Faculdade de Engenharia da Universidade do Porto

Programação Funcional e em Lógica

Compilador em Haskell

2.º Projeto

3.º Ano - Licenciatura em Engenharia Informática e Computação
2023/2024

Turma 01 - Grupo 07

Estudantes & Autores

Isabel Maria Lima Moutinho - up202108767 - 50%
Tiago Ribeiro de Sá Cruz - up202108810 - 50%

Part 1 - Assembler

Data Structure

In order to represent our assembler data structure, we firstly used the definition given by the professors of Instruction and Code.

```
      data Inst =
        Push Integer | Add | Mult | Sub | Tru | Fals | Equ | Le | And | Neg |
Fetch String | Store String | Noop |
        Branch Code Code | Loop Code Code
        deriving Show
      type Code = [Inst]
```

Secondly, in order to represent our Stack, since we had to accept both Integers and Booleans in it, we created a new data called *StackValue*, therefore, our Stack consists of a list of StackValues.

```
      data StackValue = Value Integer | TT | FF deriving (Show, Eq)
      type Stack = [StackValue]
```

For the representation of the machine's state, we decided to use a HashMap in order to better and easily map the value's to its key, where the key is a string and the value is a StackValue.

```
      type Key = String
      type Value = StackValue

      type State = HashMap.Map Key Value
```

Interpreter

To run the given code, we developed the function run, as given in the instructions. This function, upon receiving the code, iterates through every instruction and acts accordingly, calling a handler function for each of these instructions until the list of instructions is empty.

In order for the run function to work, we also developed functions that start an empty stack and state, so that they represent the initial state of the program, and that transform them in string, so a user can see the final result.

Since there are many instructions, we used a *case of* to handle them all and each case calls a function we developed for it, always throwing the error "Run-time error" if said operation isn't valid.

To handle *Push n*, *Tru* and *Fals*, which pushes the integer value n, True or False to the stack, we developed a function called *push*, that receiver Either Integer Bool and that adds the given value to the stack.

To handle *Add*, which adds the two topmost integers in the stack, we developed *add*, that receives the stack, adds the two topmost integers and returns the new stack with the new value on top and without the two prior values that were used in the operation.

To handle *Mult,* which multiplies the two topmost integers in the stack, we developed *mult*, which receives the stack, multiplies the two topmost integers and returns the new stack with the new value on top and without the two prior values that were used in the operation.

To handle *Sub*, which subtracts the two topmost integers in the stack, we developed *sub,* which receives the stack, subtracts the two topmost integers and returns the new stack with the new value on top and without the two prior values that were used in the operation.

To handle *Equ*, which verifies if two given integers or booleans values are equal (i.e. 1=1 or True=True), we developed *eq*, that receives the stack, compares the two topmost StackValues, throwing an error if they're not of the same type and replacing those values with the result of comparison.

To handle *Le*, which verifies if a given integer is less than or equal to another, we developed *le*, that receives the stack and replaces the two topmost values with the result of the comparison.

To handle *And*, which verifies if two boolean values are both true, we developed *ande*, which receives the stack and checks if the two topmost values are true, replacing them with the result.

To handle *Neg*, which negates a boolean value, we developed *neg*, which receives a stack and replaces the topmost boolean with its negation.

To handle *Fetch x*, which pushes to the stack the StackValue bound to the Key received (x) in the state, we developed *fetch*, which receives the key, the stack and the state and returns the new stack with the StackValue of the key x at the top.

To handle *Store x*, which pops the topmost StackValue of the stack and binds it to the Key received (x) in the state, we developed *store*, which receives a key, a stack and a state and returns a new stack and a new state, where the stack no longer has its topmost StackValue and the state has a new StackValue mapped to its key x.

To handle *Noop*, which does nothing, we developed *noop*, which returns the given stack and state unaltered.

To handle *Branch c1 c2*, which pops the topmost StackValue of the stack and returns the first Code received (c1) if that value was true, the second Code received (c2) if it was false and fails otherwise, we developed *branch*, which receives two lists of instructions (Code) and a stack and returns Code c1 if the topmost value is true and c2 otherwise and the new stack without the boolean at the top.

To handle *Loop c1 c2*, which is used to loop through the two received Codes, c1 and c2, in that order, we developed the function *loop, which* returns a Code made of c1 followed by a Branch between c2 followed by the loop itself, and a Noop.

Part 2 - Parser

Data Structure

In order to represent our parser's data structure, we first defined the three datas given in the example, Aexp, Bexp and Stm, given for arithmetic expressions, boolean expressions and statements, respectively.

```
data Aexp =
  Num Integer | Var String | AddAexp Aexp Aexp | MultAexp Aexp Aexp | SubAexp
Aexp Aexp
  deriving Show

data Bexp =
  TruBexp | FalsBexp | NegBexp Bexp | EquBexp Aexp Aexp | EqBexp Bexp Bexp |
LeBexp Aexp Aexp | AndBexp Bexp Bexp
  deriving Show

data Stm =
  AssignStm String Aexp | IfStm Bexp [Stm] [Stm] | WhileStm Bexp [Stm]
  deriving Show
```

Here, we added what we would use as arithmetic expressions to **Aexp -** Num, that will be the *Push n*, Var that will also be *Push n* but for strings, AddAexp, that will work for the add operation, MultAexp for the multiplication operation and SubAexp for the subtraction operation.

For **Bexp**, we added the boolean expressions such as TruBexp and FalsBexp that function as *Push TT* and *Push FF*, NegBexp for the negation, EquBexp to evaluate the equality of arithmetic expressions, EqBexp to evaluate the equality boolean expressions, LeBexp to evaluate if a given arithmetic expression is lesser than or equal to another and AndBexp, for the *and* operation.

Inside the data **Stm**, we added the Assign statement that receives a string and an arithmetic expression (e.g. x:=5), the if statement that receives the boolean expression that it will evaluate and two lists of statements, the first one corresponding to the "then" and the last one to the "else". The while statement that receives a boolean expression and a list of statements that it will do while said boolean expression is true.

To represent our program, which is a list of statements, we defined the type program.

```
type Program = [Stm]
```

Compiler

Firstly, we developed two auxiliary functions compA and compB that compile arithmetic and boolean expressions respectively.

CompA receives an arithmetic expression and, depending on what the command is, returns a Code.

- Num n → [Push n]
- Var x → [Fetch x]
- AddAexp a1 a2 → compA a2 ++ compA a1 ++ [Add], which is done in this order since the stack is LIFO (last in first out), so we want to add a1 with a2. We also want to compile a2 and a1 again, because they may be arithmetic expressions.
- MultAexp a1 a2 → compA a2 ++ compA a1 ++ [Mult], with the same rules as AddAexp.
- SubAexp a1 a2 → compA a2 ++ compA a1 ++ [Sub], with the same logic behind MultAexp and AddAexp.

CompB receives a boolean expression and, depending on what the command is, returns a Code.

- TruBexp → [Tru]
- FalsBexp → [Fals]
- NegBexp b → compB b ++ [Neg], where we also compile *b* because it may be a boolean expression itself.
- EqBexp a1 a2 → compB a1 ++ compB a2 ++ [Equ]
- EquBexp a1 a2 → compA a1 ++ compA a2 ++ [Equ]
- LeBexp a1 a2 → compA a2 ++ compA a1 ++ [Le], where a1 and a2 are reversed because the order matters and we want to check if a1 is lesser than or equal to a2 and not the other way around.
- AndBexp b1 b2 → compB b1 ++ compB b2 ++ [And]

The compile function itself, upon receiving the program, goes through the statements and compiles them accordingly.

For the assign statement (AssignStm), since it receives a string and an arithmetic expression like explained before, it needs to compile the arithmetic expression, calling compA and to store the string, calling [Store x]. In the end, it compiles the rest recursively.

For the “if” statement (IfStm), since it receives a boolean expression and two lists of statements, it calls compB to compile the boolean expression, and the Branch function with the compile of the lists of statements in order.

For the “while” statement (WhileStm), since it receives a boolean expression and a list of statements, it calls the Loop function with compB to compile the boolean expression and compile to compile the list of statements.

Lexer

In order to parse what the user wrote into an actual program, we first developed a lexer, which on receiving the user's code, will split it into a list of tokens.

These list of tokens consist of tokens that need to be isolated such as "(", ")", "+", "-", "*", "=", "<=", "==", ":=", and ";", as well as the other strings that are separated by an empty space while also not turning the space into a token (i.e. "this token" would be ["this","token"]).

To do this, we developed a function called *lexer*, which upon receiving a string, will iterate through it recursively.

First, it tries to find either "<=", "==" or ":=" on the first 2 chars, if it does, it will save them in the new list of strings and call lexer again without those first 2 chars.

Otherwise, it will look in the head of the string (the first char) for any of the other special characters as mentioned above, such as "(". If it does, it will also save it as a token and call the lexer again with the rest of the string. Alongside these tokens, we also look for empty spaces, but since we do not want to save them, we just call the lexer again without the space.

Lastly, if none of the above conditions are met, it means that it will be a variable or a number or one of the other keywords such as "not", so we use the function *takeWhile* along with *dropWhile*, in order to save every character that is alongside each other without any special token that should separate them.

With everything done, the function simply returns the list with every token separated so it can be built.

Data Builder

In order to parse the program, represented as a list of tokens thanks to the function *lexer*, into our data structure defined for the language, we used the function *buildData*, which makes use of buildStm, buildAexp and buildBexp to parse statements, arithmetic expressions and boolean expression, respectively.

The function **buildData** distinguishes between each statement by its ending semicolon, but never considers semicolons that appear between parentheses or inside *then* statements (from if statements), as those do not mark the end of the statements we want. If no valid semicolon is found to divide the statements, then the list of statements received is all between parentheses, which are removed and the list is analysed again.

Each statement selected by buildData is then handled by **buildStm**, which detects whether it's facing an if statement, while statement or assign, and reacts accordingly:

- If statements are separated into boolean expressions, *then* statements and *else* statements, and each is built by the proper function - buildBexp and buildData/buildStm. In *else* statements buildData and buildStm are used depending on whether there are one or multiple statements, for *then* statements, since a singular statement will contain a semicolon, both cases are handled by buildData, which takes care of removing the parentheses in the case of multiple statements and the semicolon in the case of one.
  An IfStm is created from these built statements and expressions.

- While statements are separated into boolean expression and *do* statement, and each is built by the proper function - buildBexp, buildData for multiple *do* statements and buildStm for one statement.
  A WhileStm is created from these built statements and expressions.
- Assign statements are separated into variable and arithmetic expressions, which is built by buildAexp.
  An AssignStm is created from this variable and built expressions.

In each division of the list of tokens, breakpoints such as ";", "if" or "do" are not considered if found between parentheses, since they would belong to an inner statement in that case. To handle that, we use the function **findNotInner**, which searches the list of tokens for the tokens given and tracks the depth of the current token, only considering a match in the search if the depth is zero, and returns its index. An increase in depth happens when the iteration finds an opening parenthesis (or a "then", needed when searching semicolons in buildData), and a decrease happens when it finds a closing parenthesis (or an "else"). Note that an assessment on the value of depth should never care for values other than zero, since if the list is given in reverse order, which does happen, the depth will go below zero, instead of above.

buildAexp is responsible for building any arithmetic expression present in a statement, and follows a recursive algorithm that builds the expression starting by the least precedence operator, whose hierarchy (from highest to lowest) is as follows:

- Anything between parentheses
- Multiplication
- Addition and Subtraction

Between two operators of the same level, the one on the left has higher precedence.

To achieve this, the function uses findNotInner to get the index of the last non nested "+" or "-" present in the expression and splits the list into two at that mark, each list obtained will be sent to buildAexp to be built and then added to the return value AddAexp/SubAexp, since the result of each should be added/subtracted in the expression last. Eventually through the recursion there will be no non nested "+" and "-" left, so it will search for the last non nested "*", separating the expression in two at that mark and sending them to buildAexp to be built, the obtained arithmetic expressions will then be added to the return value MultAexp. Finally, once there are no "*" left, there can be only a group of expressions between parentheses left, and, after removing the parentheses, those expressions are sent to buildAexp to be built.

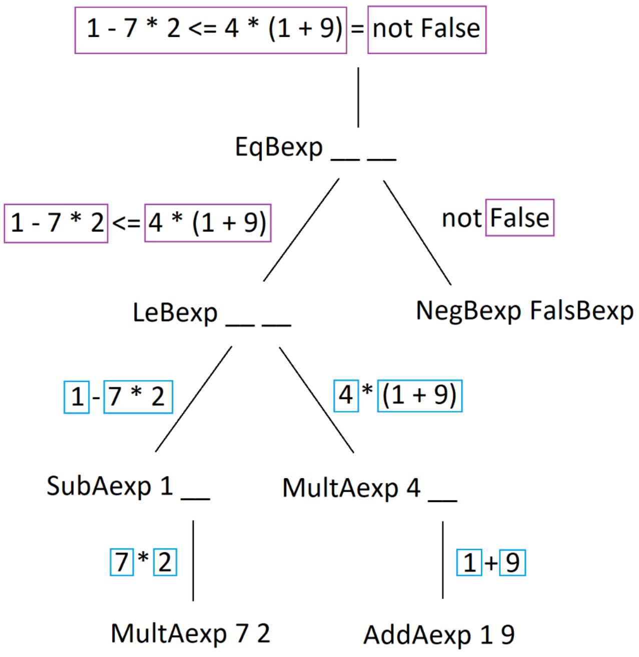
buildBexp is responsible for building any boolean expression present in a statement, and follows the same recursive algorithm that builds the expression starting by the least precedence operator, whose hierarchy (from highest to lowest) is as follows:

- Anything between parentheses
- Integer inequality (<=)
- Integer equality (==)
- Logical negation (not)
- Boolean equality (=)
- Logical conjunction (and)

Between two operators of the same level, the one on the left has higher precedence.

This function follows the same structure as buildAexp, where, with the help of findNotInner, it splits the boolean expression, with breakpoint at the operator with least precedence, and returns the Bexp relative to the operator in question, with the built boolean/arithmetic expressions of the lists obtained from the split.

Next is a visual representation of the logic tree of a boolean expression with arithmetic expressions:



EqBexp (LeBexp (SubAexp 1 (MultAexp 7 2)) (MultAexp 4 (AddAexp 1 9))) (NegBexp FalsBexp)