CSPC 62
Compiler Design Project
Rajinican

106119045 | Hanan Abdul Jaleel

106119047 | Hari Rahul V

106119139 | Udith Kumar V

Table of Contents

Introduction

This project implements an interpreter, compiler and a simulator to check the results of the assembly file generated by the compiler. The interpreter was initially implemented for understanding the building of a syntax tree, as statements that require conditional execution cannot be implemented with an action and recursive return flow system like what yacc uses. An simulator was created along with a tiny ISA called xHUH assembly, so that the output generated by the compiler could be tested.

Languages and tools used

Lex - were used for the purpose of lexical analysis and generation of tokens.
Yacc - used for parsing the grammar and helps in constructing the parse tree.
Js - used to write the simulator for the intermediate code generated.
Node - a runtime for javascript code, used to run simulator.js
C++ - was used along with a few standard libraries for ease of use of the data structures.

TL;DR

- Implemented syntax tree creator for rajinican
- Implemented Interpreter for Rajinican using c++ data structures.
- Implemented Compiler which converts syntax tree into xHUH assembly.
- Created an ISA called xHUH for the compiler to compile code into.
- Created a behavioural implementation of a system that implements the given ISA.

Language and Grammar Design

```
statements -> statements statement
statement  ->   VECHUKO ( ID ) : // printing
             | VECHUKO ( STRING ) : // printing strings
             | expr SOLRAN ID SEIRAN:  // assignment of expr result to id
             | ORUVELA ( expr ) { statements } // if statement
             | ORUVELA ( expr ) { statements } ILLENA { statements }  //if
else
             | NA statement VAATI SONNA statement VAATI SONNA MAARI statement
{ statements } // for loop
             | ID ( params ) { statements } // function definition
             | ITHU EPIDI IRUKU ID : // return statement
             ;
expr       ->   expr operator expr
             | expr comparator expr
             | ( expr )
             | ID // variable
             | ID ( args ) // function call
             ;
```

As one can see the above is the grammar for the language commented out, more logically, we can list out the features of our language without focusing on the grammar.
We shall discuss the features in terms of three major types or blocks

Expr eval block
Jump block
Function block

Rajinican offers the following features:

Expr eval block:

Expr:
Expr op expr
Id
Assignment:
Id <- expr
Print:
Print (expr)
Declare:
Id

All the above statements constitute the absolute bare basics of our language. The above isn't enough to make our language turing complete, but is useful while thinking of as a unit when developing the project.

Jump block:

```
If:
        If (expr) { eval block } else { eval block }
For:
        For (eval block : eval block : eval block) { eval block }
```

Function Block:

```
Call:
        ID ( args )
Definition:
        ID ( params ) { eval block }
```

All programming languages can logically divide their features into blocks like above, usually modern languages have more blocks (not explicitly but it is dividable into blocks) for classes etc. and some additional helper boxes might be added to handle multiple data types.

Datatypes in Rajinican

Rajinican has only one type of variable and that is double. Strings do exist, but they are not mutable and can **only** be printed.

Implementation

We have implemented three programs in the process of developing a compiler for our language.

Interpreter
Compiler
Simulator

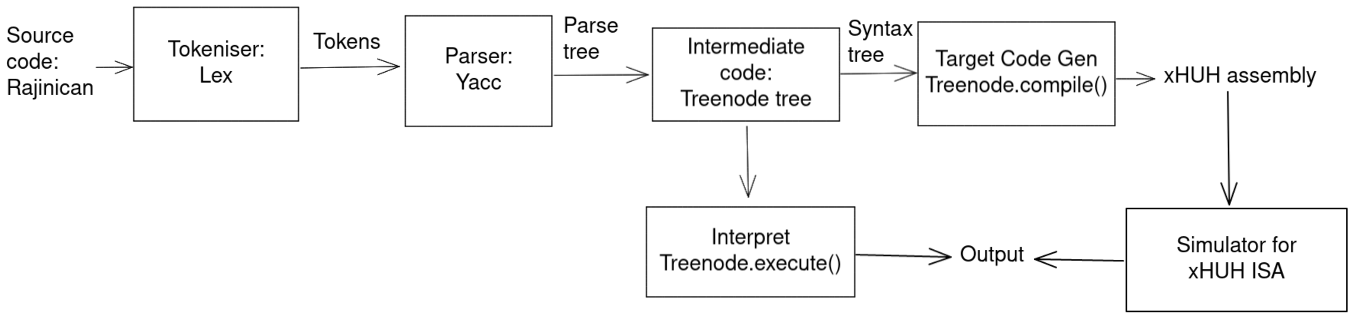


Fig: 1 Block diagram of implementation

In the above diagram the treenode.compile() block compiles the code into xHUH assembly. The treenode.execute()

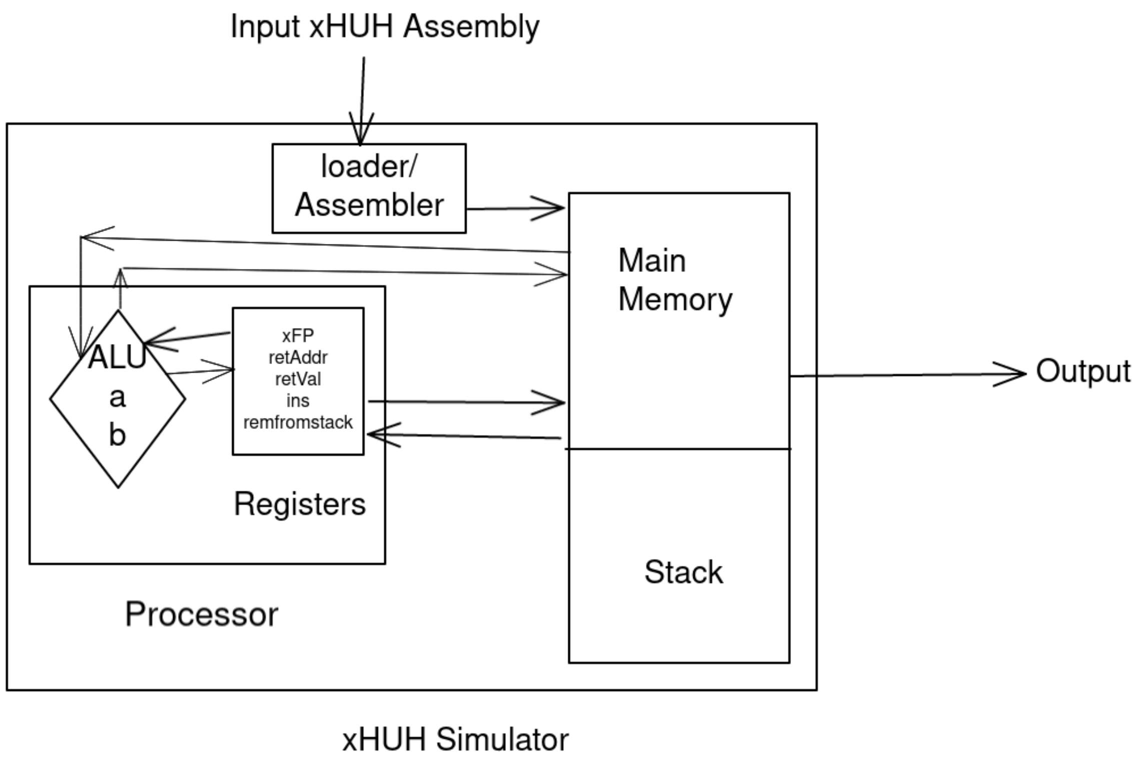


Fig:2 Explaining the simulator in detail

Both the interpreter and compiler first build the parse tree with the help of lex and yacc. The parse tree is then used to build a syntax tree which is used to interpret and compile into output and assembly.

The Simulator is an interesting part of the assignment. Just its existence means that the intermediate code we generated using the compiler, is not just intermediate code, but assembly code that runs on a machine (which is our simulator).

The simulator itself is not an entire machine with its own implementation of an ISA, but is an abstract machine that behaves as per the ISA used. Here we have made our own ISA. We have decided to call it xHUH (after Hanan Udith Hari).

And since the simulator does not run the machine code but runs assembly, it can be thought of as having an assembler and loader that places the program onto the memory.

One other way to see the assembly code is as intermediate bytecode, which even the likes of python generates, some unrealistic features (in the way how it’s handled) like string manipulation also make complete sense this was as python bytecode use

xHUH ISA

This assembly is a one address code format stack based ISA. The system is required to have a few registers as mentioned below, main memory to store variable values and a stack to operate on. The following is the code that implements this ISA.
Simulator.js
There are a few special registers in the ISA used to store values temporarily, and they are:

S.No	Register	Purpose
1.	xFP	Stores a pointer to the current frame beginning on the stack.
2.	retAddr	Temporarily stores return addresses when returning from a function.
3.	retval	Temporarily stores return value for a function when returning
4.	ins	Program counter keeps track of current instruction.
5.	remfromstack	Keeps track of the count of the arguments in a function for helping when popping off arguments during return

Eval block:
Every arithmetic operation pops from the stack to get the arguments and then computes the result and places the result back in the stack.

```
Add
Sub
Mul
Div
Mod
Lt
Gt
Leq
Geq
Eq
Neq
```

Variables are declared // memory is allocated on declaration, using the decl command

```
decl
```

Values are pushed into the stack from memory using the push command
Push ST is a special variation of push where it it takes the value at the top of the stack as index, and pushes stack[index] onto the stack. This addressing mode is useful and necessary for implementing functions.

```
Push
Push ST
```

Values are placed back in memory front the top of the stack using pop command

```
pop
```

Jump block

Both if and for statements will use jmps and labels in different ways

```
Jmp
Jeq
Jneq
label:
```

There is no need for a lot of explanation as it’s just jump statements jumping to labels.

Function block

Function block will implement 2 commands and we will see in detail the way the simulator works, and how the compiler will have to write code to deal with it.

```
Call
Ret
```

Our language has functions that can have arguments and return so we have to have mechanisms that handle all of that.
We will deal with functions in the following way.

- The function definition will follow a label with the function name, so one will have to jump to that label to run the function.
- The compiler, even **before** using the call command to go to the function label, will have to **push all the arguments passed onto stack** while calling.
- Now the call operation is used which would do the following things in the implementation.
    - Store the current frame pointer onto stack
    - Update the frame pointer to point to the top of stack right now.
    - Push return address onto stack.
    - Jump to function block
        - Inside the function block we first push the number of parameters the function uses onto the stack. This is very important and will

        be used to later remove the arguments that were pushed into the stack before the function call. This part is what helps our implementation of function truly work. As this is what allows for local variables to exist and hence allowed recursion.*
        - Push 0 onto stack for all the local variables declared inside the function. This will serve as the local variables
        - Execute rest of what's in the function label

- And when inside the function, returning should take care of cleaning up the stack and a few other things which are mentioned below.
    - Store the value of retval, retarder, remfromstack into their respective registers.
    - Change the fp to old fp by replacing it with the value that the current fp is pointing to.
    - Pop stack remfromstack number of times to remove all the pushes made for arguments before calling function.
    - Push retval onto stack
    - Jump to return addr

* Without that step, if we were to use memory location directly, the compiler will use the same memory location for all calls of the function and hence recursion will not exist. Also one will have to make changes to the compiler so as to use different names for variables inside the function as some sort of redeclaration might happen. But all that is not required as all local variables from now on will be referenced as offsets from the frame pointer. This solves all the issues and lets us use functions as we do in modern languages.

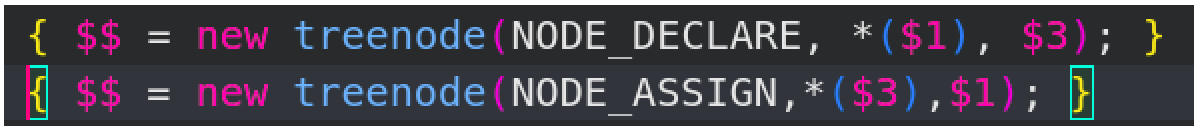
Here we have seen how the simulator handles the call and ret operations, but while writing the compiler we do not need to know all this, just knowing the following is enough

- Push arguments into stack
- Call function
- Push arg count
- Push 0 for each local variable in function
- Execute what is inside function block
- Return

Lex and Yacc

This is the most basic part of the implementation and is common to both the compiler and interpreter. Basically the parser.y file is used to make our own version of the tree. The syntax tree, where each node is of type treenode. The treenode and all related functions are in the tree.hpp header file.
Lexer.l
Parser.y
Tree.hpp

The syntax tree is actually being created in the parser.y file, as you can see the new treenode(NODE_TYPE, args) statement. As the parser returns the $ value one can see how a tree will be made as the parser parses the code.



```
{ $$ = new treenode(NODE_DECLARE, *($1), $3); }
{ $$ = new treenode(NODE_ASSIGN,*($3),$1); }
```

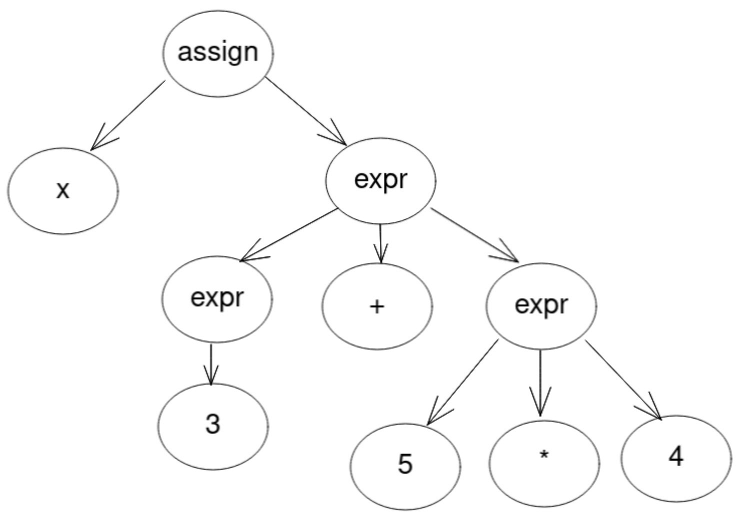
Generally implementations of syntax tree makers use class for each type of node, that is expr will have a class, statement will have a class etc, but here it is implemented as a single node type with a variable type which stores the type of node. And the execute function executes conditionally based on what the type variable is.
In the Parser.y file one can see that line the top level which calls execute of compile based on the command line argument passed. As mentioned before, compile and execute used for compiling and interpreting statements.

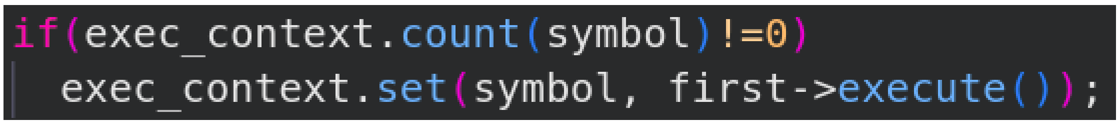
Interpreter

The interpreter uses the parse tree made in the previous step and executes each of the nodes as per the the type of node.
For variables, it uses a custom class called exec_context, which handles scoping of variables when entering blocks, this is what helps in the interpreting of functions, as functions require a new scope for each call. All function definition nodes are stored in a key value map, so on function call those nodes are executed after adding the arguments to new scope with names defined in the function definition.

Eval block:
The interpreter just recursively executes the expression as part of a tree. For example:
Here the interpreter calls execute on assign and for statement for type assign, the second child is executed first, and the result is stored in the exec context for the first parameter.

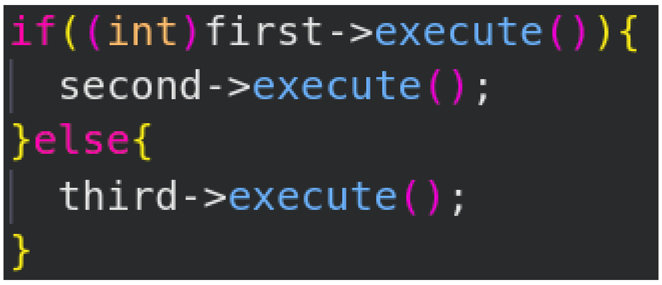
Syntax tree for x = 3+5*4





```
if(exec_context.count(symbol)!=0)
  exec_context.set(symbol, first->execute());
```

Jump block:
The interpreter first executed the first child, then based on the result execute the second or third.

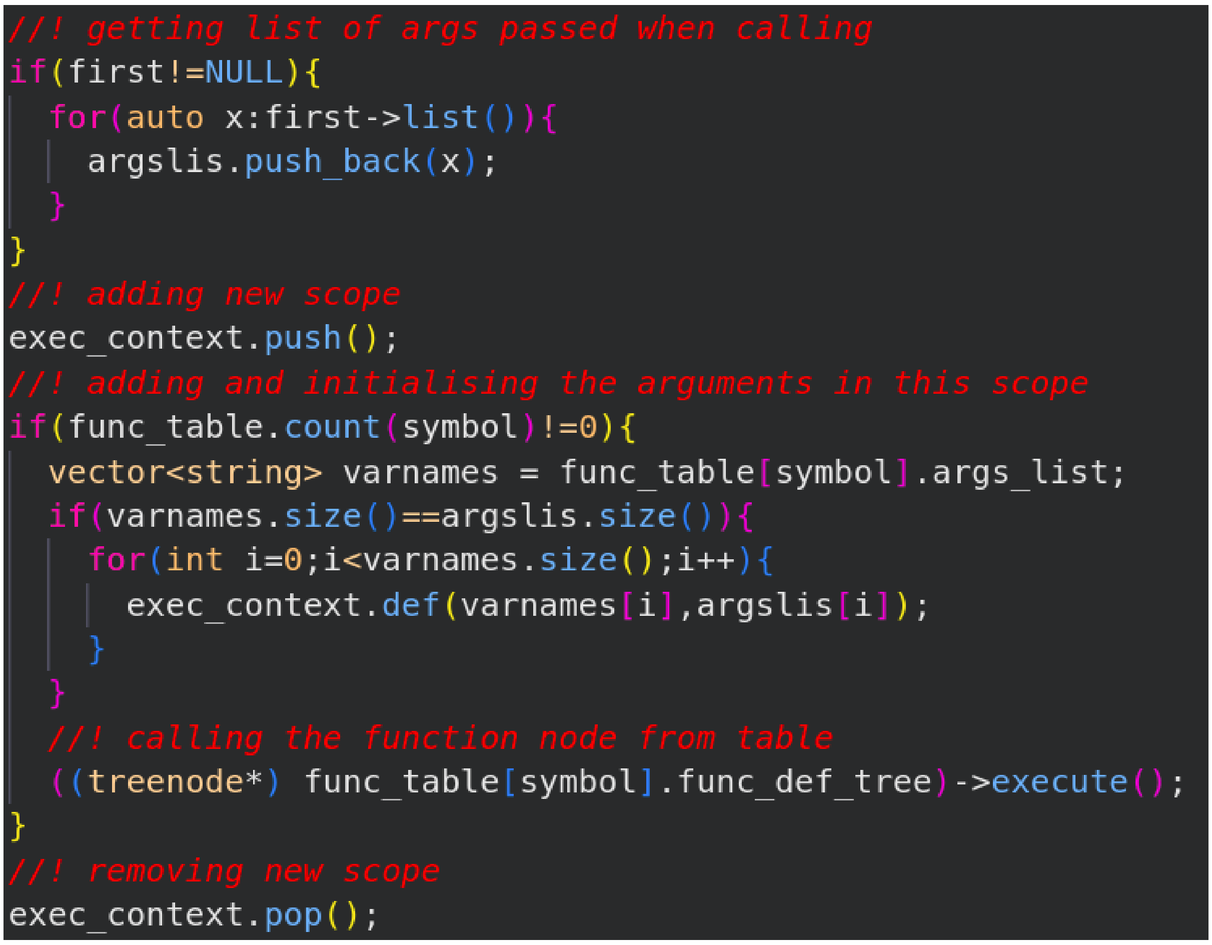


```
if((int)first->execute()){
    second->execute();
}else{
    third->execute();
}
```

$$\begin{tikzcd}
 & \text{If\_else} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{expr} & \text{stmt} & \text{stmt}
\end{tikzcd}$$

Function Block:
Let's talk about calling as the definition part is just treated as a normal node whole pointer is stored in a map.
The calling of a function involves the following steps,

- Adding a new scope
- Declaring the function's arguments on this new scope and initialising them.
- Run the function node.



```
//! getting list of args passed when calling
if(first!=NULL){
    for(auto x:first->list()){
        argslis.push_back(x);
    }
}
//! adding new scope
exec_context.push();
//! adding and initialising the arguments in this scope
if(func_table.count(symbol)!=0){
    vector<string> varnames = func_table[symbol].args_list;
    if(varnames.size()==argslis.size()){
        for(int i=0;i<varnames.size();i++){
            exec_context.def(varnames[i],argslis[i]);
        }
    }
    //! calling the function node from table
    ((treenode*) func_table[symbol].func_def_tree)->execute();
}
//! removing new scope
exec_context.pop();
```

Compiler

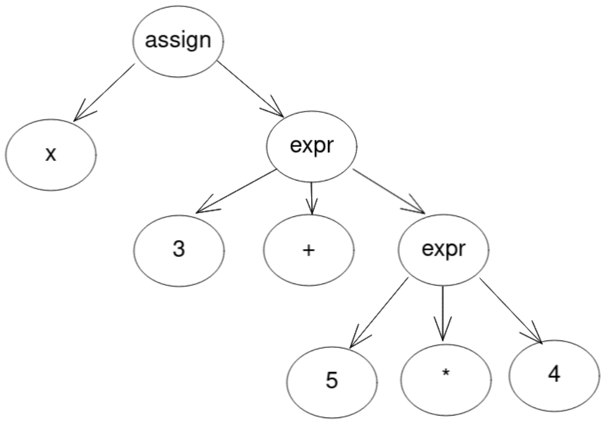
The compiler also uses the syntax tree formed using treenode in the previous step. The compiler has to convert the syntax tree into xHUH assembly. The details about the ISA have been mentioned in the previous sections.

Eval block

The compilation of normal eval blocks use push, pop, and arithmetic operations.
The following is how it converts the given tree into code
(compile called on assign)
(assign Node is programmed to call compile on second child, whose solution would be in stack, and then pop top of stack into x)
(expr compile compiles first third then the operation)
(for ids, they just get pushed onto stack)

```
Push 3
Push 5
Push 4
Mul
Add
Pop x
```

Syntax tree for x = 3+5*4



Jump block

For jump blocks random labels will have to be generated and placed and jmp statements called to handle if and for loops. The following template shall be used to implement

If-else if (expr) { statement1 } else { statement2 }

```
expr.compile()
Push 0
Jeq label1
stmt1.compile()
Jmp label2
label1:
stmt2.compile()
label2:
```

For for (statement1; statement2; statement3) { statement4 }

```
stmt1.compile()
Label1:
stmt2.compile()
Push 0
Jeq label2:
stmt4.compile()
stmt3.compile()
Jmp label1
label2:
```

Function block

Since function definitions shouldn’t be executed (in order) but compiled, we wrap the function definition around an unconditional jump to after the end of the function.

Function definition:

```
Jmp label1
Func_name:
Push (number of params)
(for each variable declared inside function, push a 0)
Push 0
funcbody.compile()
Label1:
```

Function call:

```
( call compile for each arg of the function )
arg.compile()
call func_name
```

The cool thing about compiling code inside functions is that all variables inside the function will be referred to as an offset of the frame pointer.

So first argument of the function will be retrieved from the stack the following way:

```
Push xFP
Push -1
Add
Push ST
```

Example Input and output

The following code was used to run.

Code

link

Before that we first make our compiler by using make.

> make clean

> make

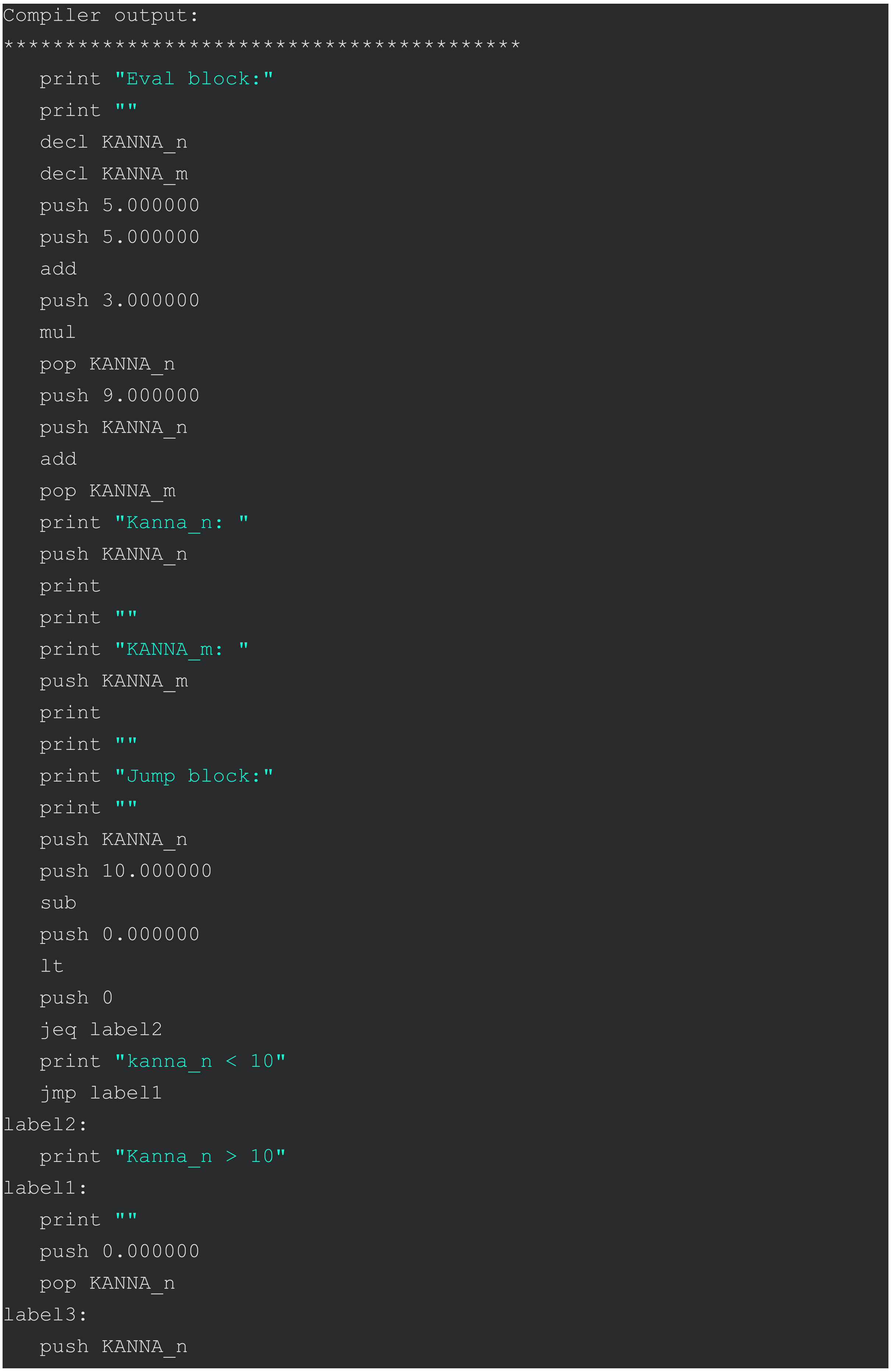
Then run the following script to auto run everything.

```
printf "******************************************\nInput:\n******************************************\n"
cat $1
printf "******************************************\nCompiler output:\n******************************************\n"
./main $1
./main $1 > out.md
printf "******************************************\nInterpreter output:\n******************************************\n"
./main $1 false
printf "******************************************\nSimulator output:\n******************************************\n"
node simulator.js
```

Result:

```
******************************************
Input:
******************************************
VECHUKO("Eval block:"):
VECHUKO(""):

KANNA_n:
KANNA_m:
(5+5)*3 SOLRAN KANNA_n SEIRAN:
9+KANNA_n SOLRAN KANNA_m SEIRAN:

VECHUKO("Kanna_n: "):
VECHUKO(KANNA_n):
VECHUKO(""):
VECHUKO("KANNA_m: "):
VECHUKO(KANNA_m):
VECHUKO(""):
```

```
VECHUKO("Jump block:"):
VECHUKO(""):
ORUVELA ((KANNA_n - 10) < 0) {
   VECHUKO("kanna_n < 10"):
}
ILLENA {
   VECHUKO("Kanna_n > 10"):
}

VECHUKO(""):
NA (0 SOLRAN KANNA_n SEIRAN:) VAATI SONNA (KANNA_n < 5:) VAATI SONNA
MAARI (KANNA_n + 1 SOLRAN KANNA_n SEIRAN:) {
 VECHUKO(KANNA_n):
 VECHUKO(" "):
}

VECHUKO("Function block:"):
VECHUKO(""):

KANNA_fib(KANNA_n){
 ORUVELA (KANNA_n == 0){
   ITHU EPDI IRUKU 0:
 }
 ORUVELA (KANNA_n == 1){
   ITHU EPDI IRUKU 1:
 }
 ITHU EPDI IRUKU KANNA_fib(KANNA_n-1) + KANNA_fib(KANNA_n-2):
}

VECHUKO("Printing Fibonacci sequence from 0 to 20..."):
VECHUKO(""):

KANNA_n:

NA (0 SOLRAN KANNA_n SEIRAN:) VAATI SONNA (KANNA_n < 20:) VAATI SONNA
MAARI (KANNA_n + 1 SOLRAN KANNA_n SEIRAN:) {
 VECHUKO(KANNA_fib(KANNA_n)):
 VECHUKO(" "):
}

VECHUKO(""):
*****************************************
```



```
Compiler output:
*****************************************
   print "Eval block:"
   print ""
   decl KANNA_n
   decl KANNA_m
   push 5.000000
   push 5.000000
   add
   push 3.000000
   mul
   pop KANNA_n
   push 9.000000
   push KANNA_n
   add
   pop KANNA_m
   print "Kanna_n: "
   push KANNA_n
   print
   print ""
   print "KANNA_m: "
   push KANNA_m
   print
   print ""
   print "Jump block:"
   print ""
   push KANNA_n
   push 10.000000
   sub
   push 0.000000
   lt
   push 0
   jeq label2
   print "kanna_n < 10"
   jmp label1
label2:
   print "Kanna_n > 10"
label1:
   print ""
   push 0.000000
   pop KANNA_n
label3:
   push KANNA_n
```

```
    push 5.000000
    lt
    push 0
    jeq label4
    push KANNA_n
    print
    print " "
    push KANNA_n
    push 1.000000
    add
    pop KANNA_n
    jmp label3
label4:
    print "Function block:"
    print ""
    jmp label5
KANNA_fib:
    push 1
    push xFP
    push -1
    add
    push ST
    push 0.000000
    eq
    push 0
    jeq label6
    push 0.000000
    ret
label6:
    push xFP
    push -1
    add
    push ST
    push 1.000000
    eq
    push 0
    jeq label7
    push 1.000000
    ret
label7:
    push xFP
    push -1
    add
```

```
   push ST
   push 1.000000
   sub
   call KANNA_fib
   push xFP
   push -1
   add
   push ST
   push 2.000000
   sub
   call KANNA_fib
   add
   ret
label5:
   print "Printing Fibonacci sequence from 0 to 20..."
   print ""
   decl KANNA_n
   push 0.000000
   pop KANNA_n
label8:
   push KANNA_n
   push 20.000000
   lt
   push 0
   jeq label9
   push KANNA_n
   call KANNA_fib
   print
   print " "
   push KANNA_n
   push 1.000000
   add
   pop KANNA_n
   jmp label8
label9:
   print ""
******************************************
Interpreter output:
******************************************
Eval block:
Kanna_n: 30
KANNA_m: 39
Jump block:
```

```
Kanna_n > 10
0 1 2 3 4 Function block:
Printing Fibonacci sequence from 0 to 20...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
******************************************
Simulator output:
******************************************
Eval block:
Kanna_n: 30
KANNA_m: 39
Jump block:
Kanna_n > 10
0 1 2 3 4 Function block:
Printing Fibonacci sequence from 0 to 20...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

As we can see both the compiler and interpreter give the same results and verify each other. The program input was made to use almost every feature provided by Rajinican, so all the features mentioned here works.

Improvements and scope

More optimisations could be done with syntax trees and assembly code.
The assembly is not usable since there is no machine that uses xHUH architecture, so xHUH could be converted to x86
Types could be implemented in the language.
Arrays could be implemented.
More error handling could have been included.