2nd PFL Project

Group T08_G02:

- Adriano Alexandre dos Santos Machado (up202105352) - 50%
- Tomás Alexandre Soeiro Vicente (up202108717) - 50%

Project Description

This project is divided into two parts. In the first part, we were asked to implement a low-level machine that supports arithmetic, boolean, and flow control instructions. Later, we were required to implement a compiler to compile an imperative language into the previously developed low-level machine.

Part 1: Implementation of a Low-Level Machine

Data Structure

Inst:

The Inst data structure represents machine instructions.

```
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
```

- Push n: inserts the value n into the stack
- Add: adds the two values at the top of the stack and inserts the result into the stack
- Mult: multiplies the two values at the top of the stack and inserts the result into the stack
- Sub: subtracts the value at the top of the stack from the 2nd value and inserts the result into the stack
- Tru: inserts the value tt into the stack
- Fals: inserts the value ff into the stack
- Equ: checks if the two values at the top of the stack are equal and inserts the result into the stack
- Le: checks if the value at the top of the stack is less than or equal to the second and inserts the result into the stack
- And: checks if the two values at the top of the stack are equal to tt and inserts the result into the stack
- Neg: inverts the boolean value at the top of the stack
- Fetch var: inserts the value associated with the variable var into the stack
- Store var: removes the value at the top of the stack and inserts the value associated with the variable var into the state
- Noop: Instruction with no effects
- Branch c1 c2: if the value at the top of the stack is tt, execute the list of instructions c1; otherwise, execute the list of instructions c2
- Loop c1 c2: execute c1, placing tt or ff on the top of the stack. If tt is at the top of the stack, execute c2 and then execute Loop c1 c2 again; otherwise, end execution

State:

The state is represented by a Binary Search Tree, where each node contains a key (corresponding to the name of a variable), an associated value, and two sub-trees.

```
data State = Empty
           | Node String String State State
```

This data structure supports the following operations:

- newState: creates a new empty state
- fromList: creates a new state from a list of pairs (variable, value)
- insert: inserts a new pair (variable, value) into the state
- load: returns the value associated with a variable
- toList: returns a list of pairs (variable, value) from a state
- toStr: returns a string from a state

Stack:

The stack is represented by a list of strings.

```
newtype Stack = Stk [String] deriving Show
```

In this data structure, the following operations exist:

- newStack: creates a new empty stack
- fromList: creates a new stack from a list of strings
- push: inserts a new string into the stack
- pop: removes the string at the top of the stack
- top: returns the string at the top of the stack
- isEmpty: checks if the stack is empty

Program Logic

The `run` function takes the arguments `(code, stack, state)`. As long as the code list is not empty, the `run` function executes the instruction at the top of the code list and recursively calls the `run` function with the remaining list of instructions.

Part 2: Compiler for an Imperative Language

In this part of the project, we were tasked with implementing a compiler for an imperative language. This process involved three main stages.

String	**x := 5; x := x - 1;**
Lexer Result	[TokVar "x",TokAssign,TokNumber 5,TokSemicolon,TokVar "x",TokAssign,TokVar "x",TokSub,TokNumber 1,TokSemicolon]
Parser Result	[AssignStm "x" (NumExp 5), AssignStm "x" (SubExp (VarExp "x") (NumExp 1))]
Compile Result	[Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"]

Lexer

Responsible for assigning tokens to each element of the language. In our case, the `lexer` function takes a string and returns a list of tokens.

Tokens: Our language supports the following tokens:

```
data Token = TokAssign          -- ':='
           | TokSemicolon       -- ';'
           | TokVar String      -- variable name
           | TokNumber Integer  -- number
           | TokOpenParen       -- '('
           | TokCloseParen      --')'
           | TokAdd             -- '+'
           | TokSub             -- '-'
           | TokMul             -- '*'
           | TokIf              -- 'if'
           | TokThen            -- 'then'
           | TokElse            -- 'else'
           | TokWhile           -- 'while'
           | TokDo              -- 'do'
           | TokBoolEqu         -- '='
           | TokIntEqu          -- '=='
           | TokLE              -- '<='
           | TokNot             -- 'not'
           | TokAnd             -- 'and'
           | TokTrue            -- 'True'
           | TokFalse           -- 'False'
           deriving (Show, Eq)
```

Our lexer works as follows:

- **Spaces:** Ignored.

- **Letters:** The `lexIdentifier` function checks if the string is a reserved word or a variable (starting with a lowercase letter) and returns the corresponding token.
- **Numbers:** The `lexNumber` function returns the TokNumber token (number).
- **Single-character operators, parentheses, and semicolons:** The `lexer` function adds the corresponding token to the list of tokens.
- **Operators with more than one character:** Operators like `:=`, `==`, `=`, and `<=` are handled by the `lexAssign`, `lexEqual`, and `lexLessEqual` functions, respectively.

After each of the previous steps, the `lexer` function calls itself recursively with the remaining string until the string is empty.

Parser

Responsible for transforming the list of tokens into a syntax tree. This is where we handle the precedence of operators. We define three different data structures to represent arithmetic expressions, boolean expressions, and statements.

```
data Aexp = NumExp Integer      -- Integer
          | VarExp String        -- Variable
          | AddExp Aexp Aexp    -- Addition
          | SubExp Aexp Aexp    -- Subtraction
          | MulExp Aexp Aexp    -- Multiplication
          deriving Show

data Bexp = TrueExp             -- True
          | FalseExp            -- False
          | EqArExp Aexp Aexp   -- Equality between two arithmetic expressions
          | EqBoolExp Bexp Bexp -- Equality between two boolean expressions
          | LeExp Aexp Aexp     -- Less than or equal between two arithmetic
expressions
          | NotExp Bexp         -- Negation of a boolean expression
          | AndExp Bexp Bexp    -- Conjunction between two boolean expressions
          deriving Show

data Stm = AssignStm String Aexp -- Assignment
          | SeqStm [Stm]          -- Sequence of statements
          | IfStm Bexp Stm Stm
          | WhileStm Bexp Stm
          deriving Show
```

The `parser` function applies the `lexer` function to the string and calls the `buildData` function with the resulting list of tokens. The `buildData` function takes this list of tokens and returns a syntax tree.

```
parser :: String -> Program
parser = buildData . lexer
```

The `buildData` function repeatedly calls the `parseStm` function until the list of remaining tokens is empty and checks if the list of tokens has been processed in its entirety. Otherwise, an error is thrown.

```
buildData :: [Token] -> Program
buildData tokens =
  case parseStm tokens of
    Just (stm, []) -> [stm]
    Just (stm, restTokens) -> stm : buildData restTokens
    _ -> error $ "Unexpected error parsing statement (buildData): " ++ show tokens
```

Instruction Parser

The `parseStm` function is responsible for processing statements, which can be of four different types: assignment, if-then-else, while, and sequence of statements. Depending on the type of statement, the `parseStm` function then calls the `parseAexp`, `parseBexp`, or `parseSeqStm` function, as seen in the following code excerpt.

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

parseStm :: [Token] -> Maybe (Stm, [Token])
parseStm tokens = case tokens of
  TokVar var : TokAssign : restTokens ->
    case parseAexp restTokens of
      Just (aexp, restTokens1) -> case restTokens1 of
        TokSemicolon : restTokens2 -> Just (AssignStm var aexp, restTokens2)
      ...

  TokIf : restTokens1 ->
    case parseBexp restTokens1 of
      ...
          case parseStm restTokens3 of
            ...
              case parseStm restTokens4 of
                Just (SeqStm stm2, TokSemicolon : restTokens5) ->
                  Just (IfStm bexp stm1 (SeqStm stm2), restTokens5)
                Just (SeqStm stm2, restTokens5) ->
                  error $ "Missing semicolon after 'else' statement" ++ show
restTokens5
                Just (stm2, restTokens5) ->
                  Just (IfStm bexp stm1 stm2, restTokens5)
                ...

  TokWhile : restTokens1 ->
    case parseBexp restTokens1 of
        ...
          case parseStm restTokens3 of
              Just (SeqStm stm, TokSemicolon : restTokens5) ->
                Just (WhileStm bexp (SeqStm stm), restTokens5)
```

```
                Just (SeqStm stm, restTokens5) ->
                  error $ "Missing semicolon after 'else' statement" ++ show
restTokens5
                Just (stm, restTokens5) ->
                  Just (WhileStm bexp stm, restTokens5)

  TokOpenParen : restTokens1 ->
    case parseSeqStm restTokens1 of
      case parseSeqStm restTokens1 of
        Just (stmList, restTokens2) -> Just (SeqStm stmList, restTokens2)

  _ -> error $ "Unexpected error parsing statement: " ++ show tokens
```

Arithmetic Expressions Parser

In parsing arithmetic functions, we use a set of auxiliary functions that allow us to handle the precedence of operators. The precedence of operators is handled as follows: first, expressions within parentheses are processed, followed by multiplications, and finally additions and subtractions.

The `parseAexp` function takes a list of tokens, calls the `parseSumOrDifOrProdOrIntOrPar` function, and checks if the list of tokens has been processed in its entirety. Otherwise, an error is thrown.

Boolean Expressions Parser

The `parseBexp` function takes a list of tokens, calls the `parseAndOrMore` function. If the list of tokens has been fully processed, the `parseBexp` function returns. Similarly, it checks if the first token not consumed by the parser is a `TokThen` or `TokDo`. If it is, the function returns with the remaining tokens. Otherwise, an error is thrown.

```haskell
parseBexp :: [Token] -> Maybe (Bexp, [Token])
parseBexp tokens = case parseAndOrMore tokens of
  Just (bexp, []) -> Just (bexp, [])
  Just (bexp, TokThen:rest) -> Just (bexp, TokThen:rest)
  Just (bexp, TokDo:rest) -> Just (bexp, TokDo:rest)
  Just ( _, rest) -> error $ "Unparsed tokens (parseB): " ++ show rest
  _ -> error $ "Unexpected error parsing boolean expression: " ++ show tokens
```

Compiler

The compiler is responsible for processing a list of ASTs, generating code for the low-level machine implemented in the first part of the project.

While this list is not empty, the `compile` function compiles the tree at the top of the list, invoking the `compA` or `compB` functions. After processing the instruction, the `compile` function is called recursively with the remaining list of instructions.

The `compA` function is responsible for processing arithmetic expressions (NumExp, VarExp, AddExp, SubExp, and MulExp), while the `compB` function is responsible for processing boolean expressions (TrueExp, FalseExp, EqArExp, EqBoolExp, LeExp, NotExp, and AndExp).

Code Execution

To execute the program, it is necessary to have GHC installed. After installation, simply execute the following command in the src folder:

```
ghci main.hs
```