FEUP - PFL

This project explores Functional and Logic Programming TP2 using the Haskell language. It focuses on implementing the parsing and compiling of a simple imperative language.

Group T10_G09

- João Brandão Alves - up202108044 -> 50%
- Diogo Leandro Silva - up202105327 -> 50%

Part 1 - Assembler Overview

As we delve into the project, the assembler takes center stage as a critical component of the low-level machine. It encompasses the Code to be executed, an Evaluation Stack for handling integer numbers and booleans, and a Storage for managing variables and their corresponding values.

Data and Types Definition

We kick off the project by defining the Data and Types that will underpin our entire endeavor:

```
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]

data Element =
  Int Integer | Boolean Bool deriving Show

type Stack = [Element]
type State = [(String, Element)]
```

While the predefined data Inst and type Code come from the Specification file, we've introduced the data StackTypes to accommodate both Int (Integer) and Boolean (Bool), allowing for two distinct data types within the Stack, which is essentially a list of StackTypes.

The type State is a tuple consisting of String and StackTypes, reflecting the structure of the Stack and facilitating the association of a variable with an Integer/Boolean.

Relevant Functions

Let's see the essential functions and what they do:

```
createEmptyStack :: Stack
```

```
createEmptyStack :: Stack
createEmptyStack = []
```

This function creates an empty Stack.

`createEmptyState :: State`

```
createEmptyState :: State
createEmptyState = []
```

It generates an empty State.

`stack2Str :: Stack -> String`

```
stack2Str :: Stack -> String
stack2Str stack = intercalate "," (map showElement stack)
  where
    showElement (Int n) = show n
    showElement (Boolean True) = "True"
    showElement (Boolean False) = "False"
```

This function converts the Stack to a string, using the `showElement` auxiliary function.

`state2Str :: State -> String`

```
state2Str :: State -> String
state2Str state = intercalate "," [var ++ "=" ++ showElement val | (var, val) <-
sortOn fst state]
  where
    showElement (Int n) = show n
    showElement (Boolean True) = "True"
    showElement (Boolean False) = "False"
```

It converts the State to a string by sorting it based on the variable's name, with the aid of an auxiliary function `showElement`.

`run :: (Code, Stack, State) -> (Code, Stack, State)`

```
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack, state) = ([], stack, state)
run (instruction:rest, stack, state) = case instruction of
     -- Various cases and corresponding actions
     ...
```

The `run` function handles the execution of the Code and matches function calls to specific actions based on instructions. In case of a failure to match any function calls, a Run-time error is generated to ensure the inclusion of all potential and logical function calls.

Part 2 - Compiler Overview

In this section, we delve into the development of a compiler from a small imperative programming language, encompassing arithmetic and boolean expressions, as well as statements consisting of assignments, sequences, if-then-else statements, and while loops, into lists of instructions for the previously defined machine.

Data and Types Definition

The project involves the definition of Data and Types that will be utilized in this part:

```
data Aexp
  = IntExp Integer                -- Integer constant
  | VarExp String                 -- Variable reference
  | AddExp Aexp Aexp              -- Addition Aexpession
  | MulExp Aexp Aexp              -- Multiplication Aexpession
  | NegateExp Aexp                -- Negate a number
  deriving Show

data Bexp
  = TrueExp                       -- True constant
  | FalseExp                      -- False constant
  | NotExp Bexp                   -- Logical NOT
  | AndExp Bexp Bexp              -- Logical AND
  | EqExp Aexp Aexp               -- Equality comparison
  | EqBoolExp Bexp Bexp           -- Bool Equality comparison
  | LeExp Aexp Aexp               -- Less than or equal to comparison
  deriving Show

data Stm
  = Assign String Aexp            -- Assignment statement
  | IfThenElse Bexp [Stm] [Stm]   -- Conditional statement
  | While Bexp [Stm]              -- Loop statement
  deriving Show
```

The defined types include `Aexp` for arithmetic expressions (including constants), `Bexp` for boolean expressions (including constants), `Stm` for statements and structures (with support for multiple statements within if-then-else and while loops).

Relevant Functions

In the process of translating imperative programming language constructs into executable instructions for the defined machine, several key functions play a vital role. These functions are essential for parsing and

compiling different types of statements, including assignment statements, if-then-else statements, and while loops.

`parseStm`

The `parseStm` function is responsible for parsing various types of statements. It handles assignment statements, if-then-else statements, and while loops using pattern matching to extract the necessary components from the input token sequences.

`parseStmsUntilEndWhile`

The `parseStmsUntilEndWhile` function is used to parse a sequence of statements until the end of a while loop is encountered. It employs a stop condition to determine when the parsing process should stop, ensuring accurate handling of loop structures.

`parseDoWithOpenParen` and `parseDoWithoutOpenParen`

These functions handle the parsing of while loop statements, taking into account the presence or absence of an open parenthesis. They extract the condition and the body of the while loop and construct the corresponding `While` statement.

`parseAexp` and `parseBexp`

The `parseAexp` and `parseBexp` functions are dedicated to parsing arithmetic and boolean expressions, respectively. They leverage helper functions to handle different types of expressions, ensuring the correct parsing of complex expressions within the imperative language.

`processTokens`

The `processTokens` function plays a crucial role in processing tokens generated by the lexer. It ensures that specific tokens, such as `if`, `then`, `else`, `do`, and `while`, are appropriately handled by inserting additional tokens to facilitate the parsing of complex statement structures.

Examples

Assembler tests

1. Test Case: `testAssembler [Push 10,Push 4,Push 3,Sub,Mult]`
    - Expected Result: `("-10","") -> True`
2. Test Case: `testAssembler [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"]`
    - Expected Result: `("","a=3,someVar=False,var=True") -> True`
3. Test Case: `testAssembler [Fals,Store "var",Fetch "var"]`
    - Expected Result: `("False","var=False") -> True`
4. Test Case: `testAssembler [Push (-20),Tru,Fals]`

- Expected Result: `("False,True,-20","") -> True`

5. Test Case: `testAssembler [Push (-20),Tru,Tru,Neg]`
   - Expected Result: `("False,True,-20","") -> True`
6. Test Case: `testAssembler [Push (-20),Tru,Tru,Neg,Equ]`
   - Expected Result: `("False,-20","") -> True`
7. Test Case: `testAssembler [Push (-20),Push (-21), Le]`
   - Expected Result: `("True","") -> True`
8. Test Case: `testAssembler [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"]`
   - Expected Result: `("","x=4") -> True`
9. Test Case: `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"]]`
   - Expected Result: `("","fact=3628800,i=1") -> True`
10. Test Case: `testAssembler [Push 1,Push 2,And]`
    - Expected Result: `("*** Exception: Run-time error CallStack (from HasCallStack): error, called at main.hs:78:23 in main:Main`
11. Test Case: `testAssembler [Tru,Tru,Store "y", Fetch "x",Tru]`
    - Expected Result: `("*** Exception: Run-time error CallStack (from HasCallStack): error, called at main.hs:78:23 in main:Main`

Parser tests

1. Example: `testParser "x := 5; x := x - 1;"`
   - Expected Result: `("","x=4") -> True`
2. Example: `testParser "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;"`
   - Expected Result: `("","y=2") -> True`
3. Example: `testParser "x := 42; if x <= 43 then x := 1; else (x := 33; x := x+1;)"`
   - Expected Result: `("","x=1") -> Invalid input for statement`
   - Error happens do to the fact that else with brackets doesn't work
4. Example: `testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1;"`
   - Expected Result: `("","x=2") -> False`
   - Everythink is converted fine to toks, parsed fine, compiled fine and everything after else statement doesn't execute (bug)

5. Example: `testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1; z := x+x;"`
    - Expected Result: `("","x=2,z=4") -> False`
    - Everythink is converted fine to toks, parsed fine, compiled fine and everything after else statement doesn't execute (bug)
6. Example: `testParser "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);"`
    - Expected Result: `("","x=2,y=-10,z=6") -> True`
7. Example: `testParser "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i - 1;);"`
    - Expected Result: `("","fact=3628800,i=1") -> True`

Conclusions

In wrapping up, this project was a great hands-on learning experience for us. We got to explore into parsing, compiling, and implementing a small imperative language using Haskell, which really helped us grasp functional and logic programming concepts. We also learned about building an interpreter, compiling arithmetic and boolean expressions, and the vital role of thorough testing for reliable code. All in all, this project gave us a deeper understanding of parsing, compiling, and imperative programming in the context of Functional and Logic Programming with Haskell.