PFL-TP2

Developers

Grupo	Nome	Up	Contribuição
T10_G08	Diogo Alexandre Figueiredo Gomes	up201905991	50%
T10_G08	Rafael Ferreira da Costa Silva Valquaresma	up202104805	50%

Descrição do projeto

O objetivo deste projeto foi desenvolver um assembler/inerpreter e um compiler para uma máquina de baixo nível com configurações do tipo (c, e, s) em que `c` é uma lista de instruções (code) a executar, `e` é a evaluation stack e `s` é a pilha de armazenamento(stack)

Instruções para instalação e execução do programa

- Ter instalado ou instalar o interpretador Haskell GHCI. O mesmo pode ser instalado **aqui**
- Transferir a pasta do programa e exportar
- Abrir um terminal, navegar até à pasta `src` e correr o comando `ghci Main.hs`
- Depois de abrir o ficheiro correr a função main através de `:main`

Código:

Parte 1

Code

O tipo de dados **Inst** representa as instruções suportadas por esta linguagem. Inclui as seguintes instruções:

```
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]
```

O tipo **Code** foi definido para representar a lista de instruções, ou seja, o código que irá ser lido e processado pelo nosso programa

Stack

Começamos por criar um novo tipo de dados a qual demos o nome de **StackElement** que representa os valores que compõem a nossa pilha, como **Integers** e **TT** e **FF**, que representam valores booleanos, **True** e **False**, respetivamente.

Para criarmos a nossa pilha criamos o novo tipo **Stack** que representa uma pilha, composta pelo tipo de dados que criámos **StackElement**.

Além disso criámos também 3 novas funções pertencentes à **Stack**:

- `stackElementValue::StackElement -> String`. Esta função serve para transdormar um **StackElement** em um valor imprimível no terminal.
- `createEmptyStack :: Stack`. Esta função, tal como o nome indica, cria uma pilha vazia.
- `stack2Str :: Stack -> String`. Esta função percorre a pilha e imprime-a no terminal, utilizando `,` para separar todos os seus valores. Usámos a função **intercalate** do módulo **Data.List** para essa finalidade.

State

O tipo **State** representa o armazenamento interno onde as variáveis são armazenadas. Para criar o mesmo, utilizamos uma lista de listas, sendo que as listas no seu interior recebem dois valores, a string a ser guardada e o seu valor do tipo `StackElement`.

Para este tipo criamos as seguintes funções:

- `createEmptyState :: State`. Como o próprio nome indica, cria um novo `State`
- `state2Str :: State -> String`. Tal como a função `stack2Str`, itera sobre a lista de estados e dá print à mesma na seguinte forma, "key=value" , ou seja no nosso caso igualando uma string/chave ao seu valor no programa, mas transformando o valor através da função `stackElementValue`. Estando o `State` ordenado pela chave alfabeticamente.

Assembler

Para implementar o interpretador neste programa, aplicámos a correspondência de padrões à função de execução, verificando o cabeçalho/header da lista **Code** para cada instrução possível e executando-a em conformidade sobre o resto da pilha como por exemplo:

```
run (Add:code, Number v1:Number v2:stack, state) = do --add the first two values
if stack not empty
```

```
    let newState = (code, (Number (v1+v2)):stack, state)
    run newState
```

Neste caso, o código corre corretamente se como input recebermos a operação que queremos utilizar e os dois primeiros valores da lista forem números. De seguida soma ambos os números e substitui esses valores no topo da pilha pela soma dos dois.

Também usámos a mesma estratégia de correspondência de padrões para verificar se a pilha tinha argumentos inválidos para uma determinada instrução e, em caso afirmativo, lançámos um **Run-time Error**, seguido por vezes da causa do erro em questão, como por exemplo:

```
run (Add:code, _:_:stack, state) = error "Run-time error."
```

Parte 2

- Para a segunda parte do nosso projeto utilizamos os tipos de dados e funções criadas acima e adicionamos o seguinte:

Program:

- Representado por tipos de dados como Aexp (expressões aritméticas), Bexp (expressões booleanas) e Stm (instruções).
- Aexp inclui operações como adição, subtração e multiplicação, bem como números e variáveis.
- Bexp inclui operações booleanas como NOT, AND, comparação menor ou igual e igualdade.
- Stm inclui instruções de atribuição, instruções condicionais (If Then Else) e instruções de loop (While).

Compiler:

- Utiliza correspondência de padrões para traduzir expressões em uma representação chamada Code, que é uma lista de instruções interpretáveis.
- Lida com a ordem de execução, especialmente ao lidar com expressões booleanas, garantindo que a segunda expressão seja calculada antes da primeira para corresponder à ordem de uma pilha Last-In-First-Out (LIFO).
- Possui funções específicas (como compA para expressões aritméticas e compB para expressões booleanas) que geram a sequência correta de instruções.

Parser:

- Utiliza uma função lexer para dividir o código-fonte em tokens.
- Implementa parsers para diferentes construções, como expressões aritméticas, booleanas, instruções if, instruções while e instruções de atribuição.
- Usa uma abordagem recursiva para processar aninhamentos de código em instruções if e while.

Process Flow:

- Começa com a função parseProgram, que analisa o código-fonte e direciona para análise específica com base nas construções encontradas (If Then Else, While Do, atribuição).

- As funções parseIf e parseWhile dividem o código em segmentos relevantes, lidando com blocos de código dentro das instruções if e while.
- As funções parseAexp e parseBexp analisam expressões aritméticas e booleanas, respectivamente, utilizando correspondência de padrões e recursividade.
- O resultado da análise é passado para a função compile, que mapeia as expressões para instruções interpretáveis.