FEUP - PFL_TP2

Group Information

- Group Number: T13_G12
- Members:
  - Guilherme Miguel de Lima Freire (up202004809) - 50% Contribution
  - Simão Queirós Rodrigues (up202005700) - 50% Contribution

Segment 1: The Assembler

At the core of this segment is an assembler, serving as a foundational machine. It comprises an Execution Code, an Evaluation Stack for integers and booleans, and a Storage unit for variable management.

Initiating the project, we laid the groundwork with Data and Types that permeate our entire project:

```
 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]
type Stack = [(Integer, Bool)]
type State = [(String, (Integer, Bool))]
```

The data Inst and type Code remain as provided in the Specification.

Our own creation, the Stack, now accommodates a tuple of Integer and Bool, enabling a versatile datatype within the Stack, a list of such tuples.

State is similarly constructed, pairing a String with a tuple of Integer and Bool, thus allowing the binding of variables to values, be they Integer or Boolean.

Key functionalities have been developed:

- `createEmptyStack :: Stack` : Initiates an empty Stack.
- `stack2Str :: Stack -> String` : Transforms the Stack into a string representation.
- `createEmptyState :: State` : Initiates an empty State.
- `state2Str :: State -> String` : Transforms the State into a string representation.
- `run :: (Code, Stack, State) -> (Code, Stack, State)` : Processes the Code by matching function calls. Unmatched calls result in a run-time error, ensuring all logical calls are pre-handled.

Segment 2: The Compiler

Envisioning a compact imperative language inclusive of arithmetic and boolean expressions, alongside assignments, conditional sequences, and iterative loops.

This segment is dedicated to outlining a compiler that translates this language into instruction lists for the previously mentioned machine.

We've established Data and Types for use in this portion:

```
 data Aexp = Num Integer | VarExp String | AddExp Aexp Aexp
  | SubExp Aexp Aexp | MultExp Aexp Aexp  deriving Show
data Bexp = EquAExp Aexp Aexp | LeExp Aexp Aexp | AndExp Bexp Bexp
  | EquBoExp Bexp Bexp | NegExp Bexp | TruB | FalsB  deriving Show
data Stm = BranchExp Bexp [Stm] [Stm] | LoopExp Bexp [Stm]
  | Assign String Aexp deriving Show
type Program = [Stm]
```

Our implementation delves into critical functions such as:

- `compA :: Aexp -> Code` : Compiles arithmetic expressions into instructions.
- `compB :: Bexp -> Code` : Similar to `compA`, but for boolean expressions.
- `compile :: Program -> Code` : Translates a list of statements into a series of instructions.

The `lexer` and `parse` functions are integral for interpreting strings into executable programs, supporting the robust functionality of our compiler.

Our testing suite ensures the reliability and functionality of the assembler and compiler. This iterative testing guarantees the accuracy of our parsing and execution.