PFL-TP2

Group T04_G14

- Francisco Miguel Correia Mariano Pinheiro Cardoso - up202108793 - 50%
- José António Pereira Martins - up202108794 - 50%

Implementation strategy

To complete this project we used the template file given by us via Moodle and started building our program little by little following the guide steps one by one to be able to achieve all goals set to us.

Part 1 - Implementation

The main objective of this section was to develop a low-level machine capable of executing a given configuration (c, e, s), where c represents the configuration, e represents the evaluation stack, and s represents the storage. We were provided with a template file that included a data type for instructions and four essential functions that needed implementation: createEmptyStack, stack2String, createEmptyState, state2String, and run.

Data Types Definition

Value

To represent the possible types stored inside the stack and storage, we introduced the Value data type, which could be either an integer (IntValue) or a boolean (TT for true, FF for false).

Stack and State

Understanding the roles of the evaluation stack and storage, we defined two types - Stack (a list of Values) and State (a list of variable-value pairs represented by strings and Values).

Initial Functions

Once all data types that were going to be used for this part were defined we started working on the functions to implement. Firstly we implemented the following 4 initial functions:

- `createEmptyStack`: This function generates an empty stack, to initialize the evaluation stack. To do this it simply returns an empty list.
- `createEmptyState`: Similar to createEmptyStack, this function creates an empty state to initialize the machine state.
- `stack2Str`: This function converts the stack into a string, it recursively traverses the stack, transforming each Value into a string using the value2Str auxiliary function. The resulting strings are then concatenated with commas.
- `state2Str`: This function orders the state alphabetically and then converts it into a string, It does this by using the `stateSort` and the `state2StrAux auxiliary function.

Run

After implementing these initial functions we started implementing the `run` function. This function executes instructions in the low-level machine making use of the stack and changing the values in state. To do this, we recursively go through the instructions, executing each instruction with the help of the `execute` auxiliary function. At the end, we return the final (c, e, s).

Even though the `run` executes all the instructions the real brain of the machine is in the `execute` function. This function executes an instruction in the low-level machine. It receives the Instruction, the stack, and the state, manipulates the stack and the state and then it returns a tuple with the updated (stack, state). There were a lot of instructions defined, and, to not get the code too confusing, we didn't use a case of pattern matching, but instead, we implemented a different `execute` function for each instruction. The instructions implemented were the following:

- `Push`: For this instruction, we just add the IntValue to the top of the stack.
- `Tru`: For this instruction, we just add the `TT` to the top of the stack.
- `False`: For this instruction, we just add the `FF` to the top of the stack.
- `And`: For this instruction, we first check if the stack has at least 2 elements and if the top 2 elements are both Boolean Values. To do so, we use the auxiliary function `isBool`. Then, we check if both the top 2 elements are `TT`. If so, we add `TT` to the rest of the stack. If not, we add `FF` to the rest of the stack.
- `Neg`: For this instruction, we first check if the stack has at least 1 element and if the top element is a Boolean Value. To do so, we use the auxiliary function `isBool`. Then, we check if the top element is `TT`. If so, we add `FF` to the rest of the stack. If not, we add `TT` to the rest of the stack.
- `Add`: For this instruction, it was said that we need to get the top two integer values from the stack. To do so, we use the auxiliary function `findFirst2Int`. Then, we remove the two top integer values from the stack using a function called `removeFirst`, and add the sum of those two values to the rest of the stack.
- `Mult`: Just like the `Add` instruction, we need to get the top two integer values from the stack. To do so, we use the auxiliary function `findFirst2Int`. Then, we remove the two top integer values from the stack using a function called `removeFirst`, and add the multiplication of those two values to the rest of the stack.
- `Sub`: For this instruction, we first check if the stack has at least 2 elements and if the top 2 elements are both IntValues. To do so, we use the auxiliary function `isInt`. If so, we get the top two values from the stack and add the subtraction of those two values to the rest of the stack.
- `Equ`: For this instruction, we first check if the stack has at least 2 elements and if the top 2 elements have the same type using the `isInt` and `isBool` auxiliary functions. If so, we get the top two values from the stack and add the result of the equality of those two values to the rest of the stack. If the top two elements are the same value, we add `TT` to the rest of the stack. If not, we add `FF` to the rest of the stack.
- `Le`: For this instruction, it was necessary to be sure that the top two elements of the stack are IntValues. So, we first check if the stack has at least 2 elements and if the top 2 elements are both IntValues. To do so, we use the auxiliary function `isInt`. If so, we get the top two values from the stack and add the result of the comparison of those two values to the rest of the stack. If the first element is lower or equal to the second element, we add `TT` to the rest of the stack. If not, we add `FF` to the rest of the stack.
- `Fetch`: In this instruction, we just need to get the `Value` of a variable from the state. To do so, we use the auxiliary function `findValueFromKey`. Then, we put the value on the top of the stack.

- `Store`: In this instruction we need to get the top element of the stack and update the state so that the value is bound to the variable. To do so, we use the auxiliary function `remove key`, and then we add the new pair to the state and return the rest of the stack and the new state.
- `Branch`: This instruction is a conditional. If the top element of the stack is `TT`, we execute the first instruction of the list of instructions. If it is `FF`, we execute the second instruction of the list of instructions. If it is neither, we return an error.
- `Noop`: This instruction does nothing. So, we just return the stack and the state.
- `Loop`: Finally, this instruction is a loop, like a while-loop. It executes `code1` and then calls a `Branch` instruction, where the first instruction is `code2` added to a `Loop` instruction with the same `code1` and `code2`. The second instruction is a `Noop` instruction so that the program continues if the top element of the stack is `FF`.

Auxiliary Functions

Several auxiliary functions were implemented to the functionality of this parte:

- `isBool`: Determines whether a Value is a boolean.
- `findFirst2Int`: Finds the first two IntValue elements from the stack.
- `removeFirst`: Removes the first occurrence of a specified value from a list.
- `isInt`: Checks if a Value is an IntValue.
- `findValueFromKey`: Locates the Value of a variable in the state.
- `removeKey`: Removes a specified variable-value pair from the state.
- `value2Str`: Converts a Value to a string
- `stateSort`: Orders the state alphabetically
- `pair2Str`: Converts a pair to a string

Concluding, we can run the machine with the `run` function, which receives a state and returns a different state that is the result of the execution of the instructions with the function `execute`. The `execute` function receives an instruction, a stack, and a state, and returns a tuple with the updated stack and state. The `execute` function is implemented with a different procedure for each instruction, and each of those functions returns the updated stack and state.

Part 1 - Testing

To test the implementation, we used the `testAssembler` function given, and it worked as expected. When testing with the `testAssembler` function, we noticed that we were not removing the old values on the stack after using them. To do so, we had to change all the functions so that they return the rest of the stack after using the values.

Tests:

```
testAssembler [Push 10,Push 4,Push 3,Sub,Mult] == ("-10","")
testAssembler [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"] ==
("","a=3,someVar=False,var=True")
testAssembler [Fals,Store "var",Fetch "var"] == ("False","var=False")
testAssembler [Push (-20),Tru,Fals] == ("False,True,-20","")
testAssembler [Push (-20),Tru,Tru,Neg] == ("False,True,-20","")
testAssembler [Push (-20),Tru,Tru,Neg,Equ] == ("False,-20","")
testAssembler [Push (-20),Push (-21), Le] == ("True","")
```

```
testAssembler [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"] ==
("","x=4")
testAssembler [Push 10,Store "i",Push 1,Store "fact",Loop [Push 1,Fetch
"i",Equ,Neg] [Fetch "i",Fetch "fact",Mult,Store "fact",Push 1,Fetch
"i",Sub,Store "i"]] == ("","fact=3628800,i=1")
testAssembler [Push 1,Push 2,And] == "Run-time error"
testAssembler [Tru,Tru,Store "y", Fetch "x",Tru] =="Run-time error"
```

Part 1 - Conclusion

In this part, we were able to understand properly how the low-level machine works, and how to implement it. We were also able to understand how to use the auxiliary functions to make the code more readable and easier to implement. It was an important part of the work because it was the base for the next part. It gave us a better understanding of `where` cases and `guard` usage.

Part 2 - Implementation

Now that we had the low-level machine-implemented, we needed to do a compiler that would translate a program written in a high-level language into a low-level machine program. To achieve that we implement a lexer, a parser, and a compiler. We also define some data types to represent the different types of expressions and instructions:

Lexer

First thing first, we started by implementing a lexer. This function was used to extract tokens from a string and store them in an array. Tokens are sequences of chars that can be treated as units, this goes from operators to keywords, brackets, semicolons, numbers, variable names, etc... By doing this we were not only able to remove all the spaces and useless characters but it also made the process of parsing much more easier.

Lexer implementation

To implement the lexer we created a function that processed each character in the input string based on specific conditions:

- Standard characters like ";", "(", ")", "+", "-", or "*" were added to the list of tokens.
- For "=", the lexer checked the next character to distinguish between the assignment operator "=" and the equality operator "==".
- Special tokens like ":=" and "<=" were identified based on the next two characters.
- The "¬" character was treated as "not" and added to the list of tokens.
- Numbers were extracted using the auxiliary function getInt, while letters were processed using getWord.
- Spaces were ignored, and any other characters resulted in an error.

Compiler

After having implemented the lexer, we started working on the compiler itself. We decided to leave the parser for the end as it would be easier to implement it after having defined what the compiler would receive and how would it work. This function is the main piece of this second part it receives an array of high-level

instructions and converts them to low-level instructions that our previously implemented machine can execute.

Data types

First thing first we started by introducing three data types: Aexp (representing arithmetic expressions), Bexp (representing Boolean expressions), and Stm (representing statements). These data types facilitated a clear and structured representation of the program's components. Aexp included options such as Num (for numbers), Var (for variables), AddE, SubE, and MultE for addition, subtraction, and multiplication expressions. Bexp covered boolean values, equality checks (EqE and EqBexpE), lower or equal checks (LeE), negation (Neg), and conjunction (And). Stm dealt with both arithmetic and boolean expressions, variable assignments (Assign), if statements (If), and while statements (While).

Compiler implementation

Having finished defining all types we started implementing the compiler, to do that we have defined 3 functions called `compile`, `compA`, and `compB` which will convert the statements and the expressions previously defined into code for our machine.

CompA

This function will be in charge of converting all the arithmetic expressions into code, to do that it was implemented with a different procedure for each expression it receives:

- `Number x` - Adds `Push x` to the list of instructions
- `Var x` - Adds `Fetch x` to the list of instructions
- `AddE x1 x2` - Adds the result of `compA x1` then the result of - `compA x2` and in the end `Add` to the list of instructions
- `SubE x1 x2` - Adds the result of `compA x1` then the result of - `compA x2` and in the end `Sub` to the list of instructions
- `MultE x1 x2` - Adds the result of `compA x1` then the result of
- `compA x2` and in the end `Mult` to the list of instructions

CompB

Similar to compA this function will be in charge of converting all the boolean expressions into code, to do that it was implemented with a different procedure for each expression it receives:

- `Boolean x` - if x is True adds `Tru` to the list of instructions, otherwise, it adds `Fals`
- `EqE x1 x2` - Adds the result of `compA x1` then the result of - `compA x2` and in the end `Eq` to the list of instructions
- `EqBexpE x1 x2` - Similar to EqE but for boolean equality Adds the result of `compB x1` then the result of `compB x2` and in the end `Eq` to the list of instructions
- `LeE x1 x2` - Adds the result of `compB x1` then the result of `compB x2` and in the end `Leq` to the list of instructions
- `Neg x` -Adds the result of `compB x` then `Leq` to the list of instructions
- `And x1 x2` - Adds the result of `compB x1` then the result of - `compB x2` and in the end `And` to the list of instructions

Compile

This function will be in charge of converting all the statements into code, like CompA and CompB it was implemented with a different procedure for each expression it receives:

- `Aex`: For this statement, we just call the `compA` function with the arithmetic expression and add the result to the rest of the code.
- `Bex`: For this statement, we just call the `compB` function with the boolean expression and add the result to the rest of the code.
- `Assign`: For this statement, we first call the `compA` function with the arithmetic expression, add it to a `Store` instruction with the variable name, and add it to the rest of the code.
- `If`: For this statement, we first call the `compB` function with the boolean expression, add it to a `Branch` instruction with the code of the first statement and the code of the second statement, and add it to the rest of the code.
- `While`: For this statement, we just add a `Loop` instruction that receives the `compB` of the boolean expression and the `compile` of the list of statements, and add it to the rest of the code.

Parser

With everything else out of the way, it was time to implement the parser. The parser is what connects the lexer and the compiler, it transforms the list of tokens given by the lexer and converts them into the high-level instruction the compiler will process. This was the most challenging part of the project as it had not only to convert the tokens into instruction but also because it had to deal with all the precedences of the operator. Due to the difficulty of this task, we decided to use an auxiliary binary tree to maintain the instructions while the rest of the tokens were being processed, this way we were able to easily manipulate the expressions and statements in case of need. This way after the list of tokens has finished processing we only need to apply a dfs to convert the binary tree into a list of statements to the compiler.

Binary tree

The binary tree consisted of a simple data type where the values could be either a string and two Tree representing the Node and the left and right branches or a Leaf representing the null.

```
data Tree = Node String Tree Tree | Leaf  deriving Show
```

Parser Implementation

The implementation of the parser can be divided into 3 parts, Parsing of arithmetic and boolean expressions, Parsing of statements, and Parsing of the Binary Tree.

Parsing of arithmetic and boolean expressions

We started the parser by implementing a function that could parse both the arithmetics and Boolean expressions and transform them into a binary tree. First thing first, we started by creating a function `precedence` to calculate the level of precedence of an operator

```

```

```
precedence:: String -> Integer
precedence [] = 0
precedence "not" = 2
precedence "*" = 3
precedence "+" = 4
precedence "-" = 4
precedence "<=" = 5
precedence "==" = 6
precedence "=" = 7
precedence "and" = 8
precedence _ = 1
```

We also implemented a function `inside` to obtain the content inside two brackets so we could also deal with those in our expression. Once that was done we implemented a function called `ParseAex` that initializes 3 empty strings (one for the token and the other for the contents to be expression to be left and right of the operator) and starts reading the token of the expression one by one. At the first iteration the token at the beginning of the list is set as the current Node, then every time the function reads a token it compares its precedence with the current node, if the precedence of the new token is lower or equal to the precedence of the Node it simply adds the new token to the right of the current token, however, if the precedence is higher the new Token becomes the Node and both the old token and the right list are appended to the left. Once the list of tokens is empty both the left and right lists are recursively processed, and a Tree is created with the current node and the resulting left and right Trees.

After this, the only thing left in this part was to deal with brackets. For that every time a token"(" was read the contents inside it were added to the left or right list without being processed, only processing its content once the list of tokens contained only the content inside the brackets. This way we ensured that everything inside the brackets were always at the end of the tree, thus being the first to be processed.

Parsing of statements

For this part, we simply created a function `parseStatements` that iterates through the list of tokens looking for the keyword of statements, depending on the token it calls and the auxiliary function to parse that statement. For the "If" token it calls the `parseIf` function that sets the node to "if" parses the condition expression and sets it to its left branch, and then for the right branch it calls the function `parseIfElse` which will create another Tree with the Node set as "IfElse" and with the left and right branches containing the results of the parsing code of the if and else statement respectively.

For the "While" token it calls the `parseWhile` function which similar to the `parseIf` function parses the condition and sets it to the left branch, while the code to be executed is parsed and set to the right branch

For the ":=" token it calls the `parseAssign` function that sets the variable to the left branch and the parsed arithmetic expression on the right branch

Anything else it considers an arithmetic expression.

To group instructions in the binary Tree the function `parseStatement` will always create a Tree with the node set to "Seq" and with one statement in the left branch and either another "Seq" tree when there are more instructions to execute or a Leaf otherwise.

Parsing of The Binary Tree

For this last part, we simply had to create the function `parseTree` that uses a personalized dfs algorithm to traverse the binary tree and transform the nodes into instructions for the compiler. Depending on the tokens in the node the function will add a different statement to the program.

With this, we were able to transform a string into code to be executed by the low-level machine.

Part 2 - Testing

To test the compiler function, we ran the following tests:

```
compile [(Assign "y" (Num 1)), (While (NegE (EqE (Var "x") (Num 1)))
[(Assign "y" (MultE (Var "y") (Var "x"))), (Assign "x" (SubE (Var "x") (Num
1)))])] == [Push 1,Store "y",Loop [Push 1,Fetch "x",Equ,Neg] [Fetch
"x",Fetch "y",Mult,Store "y",Push 1,Fetch "x",Sub,Store "x"]]
compile [Aex (Num 1), Aex (Var "x")] == [Push 1,Fetch "x"]
```

To test the lexer function, we used the arguments of the tests given, and it worked as expected.

To test the whole program we ran the following tests:

```
testParser "x := 6; y := 6; if y<=5 then x:=5; else (x:=10; y:=20;);" ==
("","x=10,y=20")

testParser "x:=10;" == ("","x=10")

testParser "if 2 == 3 then x:=20; else x:=10;" == ("","x=10")

testParser "par := 0; x:=10; while (not (x==0)) do (if (par == 0) then
par:=1; else par := 0; x := x - 1;);" == ("","par=0,x=0")

testParser "par := 0; x:=9; while (not (x==0)) do (if (par == 0) then
par:=1; else par := 0; x := x - 1;);" == ("","par=1,x=0")

testParser "x:=5; acc := 1; while (not(x==1)) do (acc := acc * x; x := x-
1;);" == ("","acc=120,x=1")

testParser "if (True and True and True and True) then x:=1; else x:=0;"
("","x=1")

testParser "if (not True and True and True and True) then x:=1; else x:=0;"
("","x=0")

testParser "x:=10; if (True and 1+2 == 2+1) then x:=x+1; else x:=x-1;" ==
("","x=11")

testParser "x := 5; x := x - 1;" == ("","x=4")

testParser "x := 0 - 2;" == ("","x=-2")
```

```
testParser "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;" ==
("","y=2")

testParser "x := 42; if x <= 43 then x := 1; else (x := 33; x := x+1;);" ==
("","x=1")

testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1;" ==
("","x=2")

testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1; z :=
x+x;" == ("","x=2,z=4")

testParser "x := 44; if x <= 43 then x := 1; else (x := 33; x := x+1;); y
:= x*2;" == ("","x=34,y=68")

testParser "x := 42; if x <= 43 then (x := 33; x := x+1;) else x := 1;" ==
("","x=34")

testParser "if (1 == 0+1 = 2+1 == 3) then x := 1; else x := 2;" ==
("","x=1")

testParser "if (1 == 0+1 = (2+1 == 4)) then x := 1; else x := 2;" ==
("","x=2")

testParser "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);" ==
("","x=2,y=-10,z=6")

testParser "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i
:= i - 1;);" == ("","fact=3628800,i=1")
```

Part 2 - Conclusion

In this part, we were able to start understanding how to implement a compiler. It was interesting to see how the lexer, the parser, and the compiler worked since we are just used to writing the code without being aware of what happens behind the scenes. It was a difficult part of the work, mainly because there were a lot of different cases and rules to consider, and it was hard to understand how to implement them.

Conclusion

In conclusion, we were able to implement a low-level machine and a compiler. It was a very interesting work because we were able to understand how the low-level machine works, and how to implement it. We were also able to understand how to implement a compiler, and how to use the lexer and the parser. It was difficult work, mainly because there were a lot of different cases and rules to consider, and it was hard to understand how to implement them. We were able to implement all the functions and the tests, and we were able to understand how to use the auxiliary functions to make the code more readable and easier to implement. This practical work was very important to develop our skills in Haskell and to understand how to implement a low-level machine and a compiler. We consider that we were able to achieve the goals of the work, and we are satisfied with the final result.