Trabalho Prático nº1

Métodos de Programação I

2004/2005

Resumo

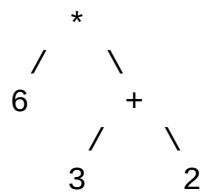
O objectivo deste trabalho é escrever um avaliador de expressões aritméticas, bem como um compilador de expressões aritméticas em código de uma máquina de *stack*, e um interpretador para esse código.

1 Preâmbulo

Este trabalho deve ser realizado por grupos de três alunos, e deve ser submetido de acordo com as instruções divulgadas na página da disciplina, antes da data limite aí mencionada.

2 Introdução

A representação interna natural para expressões aritméticas é por *árvores de sintaxe abstracta*, em que os nós correspondem a ocorrências de operadores aritméticos, cujas sub-árvores representam os respectivos operandos. Os valores (consideraremos apenas números inteiros) são guardados nas folhas da árvore. Por exemplo, a expressão $(2+3)*6$ seria naturalmente representada como se segue



É fácil imaginar como uma tal expressão pode ser representada e avaliada (i.e., calculado o seu valor) num computador moderno: basta definir um tipo de dados recursivo adequado para a representação das árvores; a avaliação de uma tal árvore é uma simples função recursiva.

Um outro processo importante, mas que deixamos fora do âmbito deste trabalho, é o *parsing* de expressões, que permite construír árvores de sintaxe. Por exemplo, o *parsing* da string `"6*(3+2)"`+ produz a árvore acima.[1]

[1] Este tópico será alvo de estudo em várias outras disciplinas da LESI e LMCC

Neste trabalho vamos no entanto imaginar uma segunda arquitectura, mais concretamente aquilo a que se chama usualmente uma *máquina de stack* (trata-se de um tipo de *máquina abstracta*). Esta máquina possui uma única forma de armazenar dados, numa pilha interna. Os programas da máquina consistem em sequências de instruções, podendo estas ser de dois tipos:

- `Push x`, coloca o valor `x` no topo da pilha da máquina.
- instruções aritméticas; por exemplo a instrução `ADD` retira da pilha os dois elementos de topo, adiciona-os, e coloca o resultado no topo da stack.

Vejamos passo a passo a execução do programa

```
[Push 2, Push 3, ADD, Push 6, MULT]

             -             -
            |3|           |6|
      -      -      -      -      --
     |2| -> |2| -> |5| -> |5| -> |30|
----------------------------------------
```

A pilha começa por estar vazia, evoluindo depois de acordo com cada instrução do programa.

O processo de conversão de um expressão (já representada por uma árvore) em código (i.e. uma sequência de instruções) de uma máquina concreta denomina-se *compilação*. No nosso exemplo, a compilação da árvore acima desenhada deveria produzir o programa que se executou, e cujo resultado foi o valor 30, o mesmo que resultaria da avaliação da árvore. Quando isto acontece, diz-se que o processo de compilação é *correcto*.

3 Trabalho a Realizar

Na secção 4 poderá encontrar a definição de um tipo de dados para ás árvores de sintaxe, bem com a função de avaliação dessas expressões.

1. Defina um tipo de dados para representar as instruções da máquina de *stack*.
2. Defina a função de compilação de expressões em código dessa máquina.
3. Defina uma função que execute uma instrução da máquina.
4. Escreva agora a função que efectua a execução completa de um programa, i.e., uma lista de instruções.
5. O código disponibilizado na secção 4 apresenta vários problemas. Por exemplo, o tipo de dados definido admite apenas operadores binários. Redefina agora todo o seu programa por forma a que a cada operador esteja associada uma *aridade*, i.e., um número de operandos

diferente. Isto permitirá por exemplo ter um operador de simetria ($-$ unário), bem como operadores com aridade arbitrária; por exemplo o seguinte operador tem aridade 3:

$$\mathrm{potsoma}(x, y, n) = (x + y)^n$$

Neste sentido, em cada nó de uma árvore de expressão deverá ser armazenado além do operador, uma *lista* de argumentos. O comprimento desta lista terá que coincidir com a aridade do operador, caso contrário a execução da função de avaliação resultará em erro.

Ilustre a sua solução definindo alguns operadores com aridades diversas.

Sugestão: recorra tanto quanto possível à utilização das funções `foldr`, `foldl`, e `map`.

4 Código Disponibilizado

```
data OP = SOMA | SUB | PROD | DIV

data Expressao = Folha Int | Nodo (OP, Expressao, Expressao)

aplica :: OP -> (Int, Int) -> Int
aplica SOMA (x,y) = x+y
aplica ...

avalia :: Expressao -> Int
avalia (Folha x) = x
avalia (Nodo (op, e1, e2)) = aplica op (avalia e1, avalia e2)
```

5 Valorização

Implemente um *parser* para expressões aritméticas. Para isso estude em detalhe a classe `Read` do Haskell, e defina uma instância `Expressao` dessa classe:

```
instance Read Expressao
  where ...
```