2º Projeto PFL

Grupo T08_G02:

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

Descrição do trabalho

Este projeto encontra-se dividido em duas partes. Num primeiro momento, foi-nos pedido que implementássemos uma máquina de baixo nível que suportasse instruções de cálculo aritmético, de cálculo booleano e de controlo de fluxo. Posteriormente, foi-nos solicitado a implementação de um compilador, com a finalidade de compilar uma linguagem imperativa para a máquina de baixo nível previamente desenvolvida.

Parte 1: Implementação de uma máquina de baixo nível

Estrutura de Dados

Inst:

A estrutura de dados Inst representa as instruções da máquina.

```
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: insere o valor n na stack
- Add: soma os dois valores no topo da stack e insere o resultado na stack
- Mult: multiplica os dois valores no topo da stack e insere o resultado no topo da stack
- Sub: subtrai o valor no topo da stack com o 2º valor e insere o resultado na stack
- Tru: insere o valor tt na stack
- Fals: insere o valor ff na stack
- Equ: verifica se os dois valores no topo da stack são iguais e insere o resultado na stack
- Le: verifica se valor no topo da stack é menor ou igual que o segundo e insere o resultado na stack
- And: verifica se os dois valores no topo da stack são iguais a tt e insere o resultado na stack
- Neg: inverte o valor do booleano que se encontra no topo da stack
- Fetch var: insere o valor associado à variável var na stack
- Store var: remove o valor no topo da stack e insere o valor associado à variável var no estado
- Noop: Instrução sem efeitos
- Branch c1 c2: se o valor no topo da stack for tt, executa a lista de instruções c1, caso contrário executa a lista de instruções c2
- Loop c1 c2: executa c1, colocando tt ou ff no topo da stack. Se tt estiver no topo da stack, executa c2 e volta a executar Loop c1 c2, caso contrário, termina a execução

State:

O estado é representado por uma Binary Search Tree, onde cada nó contém uma chave (que corresponderá ao nome de uma variável), um valor associado e duas sub-árvores.

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

Esta estrutura de dados suporta as seguintes operações:

- newState: cria um novo estado vazio
- fromList: cria um novo estado a partir de uma lista de pares (variável, valor)
- insert: insere um novo par (variável, valor) no estado
- load: retorna o valor associado a uma variável
- toList: retorna uma lista de pares (variável, valor) a partir de um estado
- toStr: retorna uma string a partir de um estado

Stack:

A stack é representada por uma lista de strings.

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

Nesta estrutura de dados existem as seguintes operações:

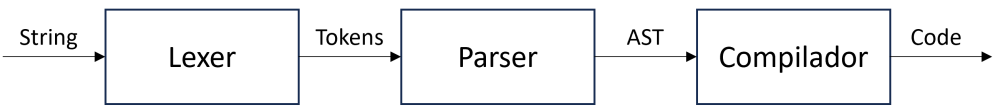
- newStack: cria uma nova stack vazia
- fromList: cria uma nova stack a partir de uma lista de strings
- push: insere uma nova string na stack
- pop: remove a string no topo da stack
- top: retorna a string no topo da stack
- isEmpty: verifica se a stack está vazia

Lógica do programa

A função `run` recebe os argumentos `(code, stack, state)`, enquanto o code não for uma lista vazia, a função `run` executa a instrução que se encontra no topo da lista code e chama recursivamente a função `run` com a lista de instruções restantes.

Parte 2: Compilador de uma linguagem imperativa

Nesta parte do projeto, foi-nos pedido que implementássemos um compilador para uma linguagem imperativa. Para tal, foram necessárias três etapas.



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

Responsável por atribuir tokens a cada elemento da linguagem. No nosso caso, a função `lexer` recebe uma string e retorna uma lista de tokens.

Tokens: A nossa linguagem suporta os seguintes tokens:

```
data Token = TokAssign          -- ':='
           | TokSemicolon       -- ';'
           | TokVar String      -- var 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)
```

O nosso lexer funciona da seguinte forma:

- **Espaços:** são ignorados
- **Letras:** a função `lexIdentifier` verifica se a string é uma palavra reservada ou uma variável(começada por uma letra minúscula) e retorna o token correspondente
- **Números:** a função `lexNumber` retorna o token TokNumber (número)
- **Operadores de um caracter, parênteses e ponto e vírgula:** a função `lexer` adiciona o token correspondente à lista de tokens

- **Operadores com mais do que um caracter:** operadores como o `:=`, `==`, `=` e o `<=` são tratados pelas funções `lexAssign`, `lexEqual` e `lexLessEqual` respetivamente

Após cada um dos passos anteriores, a função `lexer` chama-se recursivamente passando a string restante até que a string seja vazia.

Parser

Responsável por transformar a lista de tokens numa árvore sintática. É nesta etapa que tratamos a precedência dos operadores. Definimos três estruturas de dados distintas para representar expressões aritméticas, expressões booleanas e instruções.

```
data Aexp = NumExp Integer      -- Número inteiro
          | VarExp String        -- Variável
          | AddExp Aexp Aexp    -- Soma
          | SubExp Aexp Aexp    -- Subtração
          | MulExp Aexp Aexp    -- Multiplicação
          deriving Show

data Bexp = TrueExp             --Verdadeiro
          | FalseExp            --Falso
          | EqArExp Aexp Aexp   -- Igualdade entre duas expressões aritméticas
          | EqBoolExp Bexp Bexp -- Igualdade entre duas expressões booleanas
          | LeExp Aexp Aexp     -- Menor ou igual entre duas expressões
aritméticas
          | NotExp Bexp         -- Negação de uma expressão booleana
          | AndExp Bexp Bexp    -- Conjunção entre duas expressões booleanas
          deriving Show

data Stm = AssignStm String Aexp -- Atribuição
          | SeqStm [Stm]         -- Sequência de instruções
          | IfStm Bexp Stm Stm
          | WhileStm Bexp Stm
          deriving Show
```

A função `parser` aplica a função `lexer` à string e chama a função `buildData` com a lista de tokens resultante. A função `buildData` recebe essa lista de tokens e retorna uma árvore sintática.

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

A função `buildData` chama repetidamente a função `parseStm` até que a lista de tokens restantes seja vazia e verifica se a lista de tokens foi processada na totalidade. Caso contrário, é lançado um erro.

```
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
```

Parser de instruções

Por sua vez, a função `parseStm` é responsável por processar as instruções, que podem ser de quatro tipos distintos: atribuição, if-then-else, while e sequência de instruções. Dependendo do tipo de instrução a função `parseStm` chama depois a função `parseAexp`, `parseBexp` ou `parseSeqStm` como podemos observar do excerto de código seguinte.

```
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
```

Parser de expressões aritméticas

No parsing de funções aritméticas usamos um conjunto de funções auxiliares que nos permitem tratar a precedência dos operadores. A precedência dos operadores é tratada da seguinte forma: primeiro são processadas as expressões entre parênteses, depois as multiplicações e por fim as somas e subtrações.

A função `parseAexp` recebe uma lista de tokens, chama a função `parseSumOrDifOrProdOrIntOrPar` e verifica se a lista de tokens foi processada na totalidade. Caso contrário, é lançado um erro.

Parser de expressões booleanas

A função `parseBexp` recebe uma lista de tokens, chama a função `parseAndOrMore`. Se a lista de tokens tiver sido processada na totalidade, a função `parseBexp` retorna. De igual forma, verifica se o primeiro token não consumido pelo parser é um `TokThen` ou `TokDo`. Se for, a função retorna com o resto dos tokens. Caso contrário, é lançado um erro.

```
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
```

Compilador

O compilador será responsável pelo processamento de uma lista de ASTs, gerando o código para a máquina de baixo nível implementada na primeira parte do projeto.

Enquanto a esta lista não estiver vazia, a função `compile` compila a árvore no topo da lista, invocando as funções `compA` ou `compB`. Após o processamento da instrução, a função compile é chamada recursivamente com a lista de instruções restantes.

A função `compA` é responsável por processar as expressões aritméticas(NumExp, VarExp, AddExp, SubExp e MulExp) enquanto a função `compB` é responsável por processar as expressões booleanas(TrueExp, FalseExp, EqArExp, EqBoolExp, LeExp, NotExp e AndExp).

Execução do código

Para proceder à execução do programa, é necessário ter instalado o GHC. Após a instalação, basta executar o seguinte comando na pasta src:

```
ghci main.hs
```