Haskell Low Level Machine

Group Project: T06_G10

Filipe de Azevedo Cardoso (**up202006409**)

- Contribution: 50%

José Pedro Almeida Batista de Sousa Santos (**up202108673**)

- Contribution: 50%

Part 1

Data Decisions

On part 1 two data structures were created. Stack and State:

- Stack needed to Handle both Booleans and Integers, once it could have the values "tt" and 1, 2, … . Therefore, a new data called StackElement was created, being defined as StackInt Int and StackString String. StackInt would handle all the Int data and StackInt all the Boolean or "tt"/"ff" (in it's final state). Finally, the type Stack was defined as a list of StackElements. Hence, a new data type was not needed, as Stack is just a ALIAS for [StackElement].
- State followed the same strategy. It needed to handle both Integers, Booleans and Strings. Thus, a data type called StateVariable (to handle variables) was created. In addition StateVariableVal, having the types StateVariableValInt and StateVariableValBool were created, as well, in order to handle the variables values (Int or Bool). To an extent, the State was a list of tuples of (StateVariable, StateVariableVal), so the type StateTuple was defined as mentioned and the type State was simply a list of StateTuple ([StateTuple]).

Functions Defined:

Due to the data types created functions to go to the vanilla type or accessing the desired variable value had to be created. As an example the functions are:

- State.hs

```haskell
-- Converts Bool to StateVariableValBool
boolToVariableVal :: Bool -> StateVariableVal

-- Converts StateVariableValInt to Int
variableValToInt :: StateVariableVal -> Int

-- Converts Int to StateVariableValInt
intToVariableVal :: Int -> StateVariableVal
```

```haskell
-- Converts Bool to StateVariableValBool
boolToVariableVal :: Bool -> StateVariableVal

-- Converts the StateVariable into a String
variableToString :: StateVariable -> String

-- Converts StateVariableVal into a String
stateVariableVal2Str :: StateVariableVal -> String
```

- Stack.hs

```haskell
-- Converts a StackElementInt back to Int
fromStackElementInt :: StackElement -> Int

-- Converts a StackElementString back to String
fromStackElementString :: StackElement -> String

-- Assigns a StackElementString to its correct Boolean Value
stackElementStringToBool :: String -> Bool

-- Once our data type uses strings we translate the "tt" and "ff" values to the expected ou
outputCorrectValue :: String -> String
```

A very important function had to be created in the case of the State Data Structure due to the fetch and store functions, who were hable to handle StateVariables that were both Int or Bool (fetch function).

```haskell
-- Detects if wheter the VariableValType is Bool or not
isBoolVariableValType :: StateVariableVal -> Bool
isBoolVariableValType (StateVariableValBool _) = True
isBoolVariableValType _ = False

-- pushes the value bound to var onto the stack
fetch :: StateVariable -> State -> Stack -> Stack
fetch var state stack
                | isBoolVariableValType variableval = pushBool (variableValToBool variablev
                | otherwise = pushInt (variableValToInt variableval) stack
                where variableval = getVariableVal var state
```

The same was done in the Stack data structure with the aid of the isNumber function in order to ease the store process. This hadn't to be done in the push function, once the assembler only pushed integer values through the push-x instruction and booleans through Tru and Fals instructions. So, it could be easily done just by creating a pushInt and pushBool function

```haskell
-- Detects if wheter the VariableValType is Bool or not
-- Verify if the element on the stack is a number
isNumber :: StackElement -> Bool
```

```haskell
isNumber (StackInt num) = True
isNumber _ = False

-- pops the topmost element of the stack and updates the State so that the popped value is b
store :: StateVariable -> Stack -> State -> (Stack, State)
store var stack state = (newstack, newstate)
        where newstack = pop(stack)
              newstate
                     | isNumber topelem = updateVariable var (intToVariableVal(fromStackEle
                     | otherwise = updateVariable var (boolToVariableVal(stackElementString
                     where topelem = top stack

-- Pushes an Int/Integer into the Stack
pushInt :: Integral a => a -> Stack -> Stack
pushInt num xs = StackInt (fromIntegral num) : xs

-- Pushes a String into the Stack
pushBool :: Bool -> Stack -> Stack
pushBool True xs = StackString "tt" : xs
pushBool False xs = StackString "ff" : xs
```

Part 2

Data Decisions

On the second part of the project we created the structures **Aexp**, **Bexp**, **Stm** and **Program**:

- `Aexp` was designed to handle arithmetic expressions. It includes Num for integers, Var for variables, and Add, Mult, Sub for arithmetic operations. Each operation takes two Aexp as operands, allowing for nested expressions.
- `Bexp` was created to handle boolean expressions. It includes True and False for boolean values, Not for negation, Equ and Eq for equality checks (between arithmetic and boolean expressions respectively), Le for less than operation on arithmetic expressions, and And for logical AND operation. Like Aexp, it allows for nested expressions.
- `Stm` was defined to represent statements. It includes Assign for assignment statements, If for if-else statements, and While for while loops. If and While can contain a list of Stm, allowing for nested statements and block structures.
- `Program` was defined as a list of Stm, representing a program as a sequence of statements.

Functions Defined

- Compiler.hs

```haskell
-- compA: Compiles an arithmetic expression into machine code.
compA :: Aexp -> Code

-- compB: Compiles a boolean expression into machine code.
compB :: Bexp -> Code

-- compile: Compiles a Program into machine code.
compile :: Program -> Code

-- lexer: Breaks down a string of code into a list of tokens.
lexer :: String -> [String]

-- parse: Parses a string of code into a Program.
parse :: String -> Program
```

The functions `parseStm`, `buildStm`, `findTarget`, `makeAExp`, and `makeBExp` are crucial to the functionality of our program as they are responsible for parsing and building the abstract syntax tree (AST).

- **`parseStm`**: This function serves as the entry point for parsing the source code. It takes a list of tokens and returns a list of statements. It iteratively calls `buildStm` to construct individual statements until no tokens remain. This function is essential as it transforms the raw source code into a structured AST, which can be further processed by the program.

```haskell
parseStm :: [Token] -> Program
parseStm tokens = ...
```

- **`buildStm`**: This function constructs a single statement from a list of tokens. It utilizes findTarget to locate the outermost occurrence of a target token in the token list, and uses this information to decide the type of statement to construct. For instance, if the target token is an assignment operator, buildStm will construct an assignment statement.

```haskell
buildStm :: [Token] -> Stm
buildStm tokens = ...
```

- **`findTarget`** is a helper function used by buildStm, and it identifies the outermost occurrence of a target token in a list of tokens. This function is crucial for correctly parsing nested expressions and statements.

```haskell
findTarget :: Token -> [Token] -> Int
findTarget target tokens = ...
```

- **`makeAExp`** and **`makeBExp`** are functions that construct arithmetic and boolean expressions, respectively, from a list of tokens. They are similar to buildStm, but they construct expressions instead of statements. These functions are crucial for parsing expressions within statements.

```haskell
makeAExp :: [Token] -> Aexp
makeAExp tokens = ...

makeBExp :: [Token] -> Bexp
makeBExp tokens = ...
```

Here is an example of how these functions might be used in our program:

```haskell
let tokens = lexer "if (x < 5) { x = x + 1; } else { x = x - 1; }"
let statements = parseStm tokens
```

In this example, `lexer` breaks down the source code into a list of tokens. `parseStm` then transforms this list of tokens into a list of statements (AST).