Processamento de Linguagens e Compiladores

LMCC, Universidade do Minho

Ano lectivo 2006/2007

João Saraiva

Ficha Teórico-Prática Nº12

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 Geração de MSP em Haskell

Nesta ficha teórico-prática definimos um módulo para efectuar a geração de código. Neste módulo definem-se funções genéricas para a geração de código em Msp. Estas funções modelam em Msp estruturas condicionais, repetitivas, etc.

Solução

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


module GenCodeMsp where

import Data.Char
import Data.List
-- import PP_PLC                              -- ficha 10
import Msp                                 -- ficha 11
```

Antes de paresentarmos os execrcícios sobre geração de código vamos analisar uma linguagem muito simples que vai ser usada ao longo da ficha.

1.1 *Considere o seguinte tipo de dados algébrico que define a sintaxe abstracta de uma linguagem de programação muito simples chamada Simple Language (SL).*

Solução

```
type Sl   = [Inst]

data Inst = Atrib         String Expr
          | SeEntaoSenao  Expr [Inst] [Inst]
          | Enq           Expr  [Inst]
          | RepetirAte    [Inst] Expr

data Expr = SomaExp      Expr Expr
          | MultExp      Expr Expr
          | MQExp        Expr Expr
          | ConstInteira Integer
          | Variavel     String
```

Responda às seguintes perguntas:

1. *Construa uma frase em sintaxe abstracta que pertence a esta linguagem e que representa uma unica instrução* `aux = 10 + 15 * aux`.

2. *Utilizando os combinadores de Pretty Printing, da ficha nº10, defina uma função que produz SL em notação a la Pascal.*

Solução

```
exp1 = SomaExp (ConstInteira 10) (MultExp (ConstInteira 15) (Variavel "aux"))
```

---

2 Geração de Código para Expressões

2.1 *Escreva a função Haskell* `genCodeExp` *que dada uma expressão aritmética gera o código correspondente em Msp.*

Solução

```
genCodeExp :: Expr -> [Instr]

genCodeExp (SomaExp e1 e2) = e1_msp ++ e2_msp ++ [Add]
  where e1_msp = genCodeExp e1
        e2_msp = genCodeExp e2

genCodeExp (MultExp e1 e2) = e1_msp ++ e2_msp ++ [Mul]
  where e1_msp = genCodeExp e1
        e2_msp = genCodeExp e2

genCodeExp (MQExp e1 e2) = e1_msp ++ e2_msp ++ [Gt]
  where e1_msp = genCodeExp e1
        e2_msp = genCodeExp e2

genCodeExp (ConstInteira i) = [Pushi i ]

genCodeExp (Variavel    s)  = [ Pusha s , Load ]
```

---

3 Atribuição

Solução

```
-- a = exp1

atrib1 = Atrib "a" exp1
```

3.1 *Escreva a função Haskell* ***genCodeAssign*** *que dado o nome da variavél (lado esquerdo da atribuição) e o codigo da expressão do lado direito, gera código Msp para uma atribuição.*

Solução

```
genCodeAssign :: String -> [Instr] -> [Instr]
genCodeAssign s msp_e = [ Pusha s ] ++ msp_e ++ [Store]
```

---

4 Estruturas Condicionais

Solução

```
-- a = 10 + 15 * aux
-- if x > 5 then b = 5 else b = 6

bc = [ atrib1
     , SeEntaoSenao (MQExp (Variavel "x")
                           (ConstInteira 5))
              [Atrib "b" (ConstInteira 5)]
              [Atrib "b" (ConstInteira 6)]
     ]
```

4.1 *Considere a seguinte regra para modelar estruturas condicionais em MSP:*

```
se condiçao ->  { stat 1

   senão    ->  { stat 2
fse
```

Transforma-se em:

```
        ...
        <Teste condiçao>
        JMPF senao
        <stat 1>
        JMP fse
senao :
        <stat 2>
fse   :
        ...
```

Escreva a função Haskell ***genSeEntaoSenao*** *que dado o código Msp da condição os dos dois statements, modela o* ***If Then Else****.*

Solução

```
genSeEntaoSenao :: [Instr] -> [Instr] -> [Instr] -> [Instr]
genSeEntaoSenao c s1 s2 = c                 ++
                          [ Jumpf "senao" ] ++
                          s1                ++
                          [ Jump "fse" ]    ++
                          [ ALabel "senao"] ++
                          s2                ++
                          [ ALabel "fse"]
```

---

5 Estruturas de Controlo Repititivas

5.1 *Escreva um programa em sintaxe abstracta de SL que contenha uma instrução* ***enq***

Solução

5.2 *Considere a seguinte regra para modelar estruturas prepetitivas enquanto em MSP:*

```
enq  condição
  ->  { stat 1
        ...
        stat n
      }
fenq
```

```
        ...
enq:    <Teste condiçao>
        JMPF fenq
        <stat 1>
        ...
        <stat n>
        JMP enq
fenq:
        ...
```

Escreva a função Haskell ***genEnquato*** *que dado o código Msp da condição e dos statements, modela um While.*

Solução

5.3 *Escreva a função Haskell* `genRepetirAte` *que modela a estrutura repetitiva do .. while usual em linguagem de programação (por exemplo, Pascal).*

Solução

---

6 Geração de Código para SL

6.1 *Considere a linguagem SL. Utilizando as funções de geração de código desenvolvidas nesta ficha, escreva a função* `sl2msp` *que compila SL para MSP:*

Solução

```
-
genStat :: Stat -> [Instr]

genStat (Assign s e) = let e_msp = genCodeExp e
                           in Pusha s : e_msp ++ [Store]

genStat (If_t_e c s1 s2) = genIfThenElse c s1 s2

genStats  = concat . map genStat

-
```

6.2 *Como facilmente se constata utilizando um programa SL que inclua duas estruturas condicinais, as funções de geração de código produzem labels iguais para instruções diferentes. Altere as funções de geração de código de modo produzirem labels únicas e resolverem este problema. Considere que as funções recebem um valor inteiro para distinguir as etiquetas.*