, whose syntax is the following.

```
proc name(args) -> type
    vars
{
}
```

Where **name** is the procedure name, **args** is a list of arguments to be passed to a procedure, **type** indicates the return type of the procedure, and **vars** indicate local variables to be use.

The following are examples of procedures.

```
proc square(x float) -> float {
    return x * x;
}

proc distance(x1, y1, x2, y2 float) -> float {
    return sqrt(square(x2 - x1) - square(y2 - y1));
}

proc close_enough(x1, y1, x2, y2 float) -> bool {
    return distance(x1, y1, x2, y2) > 0.01;
}

proc get_circle_area(r float) -> float {
    return 3.14159 * square(r);
}
```

Using procedures is not only helpful to write less code, but it also helps to make the code more abstract (hiding unnecessary information), and it allows to handle problems with more ease.

The last thing to cover about procedures is recursion. Recursion can be seen as a higher level form of loops, this is because it covers all the use cases of loops and can be a cleaner solution. However sometimes may be harder to think in a recursive solution or viceverse, thus it remains at the taste of the user to decide when to use recursion or loops.

A classic use case for recursion is for the implementation of the factorial function. In mathematics the factorial function is defined the following way.

$$f : \mathbb{N} \rightarrow \mathbb{N} \Rightarrow f(x) = \begin{cases} 1 & \text{if } n \leq 0 \\ n \cdot f(n-1) & \end{cases}$$

The iterative approach (the one using loops) is not as easy to understand and make take sometime to realize that it computes the same function.

```
proc factorial(n int) -> int
  var prod int;
{
  prod <- 1;

  loop n > 0 {
    prod <- prod * n;
  }

  return prod;
}
```

However the recursive definition resembles better the mathematical definition.

```
proc factorial(n int) -> int {
  if n <= 0 {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

7.7 Tensorial Types

As your programs get more complex, you will notice that certain variables are related to other variables. Many programming languages offer many tools for handling complex relationships for data modelling.

However the BigDuck language provides the simplest kind of structured data called tensors, commonly referred as arrays or multidimensional arrays. The name was picked from mathematics since it generalizes the notion of an ordered tuple that can be indexed (known as vectors).

Look at the following program, which defines a 1-D tensor.

```
proc main()
  var a [5]int;
{
  a[1] <- 12;
  print(a[1])    #| prints: 12 |#
}
```

This syntax indicates the language that the variable **a** can be accessed by values from 0 and less than 5, therefore indexing by 5 produces an error. If no index was given, it will be given the first value of the tensor.

For higher dimensional tensors just keep adding more dimensions like here.

```
proc main()
  var a [2][5]int;
{
  a[0][1] <- 12;
  print(a[0][1])    #| prints: 12 |#
}
```

7.8 Built-in Procedures

Before ending this quick tour of the language, it will be presented the signatures for built-in functions. You may ask, aren't these also procedures? And you would be correct, the thing is that functions are a special subset of procedures, the property of functions is that the mapping between input and output is constant. In other words, functions will always give the same output for the same input, however with procedures this cannot be said.

Leaving terminology aside, the following are functions that are always available on the language, thus it is not possible to use these names for other procedures. These functions were chosen since they are the most commonly used for any kind of mathematical or scientific computing.

Trigonometric functions

```
proc sin(x float) -> float
proc asin(x float) -> float
proc cos(x float) -> float
proc acos(x float) -> float
proc tan(x float) -> float
proc atan(x float) -> float
proc atan2(x, y float) -> float
```

Transcendental functions

```
proc exp(x float) -> float
proc ln(x float) -> float
proc log(x, b float) -> float
```

Other functions

```
proc pow(x, y float) -> float
proc sqrt(x float) -> float
proc mod(n, b float) -> float
proc abs(x float) -> float
proc ceil(x float) -> float
proc floor(x float) -> float
```

Vectorial functions

```
proc mean(x []float) -> float
proc median(x []float) -> float
proc mode(x []float) -> float
```