Processamento de Linguagens e Compiladores

LMCC, Universidade do Minho

Ano lectivo 2006/2007

João Saraiva

Ficha Teórico-Prática N°11

Este texto está escrito em **literate Haskell**. Isto é, pode ser interpretado como um documento LaTeX ou como um puro programa na linguagem Haskell. Responda às perguntas sobre Haskell neste próprio ficheiro para assim produzir o programa e a sua documentação.

1 MSP em Haskell

Solução

```
--
--
-- Processamento de Linguagens e Compilação
-- 2006/2007
--


module Msp where

import Data.Char
import Data.List
-- import LRC_Pretty                                  -- Biblioteca de Combiandores
                                                    -- de pretty printing
```

A estrutura abstracta de MSP é definida pelo seguinte tipo de dados algébrico:

Solução

```
data Msp   = Msp [Decl] [Instr]

data Decl  = Decl String Integer Integer

data Instr = ALabel String      -- label

           | Call String        -- call a function
           | Ret                -- return from a function

           | Add                -- arithmetic and boolean instructions
           | And
           | Div
           | Eq
           | Gt
           | Lt
           | Minus
           | Mul
           | Neq
           | Not
           | Or
           | Sub

           | Halt                -- Halt the machine

           | IIn                 -- IO
           | IOut
           | Jump  String        -- Jump Instructions
           | Jumpf String

           | Pusha String        -- Stack Operations
           | Pushi Integer
           | Load
           | Store
   deriving (Show , Eq , Ord)
```

Um exemplo de um programa MSP em notação abstracta é seguinte:

Solução

```
prog1 = Msp [ Decl "a1" 0  10
            , Decl "a2" 10  1 ]
            [ Pushi 12
            , IOut
            , Halt ]
```

1.1 *Escreva um programa MSP directamente em Haskell para modelar o seguinte programa na linguagem* C^2_{--}*:*

```
int aux;
int f;
f = 3 * 4;
aux = f + 4;
```

Solução

1.2 *Utilize os combinadores de Pretty Print apresentados na aula anterior de modo a ter uma função que produz MSP em notação concreta e alindada.*

Solução

---

2 Máquina Virtual de MSP em Haskell

2.1 Stack

Solução

```
emptyStack = []

-- push :: Int -> [Int] -> [Int]
push v stack = v : stack

pop  []     = error "Pop of an empty stack"
pop  (h:t)  = t

top []      = error "Top of an empty stack!"
top (h:t)   = h
```

---

3 Symbols

Solução

```
type Symbol = (String    -- name
              ,Int       -- size
              ,Int)      -- address in the heap

-- lookupSymb :: String -> [Symbol] -> Symbol
lookupSymb n []     =  error "Symbol not in the heap!"
lookupSymb n (h:t)  |  n == n'    = h
                    |  otherwise  = lookupSymb n t
  where (n',s,a) = h
```

3.1 Heap

Solução

```
-- allocMem :: [b] -> Int -> [b]
allocMem mem nbytes = mem ++ (map (~ -> 0) [1..nbytes])


-- allocMem mem nbytes = mem ++ (replicate nbytes 0)


--updateMemAddress :: [a] -> Int -> a -> [a]
updateMemAddress (h:t) 0 v = v:t
updateMemAddress (h:t) i v = h : updateMemAddress t (i-1) v


-- getMemAddress :: [a] -> Int -> a
getMemAddress mem address = ith mem  address

ith (h:t) 0 = h
ith (h:t) n = ith t (n-1)
```

3.2 Debug e Trace

Solução

```
debug p (stack,heap,symbs) =
  do  putStrLn ""
      putStrLn ("Instruction: " ++ (show $ head p))
      putStrLn ("Stack      : " ++ (show stack))
      putStrLn ("Heap       : " ++ (show heap))
--      getChar
```

3.3 A Máquina Virtual haMsp

Solução

```
haMsp prog = runMspProg prog


runMspProg (Msp decls instr) = runMSP instr instr initialState
  where (heap,symb)  = runMspDecls decls ([],[])
        initialState = (emptyStack,heap,symb)
```

As declarações de variáveis da heap são armazenadas numa tabela de símbolos.

Solução

```
runMspDecls ((Decl n a s) : t) (heap,symbs) =  runMspDecls t (heap',symbs')
  where  symbs' = (n,a,s) : symbs
         heap'  = allocMem heap s

runMspDecls [] (heap,symbs)  = (heap,symbs)
```

Instruções para parar a máquina:

Solução

```
runMSP prog [] state = return state

runMSP prog (Halt :t) (stack,heap,symbs) =
    do putStrLn (show stack)
       putStrLn (show heap)
       putStrLn (show symbs)
       return (stack,heap,symbs)
```

3.4 Stack Instructions

Solução

```
runMSP prog p@(Pushi i:t) state =
    do  debug p state
        let (stack,heap,symbs)  =  state
        let state'              =  (push i stack , heap , symbs)
        runMSP prog t state'

runMSP prog p@(Pusha n :t) state =
    do  debug p state
        let (stack,heap,symbs)  =  state
        let  (n',a,s)           =  lookupSymb n symbs
        let  state'             =  (push a stack , heap , symbs)
        runMSP prog t state'

runMSP prog p@(Store :t) state  =
    do  debug p state
        let (stack,heap,symbs)  =  state
        let v                   =  top stack
        let stack'              =  pop stack
        let address             =  top stack'
        let heap'               =  updateMemAddress heap address v
        runMSP prog t (pop stack' , heap' , symbs)

runMSP prog p@(Load :t) state  =
    do  debug p state
        let (stack,heap,symbs)  =  state
        let address             =  top stack
        let v                   =  getMemAddress heap address
        let stack'              =  push v (pop stack)
        runMSP prog t (stack' , heap , symbs)
```

Calling Functions Solução

```
runMSP prog p@(Call n:t) state =
    do  debug p state
        let (stack,heap,symbs)  =  state
        let pc       =  npc (length prog)  (length t)
        let stack'   =  push pc stack
        jmp prog n (stack',heap,symbs)


runMSP prog p@(Ret : t) state =
    do  debug p state
        let (stack,heap,symbs)  =  state
        let v                   =  top stack
        let stack'              =  pop stack
        let prog'               =  drop (toInt' v) prog
        runMSP prog prog' (stack',heap,symbs)
```

IO Instructions   Solução

```
runMSP prog p@(IOut :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        putStrLn (show $ top stack)
        runMSP prog t (pop stack , heap , symbs)


runMSP prog (IIn :t) (stack,heap,symbs) =
    do  putStrLn ("Introduza um inteiro:")
        v <- getLine
        let v' = (read v):: Integer

        runMSP prog t (push v' stack , heap , symbs)
```

Arithmetic Instructions   Solução

```
runMSP prog p@(Add :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let stack’’’      =  push (op1 + op2) stack’’
        runMSP prog t (stack’’’ , heap , symbs)

runMSP prog p@(Mul :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let stack’’’      =  push (op1 * op2) stack’’
        runMSP prog t (stack’’’ , heap , symbs)

runMSP prog p@(Sub :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let stack’’’      =  push (op2 - op1) stack’’
        runMSP prog t (stack’’’ , heap , symbs)

runMSP prog p@(Div :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let stack’’’      =  push (op2 ‘div‘ op1) stack’’
        runMSP prog t (stack’’’ , heap,symbs)


runMSP prog p@(Eq :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let v  = if op1 == op2 then 1 else 0
        let stack’’’      =  push v stack’’
        runMSP prog t (stack’’’ , heap , symbs)

runMSP prog p@(Neq :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let v  = if op1 == op2 then 0 else 1
        let stack’’’      =  push v stack’’
        runMSP prog t (stack’’’ , heap, symbs)


runMSP prog p@(Gt :t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (op1:stack’)  =  stack
        let (op2:stack’’) =  stack’
        let v  = if op1 < op2 then 1 else 0
```

Jump Instructions Solução

```
runMSP prog p@(Jump l:t) state =
    do  debug p state
        jmp prog l state

runMSP prog p@(Jumpf l:t) (stack,heap,symbs) =
    do  debug p (stack,heap,symbs)
        let (v:stack')  =  stack
        if v == 1 then runMSP prog t (stack',heap,symbs)
                  else jmp prog l (stack',heap,symbs)

runMSP prog (ALabel n:t) state = runMSP prog t state


jmp prog label state = runMSP prog prog' state
  where (Just npc) = elemIndex (ALabel label) prog
        prog'      = drop (npc+1) prog


npc :: Int -> Int -> Integer
npc l1 l2 =  toInteger' (l1 - l2)

toInteger' :: Int -> Integer
toInteger' i = read (show i)

toInt' :: Integer -> Int
toInt' i = read (show i)
```