

# Processamento de Linguagens e Compiladores

LMCC, Universidade do Minho

Ano lectivo 2005/2006

João Saraiva

Ficha Teórico-Prática N°8

Este texto está escrito em **literate Haskell**. Isto é, pode ser interpretado como um documento  $\text{\LaTeX}$  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 Propriedades de Gramáticas Independentes do Contexto

Nesta ficha teórico-prática apresentam-se funções que definem propriedades de símbolos não-terminais, produções e das próprias gramáticas independentes do contexto. Estas funções são expressas em Haskell. Para tal define-se o módulo *CfgProp*. Nesta ficha teórico-prática apresentam-se funções que definem propriedades de símbolos não-terminais, produções e das próprias gramáticas independentes do contexto. Estas funções são expressas em Haskell. Para tal define-se o módulo *CfgProp*.

### Solução

---

```
--  
-- Módulo de Propriedades de Gramáticas Independentes do Contexto  
--  
-- Processamento de Linguagens e Compiladores  
-- 2005/2006  
--  
module CfgProp where  
import List  
import Cfg
```

---

## 1.1 Gramáticas Ambíguas

Uma gramática  $G = (T, N, S, P)$  diz-se ambígua se existirem duas sequências de derivação diferentes para uma mesma frase  $\alpha \in T^*$ . A ambiguidade surge na escolha da produção a usar durante o processo de re-escrita de um símbolo não terminal. Isto é, num dado passo o processo de re-escrita de um símbolo não terminal existe mais do que uma produção possível que conduz à aceitação da frase. Logo existe ambiguidade em saber qual das diferentes alternativas (caminhos) seguir.

1.1 Considere a seguinte gramática independente de contexto  $G_{exp} = (T, N, S, P)$ , com:

$$\begin{aligned} T &= \{+, -, *, /, i, (, )\}, \\ N &= \{E\}, \\ S &= E, \\ P &= \left\{ \begin{array}{l} \text{add: } E \rightarrow E + E \\ \text{, } \text{sub: } \quad \quad \quad | E - E \\ \text{, } \text{plus: } \quad \quad \quad | E * E \\ \text{, } \text{div: } \quad \quad \quad | E / E \\ \text{, } \text{prio: } \quad \quad \quad | ( E ) \\ \text{, } \text{numb: } \quad \quad \quad | i \end{array} \right\} \end{aligned}$$

em que  $i$  é um símbolo pseudo-terminal que representa a classe dos números inteiros. Responda às seguintes perguntas.

1. Prove que a gramática  $G_{exp}$  é ambígua. Utilize para tal uma sequência de derivação LR.
2. Prove que a gramática  $G_{exp}$  é ambígua. Utilize para tal uma sequência de derivação LL.

1.2 Considere a gramática independente de contexto  $g_4$  escrita em Haskell e apresentada a seguir. Responda às seguintes perguntas.

1. Prove que "xzwzwy"  $\in \mathcal{L}(g_4)$
2. A gramática  $g_4$  é ambígua?

### Solução

---


$$\begin{aligned} g_4 &= \text{Cfg} \left[ \begin{array}{l} 'x', 'y', 'z', 'w' \\ 'X', 'Y', 'Z', 'W' \\ 'X' \\ [ ("p0", 'X' \mapsto "xY") \\ \text{, } ("p1", 'Y' \mapsto "y") \\ \text{, } ("p1", 'Y' \mapsto "Zy") \\ \text{, } ("p2", 'Z' \mapsto "zW") \\ \text{, } ("p3", 'W' \mapsto "wZ") \\ \text{, } ("p4", 'W' \mapsto "") \end{array} \right] \end{aligned}$$


---

## 1.2 Gramáticas Não Ambíguas

Uma gramática independente do contexto diz-se *não ambígua* se para toda a frase da linguagem definida pela gramática existir uma única sequência de derivação para essa frase. Neste caso não existe qualquer ambiguidade na escolha do caminho a seguir. Neste secção apresentamos a definição formal de algumas funções sobre gramáticas que nos ajudarão posteriormente a provar que uma gramática é não ambígua.

## 1.3 Funções $First$ e $First_N$

A função  $First$  calcula o conjunto de símbolos que iniciam derivações a partir de uma sequência de símbolos terminais e não terminais. A função  $First$  de uma sequência de símbolos define-se do seguinte modo:

$$First : (N \cup T)^* \rightarrow 2^T$$

com  $First(\alpha)$  definido por, para  $a \in T$  e  $X \in N$ :

1. Se  $\alpha = a\alpha'$  então  $First(\alpha) = \{a\}$ ;
2. Se  $\alpha = A\alpha' \wedge A \rightarrow \beta_1 \mid \dots \mid \beta_n \wedge \forall_i \beta_i \not\Rightarrow^* \epsilon$  então

$$First(\alpha) = \bigcup_{1 \leq i \leq n} First(\beta_i)$$

3. Se  $\alpha = A\alpha' \wedge A \rightarrow \beta_1 \mid \dots \mid \beta_n \wedge \exists_i \beta_i \Rightarrow^* \epsilon$  então

$$First(\alpha) = First(\alpha') \cup \bigcup_{1 \leq i \leq n} First(\beta_i)$$

A função  $First_N$  determina o conjunto de símbolos que iniciam derivações a partir de um símbolo não terminal. A função  $First_N$  de um símbolo não terminal define-se como:

$$First_N : N \rightarrow 2^T$$

$$First_N(X) : \bigcup_{(X \rightarrow \alpha_i) \in P} First(\alpha_i)$$

**1.3** Considere de novo a gramática  $g_{exp}$ . Calcule os seguintes conjuntos:

1.  $First(rhs(add))$
2.  $First_N(E)$

**1.4** Considere de novo a gramática  $g_4$ . Calcule os conjuntos  $First_N$  de todos os símbolos não terminais.

A função *First* escreve-se em Haskell seguindo directamente a sua definição formal. A função Haskell *first* tem tipo:

### Solução

---

```

first :: Eq sy
        => Cfg sy -- grammar
        -> [sy]  -- sequence of symbols
        -> [sy]  -- First set

```

Na definição da função **first** temos de ter o cuidado de não voltar a considerar um símbolo não terminal que já o tenha sido antes. Para tal usamos uma função auxiliar **first'** que tem um parametro adicional onde são acumulados os símbolos já considerados.

### Solução

---

```

first g sy = nub $ first' g [] sy
first' :: Eq sy
        => Cfg sy -- grammar
        -> [sy]  -- accumulator: nonterminals visited so far
        -> [sy]  -- sequence of symbols
        -> [sy]  -- First set
first' g - [] = []
first' g v (h : t) | h 'is_terminal' g = [h]
                   | h ∈ v              = first' g v t
                   | nullable_nt g h    = (first' g (h : v) t) 'union' first_rhs g v h
                   | otherwise          = first_rhs g v h
where first_rhs g v nt = concat $ map (first' g (nt : v)) (rhs_nt g nt)

```

A função *first<sub>N</sub>* define-se como um *map* da função anterior pelos lados direitos das produções do símbolo dado.

### Solução

---

```

firstN :: Eq sy
        => Cfg sy -- grammar
        -> sy     -- nonterminal
        -> [sy]  -- First set
firstN g nt = nub $ concat $ map (first g) (rhs_nt g nt)

```

1.5 Utilize as definições Haskell para confirmar os resultados do exercício anterior.

## 2 Função Follow

A função *Follow* de um símbolo não terminal calcula o conjunto de símbolos terminais que se podem seguir a uma derivação a partir do símbolo não terminal. A função *Follow* de um símbolo não terminal define-se do seguinte modo:

$$\begin{aligned}
 \text{Follow} &: N \rightarrow 2^T \\
 \text{Follow}(X) &: \bigcup_{(Y \rightarrow \alpha X \beta) \in P} \text{First}(\beta) \cup \begin{cases} \emptyset & \text{se } \beta \not\Rightarrow^* \epsilon \\ \text{Follow}(Y) & \text{se } \beta \Rightarrow^* \epsilon \end{cases}
 \end{aligned}$$

**2.1** Considere de novo as gramática  $g_{exp}$  e  $g_4$ . Calcule o conjunto de símbolos Follow para os símbolos não terminais destas gramáticas.

**2.2** No cálculo da função Follow de um símbolo não terminal é necessário identificar primeiro o conjunto de produções onde o símbolo não terminal ocorre no lado direito e as sequências de símbolos que se segue a cada ocorrência. Escreva em Haskell as seguintes funções sobre o tipo Cfg:

1. Função `rhs_with_nt` que dado uma gramática e um símbolo não terminal, calcula o conjunto de produções (sem nome) em que esse símbolo ocorre no lado direito.
2. Função `suffices` que dado um símbolo não terminal e uma produção devolve a lista de sequências de símbolos que seguem as ocorrências do não terminal no lado direito da produção dada. Note que um símbolo não terminal pode ocorrer mais do que uma vez no lado direito de um produção. Associe a cada um dos elementos desta lista o lado esquerdo da produção onde o sufixo ocorreu. Um exemplo de execução desta função apresenta-se a seguir.

```
*CfgProp> suffices 'B' ('A' |-> ['B', 'C', 'B'])
[('A', "CB"), ('A', "")]
```

3. Função `rhs_suffices` que dado uma gramática e um símbolo não terminal  $X$  devolve as sequências de símbolos que seguem as ocorrências de  $X$  nos lados direitos das produções de  $P$ . A estes sufixos são também associados ao símbolo não terminal do lado esquerdo de  $P$ . Por exemplo, se executarmos esta função sobre a gramática  $g_3$  (definida na ficha nº 10) e o não terminal  $B$  temos:

```
*CfgProp> rhs_suffices g_3 'B'
[('A', "CB"), ('A', ""), ('B', "b"), ('C', "")]
```

## Solução

---

```
rhs_with_nt :: Eq sy => Cfg sy -> sy -> [[sy]]
rhs_with_nt g nt = filter (elem nt o tail) (map snd (prods g))
suffices :: Eq sy => sy -> [sy] -> [(sy, [sy])]
suffices nt (lhs : rhs) = rhs_suffices_nt rhs nt
  where rhs_suffices_nt [] _ = []
        rhs_suffices_nt (h : t) nt | h == nt = (lhs, t) : rhs_suffices_nt t nt
        | otherwise = rhs_suffices_nt t nt
rhs_suffices :: (Eq sy) => Cfg sy -> sy -> [(sy, [sy])]
rhs_suffices g nt = concat $ map (suffices nt) (rhs_with_nt g nt)
```

---

A função *Follow* escreve-se em Haskell seguindo de novo a sua definição formal. Porém, e tal como na definição da função **first** e **nullable**, temos de ter o cuidado de manter uma

lista de símbolos não terminais já considerados de modo à definição da função terminar. O tipo da função **follow** é:

**Solução**

---

```
follow :: Eq sy
        => Cfg sy -- grammar
        -> sy    -- nonterminal
        -> [sy]  -- Follow set
```

---

A função escreve-se em Haskell da seguinte forma:

**Solução**

---

```
follow g nt = nub $ follow' g [] nt
follow' :: Eq sy => Cfg sy -> [sy] -> sy -> [sy]
follow' g v nt | nt ∈ v    = []
                | otherwise = concat $ map (follow'' g (nt : v)) (rhs_suffices g nt)
follow'' :: (Eq sy) => Cfg sy -> [sy] -> (sy, [sy]) -> [sy]
follow'' g v (l, r) | nullable g [] r = first g r 'union' follow' g v l
                    | otherwise      = first g r
```

---

**2.3** Utilize a definição Haskell para verificar as soluções do exercício ??.

### 3 Função Lookahead

Como foi referido anteriormente, para uma gramática ser não ambígua não pode existir ambiguidade na escolha da produção a utilizar em cada passo da derivação de uma frase. Um modo de garantir que uma gramática não possui ambiguidade na escolha da produção a utilizar, consiste no seguinte: sempre que se vai expandir um símbolo não terminal  $A$ , tenta-se *olhar para a frente*<sup>1</sup>, de modo a determinar quais os símbolos terminais que iniciam as várias alternativas/produções de  $A$ . Caso estes inícios sejam disjuntos, então não existe ambiguidade na escolha do lado direito, pelo qual  $A$  vai ser expandido.

Formalmente, a função que permite "olhar para a frente", designada *Lookahead*, define-se do seguinte modo:

$$Lookahead : P \rightarrow 2^T$$

$$Lookahead(A \rightarrow \alpha) : First(\alpha) \cup \begin{cases} \emptyset & \text{se } \alpha \not\Rightarrow^* \epsilon \\ Follow(A) & \text{se } \alpha \Rightarrow^* \epsilon \end{cases}$$

E a sua escrita em Haskell segue directamente a definição formal.

**Solução**

---

```
lookahead :: Eq sy
           => Cfg sy -- grammar
           -> [sy]  -- production
```

---

<sup>1</sup>Na terminologia inglesa designa-se por *Lookahead*.

```

→ [sy]      -- Lookahead set
lookahead g p | nullable g [] (rhs p) = nub $ first g (rhs p) ++ follow g (lhs p)
               | otherwise                = nub $ first g (rhs p)

```

---

**3.1** Considere de novo as gramáticas  $g_{exp}$  e  $g_A$ . Calcule o conjunto de Lookaheads das produções de ambas as gramáticas. Utilize a definição da função em Haskell para confirmar estes resultados.

**3.2** Escreva em Haskell as seguintes funções sobre o tipo de dados Cfg:

1. Função `lookaheadsnt` que dado uma gramática e um símbolo não terminal devolve o conjunto de lookaheads das produções que tem o símbolo não terminal como lado esquerdo.
2. Função `all_lookaheads` que dado uma gramática devolve os lookaheads de todas as produções.

**Solução**

---

```

all_lookaheads g = map (lookahead g) (map snd (prods g))
lookaheads_nt g nt = map (lookahead g) (prods_nt g nt)

```

---

## 4 Condição LL(1)

Para garantir que não há ambiguidade na escolha da produção, definimos a condição  $LL(1)$ . Uma gramática  $G = (T, N, S, P)$  satisfaz a *condição LL(1)* se

$$\forall_{A \rightarrow \alpha_1, A \rightarrow \alpha_2 \in P} : \text{Lookahead}(A \rightarrow \alpha_1) \cap \text{Lookahead}(A \rightarrow \alpha_2) = \emptyset$$

Que se define facilmente em Haskell do seguinte modo:

**Solução**

---

```

ll_1_nt g nt = and (map (≡ []) (intersects xs))
  where xs = lookaheads_nt g nt
intersects [] = []
intersects (h : t) = (map (intersect h) t) ++ (intersects t)
ll_1 :: Eq sy ⇒ Cfg sy → Bool
ll_1 g = and $ map (ll_1_nt g) (nonterminals g)

```

---

**4.1** Utilizando a definição formal da condição  $LL(1)$  e a sua representação em Haskell, responda às seguintes perguntas:

1. Prove, utilizando a definição formal, que a gramática  $g_{exp}$  não é  $LL(1)$ .
2. Prove, utilizando a definição formal, que a gramática  $g_A$  é  $LL(1)$ .

3. Prove que a gramática  $g_A$  é não ambígua.
4. Prove que a gramática  $g_{exp}$  é ambígua.
5. Utilize a definição em Haskell para confirmar estes resultados.