I.. / Type / to search

Code Issues Pull requests Actions Projects Security Insights

main

pfl-proj2 / README.md

laraabastoss Update README.md ff55806 · now History

275 lines (192 loc) · 13.1 KB

Preview Code Blame Raw

Programação Funcional e Lógica - Projeto 2

Membros e contribuições

Grupo T01_G01

```
Lara Santos Bastos up202108740 ( Contribuição : 50% )
Lia Margarida Mota Sobral up202108741  ( Contribuição : 50% )
```

Introdução

O Projeto 2 de Programação Funcional e Lógica focou-se na implementação de um interpretador para uma linguagem de máquina específica, bem como na criação de um compilador para traduzir programas nessa linguagem. Este relatório aborda as estruturas de dados escolhidas, a implementação do interpretador, o processo de compilação e parse de programas, e conclui com reflexões sobre os desafios enfrentados.

Representação dos Dados e Tipos

De forma a facilitar o restante desenvolvimento do projeto, o primeiro passo deste concentrou-se na consideração da estrutura mais adequada para os Dados e Tipos utilizados.

Definimos assim as seguintes Estruturas de Dados:

- Instruções - representa todas as instruções possíveis.

```
data Inst =
  Push Integer | Add | Mult | Sub | Tru | Fals | Equ | Le | And | Neg |
  Branch Code Code | Loop Code Code
  deriving Show
```

- Bool Elements - representa elementos boleanos. Decidimos usar TT e FF, correspondentes a True e False tal como referido no enunciado.

```
data BoolElement = TT | FF
    deriving (Eq,Show)
```

- StackElement - inteiros ou BoolElements. Representa os elementos que a stack pode ter.

```
data StackElement =  Integer Integer |  BoolElement BoolElement
  deriving (Show)
```

- Aexp - representa espressões aritméticas. Suporta as operações adição, multiplicação, subtração e ainda VarA, uma string a representar o nome de uma variável, e Num a representar inteiros.

```
data Aexp = Add' Aexp Aexp | Mult' Aexp Aexp | Sub' Aexp Aexp | VarA Str
  deriving (Show)
```

- Bexp - representa espressões boleanas. Suporta as operações de igualdade de expressões bolenaos e aritméticas, menor our igual, negação e o and lógico. Pode ainda te um valor boleano (True ou False) ou uma variável a representar um boleano.

```
data Bexp = EqA Aexp Aexp | EqB Bexp Bexp |Le' Aexp Aexp | Not' Bexp | A
  deriving (Show)
```

- Stm - representa statements. Estas podem ser *assignment* de variávies, sequências de instruçoes, *if statements* e *while statements*.

```haskell
data Stm = AssignA String Aexp
         | AssignB String Bexp
         | Seq Stm Stm
         | If Bexp Stm Stm
         | While Bexp Stm
         | Aexp Aexp
         | Bexp Bexp
  deriving (Show)
```

Definimos de seguida, os seguintes novos tipos:

- Code - lista de instruções (Inst).

```haskell
type Code = [Inst]
```

- Stack - lista de "StackElements".

```haskell
type Stack = [StackElement]
```

- *State* - lista de pares de *strings* e *StackElements*.

```haskell
type State = [(String, StackElement)]
```

- Program - lista de statements (Stm).

```haskell
type Program = [Stm]
```

Por fim, definimos funções para inicar a *Stack* e *State* vazios:

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

createEmptyState :: State
createEmptyState = []
```

Parte 1

O propósito inicial da primeira fase do projeto consistia no desenvolvimento de um interpretador da linguagem da máquina descrita no enunciado. Para tal,era pedido que fosse implementado uma lista de instruções, uma pilha (do tipo Stack), e uma *storage* (do tipo *State*). O interpretador deveria percorrer a lista de instruções fornecidas resultando na devolução desta lista vazia e na preservação dos valores resultantes na *storage*.

Após definir os tipos e dados necessários, começamos por implementamos as funções "stack2Str" e "state2Str" que traduzem a *Stack* e o *State* em Strings respetivamentes.

```haskell
-- Converts a stack to its string representation
stack2Str :: Stack -> String
stack2Str [] = ""
stack2Str (Integer x : xs) = show x ++ rest
  where
    rest
      | null xs   = ""
      | otherwise = "," ++ stack2Str xs
stack2Str (BoolElement x : xs) = showBoolElement x ++ rest
  where
    rest
      | null xs   = ""
      | otherwise = "," ++ stack2Str xs

-- Converts a state to its string representation
state2Str :: State -> String
state2Str [] = ""
state2Str ((var, Integer val):xs) = var ++ "=" ++ show val ++ rest
  where
    rest
      | null xs   = ""
      | otherwise = "," ++ state2Str xs
state2Str ((var, BoolElement val):xs) = var ++ "=" ++ showBoolElement va
  where
    rest
      | null xs   = ""
      | otherwise = "," ++ state2Str xs
```

De seguida, desenvolvemos a função:

```haskell
-- Runs the stack machine with a given initial state
run :: (Code, Stack, State) -> (Code, Stack, State)
```

Esta processa cada instrução da lista de instruções de cada vez, e através de matching de padrões, reconhece que instrução realizar de forma a obter os valores esperados.

- Tru e Fals empilham os valores booleanos True ou False, respectivamente, na *Stack*.

- Add, Mult e Sub realizam a operação correspondente com os dois valores do topo da stack, empilhando o resultado.
- Equ e Le empilham o resultado da comparação dos últimos dois valores da *Stack* na forma de valores booleanos na *Stack*.
- And e Neg realizam as operações correspondentes com os valores booleanos do topo da *Stack*, empilhando o resultado.
- Push empilha o valor inteiro fornecido na *Stack*.
- Fetch vai buscar o valor associado à variável no topo de *State* e coloca-o no topo *Stack*, caso exista.
- Store atualiza o valor da variável ao valor do topo da *Stack* caso este já exista. Se a variável nao existir ainda, insere um novo túpulo (String, StackElement), mantendo a ordem alfabética das variáveis no *State*.
- Branch executa o código correspondente com base no valor booleano no topo da *Stack*.
- Noop avança para a próxima instrução.
- Loop cria um loop, executando repetidamente code1 enquanto a condição em code2 for verdadeira.

Em caso de falta de correspondência de padrões, a função retorna o erro "Run-time error", tal como pedido.

Para testar a primeira fase do projeto, utilizamos a função "testAssembler" fornecida no template:

```
-- Tests the stack machine with a given code
testAssembler :: Code -> (String, String)
testAssembler code = (stack2Str stack, state2Str state)
  where (_,stack,state) = run(code, createEmptyStack, createEmptyState)
```

Exemplo de Teste

Recebendo o input [Fals,Store "var",Fetch "var"] :

1 - A primeira instrução da lista, Fals, é consumida, resultando na colocação do valor booleano False na pilha.

Pilha: [False]

Stack: [False]

State: Vazio

2 - A segunda instrução, Store "var", é então executada, armazenando o valor presente no topo da pilha (que é False) na variável denominada "var".

Pilha: Vazia

Stack: [False]

State: [("var", False)]

3 - A terceira e ultima instrução, Fetch "var", busca o valor associado à variável "var" no estado e o coloca na pilha. Neste caso, o valor False é novamente empilhado.

Pilha: [False]

Stack: [False, False]

State: [("var", False)]

Parte 2

Por sua vez, a segunda parte do projeto destinava-se a receber uma string com código nesta linguagem, e passá-lo para uma lista de instruções, de forma a poder usar esta na função run, desenvolvida anteriormente. Esta parte é subdividida em duas funções principais: **compile** e **parse**.

Compile

A função **compile**, têm como objetivo receber uma lista de "Stm" e transformá-lo em instruções. Desta forma, vai processando cada elemento da lista, reconhece que statement está a processar (Assignments , Sequências de Instruçoes , If Statements e While Statements). No caso da statement conter uma expressão aritmética ou boleana, estas vão ser compiladas através das funções **compA** e **compB** respetivamente.

- Um número é compilado como uma instrução Push para inserir esse número na pilha.
- Uma variável é compilada como uma instrução Fetch para obter o valor associado a essa variável na memória.
- As operações aritméticas são compiladas chamando recursivamente compA para as subexpressões e adicionando instruções Add, Mult e Sub correspondentes.
- Um valor booleano é compilado como Tru ou Fals, respetivamente, para representar True ou False na pilha.
- Uma variável booleana é compilada como Fetch para obter o valor associado a essa variável na memória.
- As operações de comparação são compiladas chamando recursivamente compA ou

compB para as subexpressões e adicionando instruções Equ e Le.
- As operações lógicas são compiladas chamando recursivamente compB para as subexpressões e adicionando instruções And e Neg.
- Compila as expressões associadas às atribuições e adiciona instruções Store para armazenar os resultados na memória.
- Compila cada statement na sequência chamando recursivamente a função compile.
- Compila a expressão booleana, gera instruções Branch e compila os blocos "then" e "else".
- Compila a expressão booleana, gera instruções Loop e compila o bloco "do".

Parse

A função **parse** recebe uma string e transforma-a num programa (lista de "Stm"), para ser utilizada pelo compilador. Esta String, quando recebida na função, passa pelos seguintes passos:

1 - A função **lexer** divide esta string em tokens, que podem ser nomes de variávies, "(" ,")",";" ou operadores.

- Exemplo: lexer ″23 + 4 * 421″ = [″23″,″+″,″4″,″*″,″421″]

2 - A lista de token é passada de seguida para a função **parseProgram**. Esta identifica as "Stm" existentes no código:

- Ao encontrar um "if" o código é passado para a função **parseIfStm**. Esta encontra o próximo "then" pertencente ao if encontrado, verificando que entre estes há o mesmo número de parênteses abertos e fechados, e processa a expressão entre estas na função **parseBexp**. De seguida faz o mesmo entre o "then" e o "else" pertencente à if statement e processa a expressão na função **parseSeq**. Por fim faz o mesmo entre o "else" e o ";" correspondente.
- Ao encontrar um "while" o código é passado para a função **parseWhileStm**. Esta encontra o próximo "do" pertencente ao while encontrado, verificando que entre estes há o mesmo número de parênteses abertos e fechados, e processa a expressão entre estas na função **parseBexp**. De seguida faz o mesmo entre o "do" e o ";" correspondente à while statement e processa a sequência de instruções na função **parseSeq**.
- Ao encontrar o nome variável o código é passado para a função **parseAssignStm**. Esta processa a expressão que segue o ":=" na função **parseAexp**.
- Ao encontrar um "(" o código é passado para a função **parseSeq**. Esta começa por retirar os parênteses da sequência e processa cada statement existente na sequência (if, whiles, assignment ou outra sequência) chamando as função referidas previamente.

As funções **parseAexp** e **parseBexp** processam expressões artiméticas e boleanas respetivamente. De forma a ter em conta a precedência de operadores, começamos por processar as operações com menor prioridade. Caso tenham parênteses a delimitar a expressão, uma vez que ficam com prioridades, são processadas no fim, apendas quando todas as expressões sem parênteses tiverem sido processadas.

Exemplo:

- (1+2)*3 -> (Mult 3 (1+2)) -> (Mult 3 (Add 1 2))
- 1+2 * 3 -> (Add 1 (2 * 3)) -> (Add 1 (Mult 2 3))

Para auxiliar a implementar a lógica explicada anteriormente e lidar com parênteses criamos as funções:

- **parenthesesBalanced** - verifica se há o mesmo número de parêntese abertos e fechados numa lista de tokens.
- **removeOuterParentheses** - remove os parênteses a envolver um elemento
- **splitAtBalanced** - divide uma lista de tokens na primeira ocorrência de um target com parênteses equilibrados

Para testar a segunda fase do projeto, utilizamos a função "testParser" fornecida no template:

```
-- Tests the parser with a given program code
testParser :: String -> (String, String)
testParser programCode = (stack2Str stack, state2Str state)
  where (_,stack,state) = run(compile (parse programCode), createEmptySt
```

Exemplo de Teste

- Recebe o input "x := 5; x := x - 1";
- Após ser passar pelo **parse**: [(AssignA x 5),(AssignA x (Sub x 1))]
- Após ser passar pelo **compiler**: [Push 5, Store "x", Push 1, Fetch "x", Sub, Store "x"]
- Após passar pelo **run**: ("","x=4")

Conclusão

Após a conclusão do projeto, acreditamos que o seu desenvolvimento foi concluido com sucesso. Implementamos todas a funções pedidas e verificamos todos os casos de teste fornecidos no template. Adicionalmente, realizamos testes extra de forma a cobrir todos os casos possíveis e garantir a robustez do nosso programa. A implementação do parse revelou-se particularmente desafiante, no entanto acreditamos que não só ultrapassamos as nossas dificuldades com sucesso como consolidamos todos os conhecimentos de Haskell lecionados na Unidade Curricular.