Trabalho 1: Compilador para `MSP`

Métodos de Programação 1

6 de Outubro de 2006

1 Introdução

O objectivo deste trabalho é desenvolver um compilador para uma pequena linguagem de programação designada `Prog`. Nesta linguagem só existem duas instruções:

- **let** $x = e$, onde x é uma variável e e uma expressão aritmética sobre inteiros que lhe é atribuída.
- **print** e, que imprime a expressão aritmética e.

As instruções podem ser sequenciadas com ";" para compor programas mais sofisticados, como por exemplo:

$$\textbf{let}\ x = 2; \textbf{print}\ (-3) * (4 + x)$$

Os seguintes tipos Haskell vão ser usado representar estes programas.

```
data Prog o = Print (Exp o) | Let String (Exp o) | Seq (Prog o) (Prog o)
data Exp  o = Const Int | Var String | Op o [Exp o]
```

Uma expressão aritmética pode ser uma constante, uma variável ou a aplicação de um operador a uma lista de argumentos. Ambos os tipos são parametrizados pelo tipo dos operadores aritméticos, por forma a tornar o compilador independente do conjunto concreto de operadores a usar. Por exemplo, se apenas pretendermos ter soma, produto e simétrico, podemos declarar o seguinte tipo para os operadores.

```
data Ops = Add | Mul | Sim
```

O pequeno programa acima apresentado poderia agora ser representado pelo seguinte valor:

```
prog :: Prog Ops
prog = Seq (Let "x" (Const 2))
           (Print (Op Mul [Op Sim [Const 3],
                           Op Add [Const 4, Var "x"]]))
```

É possível definir uma função para executar directamente programas `Prog`. Para tal, vamos começar por definir uma função para avaliar o valor das expressões aritméticas. Como as expressões podem conter variáveis precisamos de

um dicionário que determine o seu valor. Este dicionário é uma correspondência entre variáveis e valores e pode ser implementado com o tipo *Map* da biblioteca *Data.Map*:

```
type VarDict = Map String Int
```

Também é necessário saber qual a função Haskell que corresponde a cada um dos nossos operadores. Para tal vamos definir uma classe *Opt* que agrupa todos os operadores.

```
class Opt o where
  arity :: o → Int
  func  :: o → ([Int] → Int)
```

A primeira função desta classe diz-nos qual a aridade de um operador. A segunda devolve uma função que pode ser usada para o executar. A instância desta classe para o nosso tipo *Ops* é a seguinte:

```
instance Opt Ops where
  arity Add = 2
  arity Mul = 2
  arity Sim = 1
  func  Add = λ[x, y] → x + y
  func  Mul = λ[x, y] → x * y
  func  Sim = λ[x]    → −x
```

É agora possível definir a função *eval* que avalia o valor de uma expressão.

```
eval :: Opt o ⇒ VarDict → Exp o → Int
eval vd (Const x) = x
eval vd (Var v)   = vd ! v
eval vd (Op o l)  = func o (map (eval vd) l)
```

A função que executa um programa em `Prog` tem que actualizar um dicionário sempre que encontra uma instrução **let**. Este dicionário é depois passado à função *eval* sempre que uma expressão é encontrada. A forma mais simples de implementar este algoritmo passa pela utilização do *monad* estado da biblioteca *Control.Monad.State*. No entanto, como também precisamos do *monad IO* (para executar a instrução **print**) temos que combinar os dois recorrendo ao transformador de *monads* $StateT$.

```
execprog :: Opt o ⇒ Prog o → IO ()
execprog p = evalStateT (aux p) empty
  where aux :: Opt o ⇒ Prog o → StateT VarDict IO ()
    aux (Print e) = do vd ← get
                       lift (print (eval vd e))
    aux (Let v e) = do vd ← get
                       put (insert v (eval vd e) vd)
    aux (Seq l r) = do aux l
                       aux r
```

Podemos agora usar a função *execprog* para executar o nosso programa.

```
> execprog prog
− 18
```

2 A máquina abstracta `MSP`

Em vez de executar directamente estes programas pretende-se agora desenvolver um compilador para a máquina abstracta `MSP` (*Mais Simples Possível*). Será conveniente conhecer bem a `MSP`, pelo que se recomenda a leitura do respectivo manual, disponibilizado no *kit* do trabalho.

A `MSP` possui duas áreas de memória: a *heap* que serve para armazenar valores usados pelo programa (no caso da nossa linguagem as variáveis) e a *stack* que é uma memória auxiliar à execução das instruções (e que funciona segundo a filosofia habitual deste tipo de dados). Para simplificar, vamos assumir que ambas as áreas de memória são listas de inteiros: a *heap* é uma lista de tamanho fixo igual ao tamanho desejado e a *stack* tem tamanho variável, sendo as inserções e remoções feitas na cabeça da lista. Podemos representar a memória pelo seguinte tipo de dados:

$$\textbf{data}\ Mem = Mem\{\, stack :: [Int], heap :: [Int]\,\}$$

Assumindo que a *heap* tem 1024 posições de memória, o estado inicial da memória é representado pelo seguinte valor.

$$\begin{aligned} &emptymem :: Mem \\ &emptymem = Mem\{\, stack = [\,], heap = replicate\ 1024\ \bot\,\} \end{aligned}$$

O conjunto de instruções da máquina `MSP` que nos interessa para este trabalho é bastante restricto:

PUSH x Acrescenta o inteiro x ao topo da *stack*.

LOAD Retira um endereço do topo da *stack* e substitui-o pelo o conteúdo desse endereço na *heap*.

STORE Retira um valor e um endereço do topo da *stack* e armazena esse valor no respectivo endereço da *heap*.

IN Lê um inteiro do teclado e armazena-o no topo da *stack*.

OUT Retira um valor do topo da *stack* e imprime-o no monitor.

OP o Dado um operador o, retira os seus argumentos do topo da *stack* e substitui-os pelo resultado da operação respectiva.

Estas instruções podem ser representadas pelo seguinte tipo de dados:

$$\textbf{data}\ Instr\ o = PUSH\ Int \mid LOAD \mid STORE \mid IN \mid OUT \mid OP\ o$$

Um programa em `MSP` é simplesmente uma sequência destas instruções:

$$\textbf{type}\ MSP\ o = [Instr\ o]$$

Por exemplo, o seguinte programa imprime o resultado de somar 2 com 3 no monitor.

$$\begin{aligned} &msp :: MSP\ Ops \\ &msp = [PUSH\ 3, PUSH\ 2, OP\ Add, OUT] \end{aligned}$$

É possível compilar programas escritos em `Prog` para `MSP`. Durante a compilação é necessário atribuir um endereço na *heap* para cada variável declarada no programa. Supondo que a variável x foi alocada ao endereço 0, o programa exemplo acima apresentado seria compilado no seguinte programa `MSP`.

$$\begin{aligned} &mspprog :: MSP\ Ops \\ &mspprog = [PUSH\ 0, PUSH\ 2, STORE, PUSH\ 3, OP\ Sim, \\ &\qquad PUSH\ 4, PUSH\ 0, LOAD, OP\ Sub, OP\ Mul, OUT] \end{aligned}$$

As 3 primeiras instruções deste programa servem para armazenar o valor 2 no endereço 0 atribuído à variável x. Como é habitual nas máquinas de *stack*, para realizar uma dada operação é necessário fazer antes o *push* dos seus argumentos. Este facto implica que a compilação de uma expressão seja feita através de uma travessia *postorder* da sua árvore sintática.

3 Tarefas a realizar

Neste trabalho deverão ser implementadas no mínimo as seguintes funções:

- $execmsp :: Opt\ o \Rightarrow MSP\ o \rightarrow IO\ ()$
  Esta função é responsável por executar um programa `MSP`. Pretende-se que seja usada uma combinação dos *monads State* e *IO* para o fazer, sendo o estado a memória da máquina em cada instante.

- $compile :: Opt\ o \Rightarrow Prog\ o \rightarrow MSP\ o$
  Esta função deve compilar programas escritos em `Prog` para `MSP`.

Estas funções devem satisfazer o seguinte critério de correcção, que obriga a que o resultado de executar um programa antes e depois da compilação seja idêntico.

$$execprog = execmsp \circ compile$$

Para obter nota de Bom ou superior devem implementar algumas funcionalidades extra, como por exemplo:

- Implementar instâncias das classes *Show* e *Read* para os tipos *Prog* e *MSP* e criar programas executáveis para compilar um programa de `Prog` para `MSP` e para executar os programas resultantes.

- Acrescentar funcionalidades de *debug* às funções *compile* e *execmsp*. Existem vários erros de compilação e execução que podem ocorrer: operadores com número de argumentos errado, variáveis não definidas, endereços de memória errados, tentativa de retirar valores de uma *stack* vazia, etc. O tratamento de erros deve ser feito usando o *monad Error*.

- Estender o conjunto de instruções da máquina virtual `MSP` e da linguagem de programação `Prog`, por forma a obter uma linguagem `Prog++`. Esta linguagem pode, por exemplo, conter mais operadores aritméticos, instruções de leitura do teclado, mais tipos de dados, etc. Se desejar, pode alterar a classe *Opt* ou mesmo usar uma estratégia diferente para representar os operadores.

O trabalho deve ser entregue no *site* respectivo até ao dia 5 de Novembro. Deve ser realizado em grupos de 3 alunos e o relatório escrito em *literate* Haskell. Boa Sorte!