Low-level Machine and Small Imperative Programming Language

Haskell Project

Programação Funcional e em Lógica T05_G07

- **Diogo Sarmento up202109663**
- **Rodrigo Póvoa up202108890**

Porto, 1 de Janeiro de 2024

Índice

Problem Description

This assignment revolves around creating a small imperative programming language and building an interpreter and compiler for it. The language includes arithmetic and boolean expressions, assignments, sequence of statements, if-else constructs, and while loops. We were asked to define data structures in Haskell to represent expressions and statements in this language, implement functions to compile programs written in this language into a list of machine instructions, and develop a parser to transform programs written in this language into our corresponding representation in the defined data structures.

Low Level Machine

Stack

Our Stack was designed to provide a simple and efficient way to manage a collection of elements in a Last-In-First-Out (LIFO) format. The following code will show the implementation of the Stack data structure in Haskell.

```haskell
module Stack (Stack, push, pop, top,  createEmptyStack, isEmpty) where

-- Define the Stack data type as a list
data Stack a = Stk [a] deriving Show

-- The push function adds an element to the top of the stack
push :: a -> Stack a -> Stack a
push x (Stk xs) = Stk (x:xs)

-- The pop function removes the top element from the stack
pop :: Stack a -> Stack a
pop (Stk (_:xs)) = Stk xs
pop _ = error "Stack.pop: empty stack"

-- The top function returns the top element of the stack
top :: Stack a -> a
top (Stk (x:_)) = x
top _ = error "Stack.top: empty stack"

-- The createEmptyStack function creates an empty stack
createEmptyStack :: Stack a
createEmptyStack = Stk []

-- The isEmpty function checks if the stack is empty
isEmpty :: Stack a -> Bool
isEmpty (Stk [])= True
isEmpty (Stk _) = False
```

State

The `State` module is designed to manage a state, which is a mapping from variable names to their values. Here are some details about the data and functions defined in the `State` module:

1. **DataType**: The `DataType` is a custom data type that can hold an integer (`Number Integer`), or one of two boolean values (`TT` for true, `FF` for false).

2. **State**: The `State` is a list of pairs, where each pair consists of a variable name and its value. For example, the state `[(x, 10), (y, TT)]` maps the variable `x` to the value `10` and the variable `y` to the value `TT`.

```
module State (State,DataType(..) ,createEmptyState) where

data DataType= Number Integer | TT | FF
    deriving (Eq, Show)

type State = [(String, DataType)]

-- The createEmptyState function creates an empty state
createEmptyState :: State
createEmptyState = []
```

stack2Str

The `stack2Str` function is used to convert a `Stack` of `DataType` to a `String`. Here's how it works:

```
convertStr :: DataType -> String
convertStr TT = "True"
convertStr FF = "False"
convertStr (Number n) = show n

-- Convert From Stack to String
stack2Str :: Stack DataType -> String
stack2Str stk
  | isEmpty stk = ""
  | otherwise = convertStr (top stk) ++ remainingStack
  where
    remainingStack = if isEmpty (pop stk) then "" else "," ++ stack2Str (pop stk)
```

The function does the following:

1. If the stack is empty, return an empty string.
2. If the stack is not empty, it converts the top element of the stack to a string using the `convertStr` function, and then concatenates this with the string representation of the remaining stack.

The `remainingStack` is defined in a `where` clause. It checks if the stack is empty after popping the top element. If it is, it returns an empty string. Otherwise, it adds a comma and then recursively calls `stack2Str` on the popped stack.

state2Str

The `state2Str` function is used to convert a `State` to a `String`. Here's how it works:

```haskell
-- Sort State, According to testAssembler cases
sortState :: State -> State
sortState [] = []
sortState (x:xs) = insert x (sortState xs)
  where
    insert x [] = [x]
    insert (var,value) ((var2,value2):xs)
      | var < var2 = (var,value):(var2,value2):xs
      | otherwise = (var2,value2):insert (var,value) xs

-- Convert From State to String
state2Str :: State -> String
state2Str state = state2StrSorted (sortState state)
  where
    state2StrSorted [] = ""
    state2StrSorted [(var, value)] = var ++ "=" ++ convertStr value
    state2StrSorted ((var, value):xs) = var ++ "=" ++ convertStr value ++ "," ++
state2Str xs
```

The `state2Str` function first sorts the state using the `sortState` function. The `sortState` function uses an insertion sort algorithm to sort the state by variable name.

After sorting the state, `state2Str` converts it to a string. It does this by recursively going through the state and for each variable-value pair, it concatenates the variable name, an equals sign, the string representation of the value, and a comma. The recursion stops when it reaches the last pair, where it doesn't add a comma at the end.

run

The `run` function is the main execution loop of your interpreter. It takes a tuple of `Code`, `Stack DataType`, and `State` as input and returns the same tuple as output after executing the code. Here's how it works:

```
-- Execute Instructions
exec :: (Code, Stack DataType, State) -> (Code, Stack DataType, State)

--Empty
exec ([], stack, state) = ([], stack, state)

-- Add Instruction
exec (Add:code, stack, state) = (code, push (Number (n1 + n2)) (pop (pop stack)),
state)
  where
    n1 = case top stack of
      Number n -> n
      _ -> error "Run-time error"
    n2 = case top (pop stack) of
      Number n -> n
      _ -> error "Run-time error"

-- Mult Instruction
exec (Mult:code, stack, state) = (code, push (Number (n1 * n2)) (pop (pop stack)),
state)
  where
    n1 = case top stack of
      Number n -> n
      _ -> error "Run-time error"
    n2 = case top (pop stack) of
      Number n -> n
      _ -> error "Run-time error"

-- Sub Instruction
exec (Sub:code, stack, state) = (code, push (Number (n1 - n2)) (pop (pop stack)),
state)
  where
    n1 = case top stack of
      Number n -> n
      _ -> error "Run-time error"
    n2 = case top (pop stack) of
      Number n -> n
      _ -> error "Run-time error"

--Equal Instruction
exec (Equ:code, stack, state) =
  case (top stack, top(pop stack)) of
    (Number n1, Number n2) -> (code, push (if n1 == n2 then TT else FF) (pop (pop
stack)), state)
    (TT, TT) -> (code, push TT (pop (pop stack)), state)
    (FF, FF) -> (code, push TT (pop (pop stack)), state)
    (TT, FF) -> (code, push FF (pop (pop stack)), state)
    (FF, TT) -> (code, push FF (pop (pop stack)), state)
    (_,_)-> error "Run-time error"

--Lower or Equal Instruction
exec(Le:code, stack, state) =
  case (top stack, top(pop stack)) of
    (Number n1, Number n2) -> (code, push (if n1 <= n2 then TT else FF) (pop (pop
stack)), state)
    (_,_)-> error "Run-time error"

--Push n Instruction
exec (Push n:code, stack, state) = (code, push (Number n) stack, state)

--Fetch x Instruction
exec(Fetch x:code, stack, state) = (code, push (stateLookup state x) stack, state)
  where
    stateLookup [] v = error ("Variable " ++ v ++ " not found")
    stateLookup ((var, value):xs) v
      | var == v = value
      | otherwise = stateLookup xs v

--Store x Instruction
exec (Store x:code, stack, state) = (code, pop stack, storeVal state (x, top
stack))
 where
    storeVal [] (x, value) = [(x, value)]
    storeVal ((var, value):xs) (x, value2)
      | var == x = (x, value2):xs
      | otherwise = (var, value):storeVal xs (x, value2)

--Negation Instruction
exec(Neg:code, stack, state) =
  case top stack of
      TT -> (code, push FF (pop stack), state)
      FF -> (code, push TT (pop stack), state)
      _  -> error "Run-time error"

--Tru Instruction
exec (Tru:code, stack, state) = (code, push TT stack, state)

--Fals Instruction
exec (Fals:code, stack, state) = (code, push FF stack, state)

--And Instruction
exec (And:code, stack, state)
   | isEmpty stack || isEmpty (pop stack) = error "Run-time error"
   | otherwise =
      case (top stack, top (pop stack)) of
        (TT, TT) -> (code, push TT (pop (pop stack)), state)
        (FF, FF) -> (code, push FF (pop (pop stack)), state)
        (TT, FF) -> (code, push FF (pop (pop stack)), state)
        (FF, TT) -> (code, push FF (pop (pop stack)), state)
        (_,_) -> error "Run-time error"

--Branch Instruction
exec (Branch code1 code2:code, stack, state) =
  case top stack of
    TT -> (code1 ++ code, pop stack, state)
    FF -> (code2 ++ code, pop stack, state)
```

```
        _ -> error "Run-time error"

--Loop Instruction
exec (Loop cond code:rest, stack, state) =
  exec (cond ++ [Branch (code ++ [Loop cond code]) [Noop]] ++ rest, stack, state)

--Noop Instruction
exec (Noop:code, stack, state) = (code, stack, state)

-- Run Machine
run :: (Code, Stack DataType, State) -> (Code, Stack DataType, State)
run ([],stack,state) = ([],stack,state)
run (code,stack,state) = run $ exec (code,stack,state)
```

The function uses pattern matching to check if the code is empty:

If the code is empty `([])`, it simply returns the current stack and state. This is the base case for the recursion and it means that all instructions have been executed.

If the code is not empty, it calls the `exec` function to execute the first instruction in the code. The `exec` function returns a new tuple of code, stack, and state, which is then passed recursively to the run function.

The `run` function effectively executes all instructions in the code, one by one, until there are no more instructions to execute.

The `exec` function can be defined as:

- Empty: Returns the same tuple.
- Add: Adds the top two elements of the stack and pushes the result back to the stack.
- Mult: Multiplies the top two elements of the stack and pushes the result back to the stack.
- Sub: Subtracts the top two elements of the stack and pushes the result back to the stack.
- Equ: Compares the top two elements of the stack and pushes the result back to the stack.
- Le: Compares the top two elements of the stack and pushes the result back to the stack.
- Push: Pushes the number n to the stack.
- Fetch: Pushes the value of the variable x to the stack.
- Store: Pops the top element of the stack and stores it in the variable x.
- Neg: Negates the top element of the stack.
- Tru: Pushes the boolean value True to the stack.
- Fals: Pushes the boolean value False to the stack.
- And: Compares the top two elements of the stack and pushes the result back to the stack.
- Branch: Executes the first code if the top element of the stack is True, otherwise executes the second code.
- Loop: Executes the first code while the top element of the stack is True.
- Noop: Does nothing.

Small Programming Language

Aexp, Bexp, Stm

The `Aexp`, `Bexp`, and `Stm` modules define the abstract syntax of the small programming language. Here are some details about the data and functions defined in these:

```haskell
-- Aexp: Arithmetic Expressions
data Aexp = Num Integer
            | Var String
            | Add Aexp Aexp
            | Mult Aexp Aexp
            | Sub Aexp Aexp
            deriving (Eq, Show)

-- Bexp: Boolean Expressions
data Bexp = TT
            | FF
            | Neg Bexp
            | And Bexp Bexp
            | Equ Aexp Aexp
            | EquB Bexp Bexp
            | Le Aexp Aexp
            deriving (Eq, Show)

-- Stm: Statements
data Stm = Assign String Aexp
            | If Bexp [Stm] [Stm]
            | While Bexp [Stm]
            deriving (Eq, Show)
```

1. **Aexp** : The `Aexp` data type defines the abstract syntax of arithmetic expressions. It can be a number (`Num Integer`), a variable (`Var String`), or an arithmetic operation (`Add Aexp Aexp`, `Mult Aexp Aexp`, or `Sub Aexp Aexp`).
2. **Bexp** : The `Bexp` data type defines the abstract syntax of boolean expressions. It can be a boolean value (`TT` or `FF`), a negation (`Neg Bexp`), a conjunction (`And Bexp Bexp`), an equality comparison (`Equ Aexp Aexp`), an equality comparison (`EquB Bexp Bexp`), or a less-than-or-equal-to comparison (`Le Aexp Aexp`).
3. **Stm** : The `Stm` data type defines the abstract syntax of statements. It can be an assignment (`Assign String Aexp`), an if statement (`If Bexp [Stm] [Stm]`), or a while loop (`While Bexp [Stm]`).

Compiler

The `Compiler` module defines the compiler that converts a `Stm` to a `Code`. Here are some details about the data and functions defined in this module:

```haskell
-- Compile Arithmetic Expressions
compA :: Aexp -> Code
compA (Num n) = [Push n]
compA (Var x) = [Fetch x]
compA (Main.Add a1 a2) = compA a1 ++ compA a2 ++ [LLM.Add]
compA (Main.Mult a1 a2) = compA a1 ++ compA a2 ++ [LLM.Mult]
compA (Main.Sub a1 a2) = compA a2 ++ compA a1 ++ [LLM.Sub]

-- Compile Boolean Expressions
compB :: Bexp -> Code
compB Main.TT = [LLM.Tru]
compB Main.FF = [LLM.Fals]
compB (Main.Neg b) = compB b ++ [LLM.Neg]
compB (Main.And b1 b2) = compB b1 ++ compB b2 ++ [LLM.And]
compB (Main.Equ a1 a2) = compA a1 ++ compA a2 ++ [LLM.Equ]
compB (Main.EquB b1 b2) = compB b1 ++ compB b2 ++ [LLM.Equ]
compB (Main.Le a1 a2) = compA a2 ++ compA a1 ++ [LLM.Le]

-- Compile Statements
compStm :: Stm -> Code
compStm (Assign x a) = compA a ++ [LLM.Store x]
compStm (Seq s1 s2) = compStm s1 ++ compStm s2
compStm (If b s1 s2) = compB b ++ [LLM.Branch (compile  s1) (compile s2)]
compStm (While b s) =  [LLM.Loop (compB b) (compile s)]


-- Compile Programs
compile :: [Stm] -> Code
compile [] = []
compile (s:ss) = compStm s ++ compile ss
```

1. **compA** : The `compA` function takes an arithmetic expression as input and returns a list of instructions that can be executed by the low-level machine. It does this by recursively going through the arithmetic expression and converting each sub-expression to a list of instructions. For example, the expression `Add (Num 1) (Num 2)` is converted to `[Push 1, Push 2, LLM.Add]`.

2. **compB** : The `compB` function takes a boolean expression as input and returns a list of instructions that can be executed by the low-level machine. It does this by recursively going through the boolean expression and converting each sub-expression to a list of instructions. For example, the expression `And TT FF` is converted to `[LLM.Tru, LLM.Fals, LLM.And]`.

3. **compStm** : The `compStm` function takes a statement as input and returns a list of instructions that can be executed by the low-level machine. It does this by recursively going through the statement and converting each sub-expression to a list of instructions. For example, the statement `Assign "x" (Add (Num 1) (Num 2))` is converted to `[Push 1, Push 2, LLM.Add, LLM.Store "x"]`.

4. **compile** : The `compile` function takes a list of statements as input and returns a list of instructions that can be executed by the low-level machine. It does this by recursively going through the list of statements and converting each statement to a list of instructions. For example, the list of statements `[Assign "x" (Add (Num 1) (Num 2)), Assign "y" (Mult (Var "x") (Num 3))]` is converted to `[Push 1, Push 2, LLM.Add, LLM.Store "x", Fetch "x", Push 3, LLM.Mult, LLM.Store "y"]`.

Parser

The `Parser` module defines the parser that converts a string to a `Stm`. Here are some details about the data and functions defined in this module:

```
languageDef =
  emptyDef {
              Token.identStart      = Parsec.letter,
              Token.identLetter     = Parsec.alphaNum,
              Token.reservedNames   = ["if", "then", "else", "while", "do", "not",
"and", "True", "False"],
              Token.reservedOpNames = ["+", "-", "*", ":=", "==", "<=", "and",
"not"]
           }

lexer = Token.makeTokenParser languageDef

identifier = Parsec.try $ do
    name <- Token.identifier lexer
    if any (`isInfixOf` name) (Token.reservedNames languageDef)
        then error $ "Variable name " ++ show name ++ " contains a reserved
keyword!"
        else return name
reserved   = Token.reserved   lexer
reservedOp = Token.reservedOp lexer
parens     = Token.parens     lexer
integer    = Token.integer    lexer
semi       = Token.semi       lexer
whiteSpace = Token.whiteSpace lexer

-- Precedence and associativity of arithmetic operators
aOperators = [
              [Infix (reservedOp "*" >> return Main.Mult) AssocLeft],
              [Infix (reservedOp "+" >> return Main.Add) AssocLeft],
              [Infix (reservedOp "-" >> return Main.Sub) AssocLeft]
             ]

-- Term parser for arithmetic expressions
aTerm = parens aexp
     Parsec.<|> liftM Var identifier
     Parsec.<|> liftM Num integer

-- Arithmetic expression parser
aexp = buildExpressionParser aOperators aTerm


-- Parser for Less than or Equal to
comparison :: Parser Bexp
comparison = do
  a1 <- aexp
  op <- reservedOp "<=" >> return Main.Le
  a2 <- aexp
  return (op a1 a2)
```

```haskell
-- Parser for == (equality)
equality :: Parser Bexp
equality = do
  a1 <- aexp
  op <- reservedOp "==" >> return Main.Equ
  a2 <- aexp
  return (op a1 a2)

-- Precedence and associativity of boolean operators
bOperators = [
              [Prefix (reservedOp "not" >> return Main.Neg)],
              [Infix (reservedOp "=" >> return Main.EquB) AssocLeft],
              [Infix (reservedOp "and" >> return Main.And) AssocLeft]
             ]

-- Term parser for boolean expressions
bTerm = parens bexp
       Parsec.<|> Parsec.try equality
       Parsec.<|> (reserved "True" >> return Main.TT)
       Parsec.<|> (reserved "False" >> return Main.FF)
       Parsec.<|> Parsec.try comparison


-- Boolean expression parser
bexp = buildExpressionParser bOperators bTerm

-- Parser for a list of statements
stms :: Parser [Stm]
stms = Parsec.many stm

-- Statement parser
stm :: Parser Stm
stm = Parsec.try stmA Parsec.<|> Parsec.try stmB Parsec.<|> Parsec.try stmIf
Parsec.<|> Parsec.try stmWhile

-- Statement parser for arithmetic expressions
stmA :: Parser Stm
stmA = do
    var <- identifier
    reservedOp ":="
    expr <- aexp
    semi
    return (Assign var expr)

thenstm :: Parser [Stm]
thenstm = parens (Parsec.many stm) Parsec.<|> liftM return stm

elsestm :: Parser [Stm]
elsestm = (parens (Parsec.many stm) <* semi) Parsec.<|> liftM return stm

-- Statement parser for if statements
stmIf :: Parser Stm
stmIf = do
    reserved "if"
    cond <- bexp
    reserved "then"
    stm1 <- thenstm
    reserved "else"
    stm2 <- elsestm
    return $ If cond (stm1) (stm2)

-- Statement parser for while loops
stmWhile :: Parser Stm
stmWhile = do
    reserved "while"
    cond <- bexp
    reserved "do"
    stm1 <- elsestm
    return $ While cond stm1

-- Parse a string into a list of statements
parse :: String -> [Stm]
parse str =
    case Parsec.parse (Parsec.spaces *> stms <* Parsec.eof) "" str of
        Left e  -> error $ show e
        Right r -> r
```

1. **languageDef and lexer**: These define the lexical structure of the language, including the valid characters for identifiers, reserved keywords, and operator names.

2. **identifier, reserved, reservedOp, parens, integer, semi, whiteSpace**: These are helper functions that use the lexer to parse identifiers, reserved keywords, operators, parentheses, integers, semicolons, and whitespace.

3. **aOperators, aTerm, aexp**: These define the parser for arithmetic expressions. The aOperators list defines the precedence and associativity of arithmetic operators. The aTerm function parses a term, which can be a variable, a number, or a parenthesized expression. The aexp function uses buildExpressionParser to combine these into a parser for full arithmetic expressions.

4. **comparison, equality, bOperators, bTerm, bexp**: These define the parser for boolean expressions in a similar way to the arithmetic expressions.

5. **stms, stm, stmA, stmIf, stmWhile**: These define the parser for statements. The stms function parses a list of statements. The stm function parses a single statement, which can be an assignment, an if statement, or a while loop. The stmA function parses an assignment statement. The stmIf function parses an if statement. The stmWhile function parses a while loop.

6. **parse**: This is the main function that takes a string as input and returns a list of statements. It does this by using the Parsec library to parse the string using the stms parser, and it expects the string to start and end with whitespace.

Conclusion

Throughout this project, we've built a simple interpreter for a custom programming language using Haskell. The language supports arithmetic and boolean expressions, variable assignments, if-else statements, and while loops.

We've implemented a parser using the Parsec library, which converts a string of code into a list of statements. The parser handles the lexical structure of the language, including identifiers, reserved keywords, operators, and whitespace. It also handles the syntax of the language, including the precedence and associativity of operators and the structure of expressions and statements.

We've also implemented a stack and a state to manage the execution of the code. The stack holds temporary values during the execution, and the state holds the values of variables. We've implemented functions to manipulate the stack and state, including pushing and popping values on the stack, and adding, updating, and looking up variables in the state.

Finally, we've implemented a run function that executes the code. It uses a recursive loop to execute each statement in the code, updating the stack and state as necessary.

Overall, this project demonstrates the power and flexibility of Haskell for implementing interpreters. It shows how Haskell's features, such as pattern matching, recursion, and higher-order functions, can be used to implement complex functionality with relatively simple and elegant code.