Conjuntos first e follow

Construção de compiladores I

Objetivos

Objetivos

- Apresentar os conjuntos first e follow e seu uso para definir gramáticas LL(1)
- Apresentar uma implementação de first e follow em Haskell.

Gramáticas LL(1)

Gramáticas LL(1)

- Classe de gramáticas que admitem analisadores sem backtracking
- LL(1) significa
  - *L*eft to right: entrada analisada da esquerda para direita.
  - *L*eft most derivation: construção de derivação mais a esquerda.
  - Uso de **1** token da entrada para decidir a derivação.

Gramáticas LL(1)

- Como determinar se uma gramática é LL(1)?
  - Vamos usar os conjuntos First e Follow.

First e Follow

First e Follow

- $first(\alpha)$: conjunto de terminais que iniciam sentenças derivadas a partir de α.
- $\alpha \in (V \cup \Sigma)^*$.

First e Follow

- $first(a) = \{a\}$, se $a \in \Sigma$.
- $\lambda \in first(A)$ se $A \to \lambda \in R$.

First e Follow

- Se $A \in V$ e
- $A \to Y_1\, Y_2 \ldots Y_k \in R$ e
- $a \in first(Y_i)$ e
- $\forall j. 1 \leq j \leq i-1. \lambda \in first(Y_j)$ então $a \in first(A)$.

First e Follow

- Aplique estas regras enquanto for possível.

First e Follow

- Calcular os conjuntos $first$ para:

$$
\begin{array}{lcl}
E & \to & TE' \\
E' & \to & +TE' \mid \lambda \\
T & \to & FT' \\
T' & \to & \mathbf{*}FT' \mid \lambda \\
F & \to & (E) \mid \mathbf{id}
\end{array}
$$

First e Follow

- $first(E) = first(T) = first(F)$
- $first(F) = \{(, \mathbf{id}\}$

First e Follow

- $first(E') = \{+, \lambda\}$
- $first(T') = \{\mathbf{*}, \lambda\}$

First e Follow

- $follow(A)$: conjunto de terminais que aparecem logo a direita de A em alguma derivação.
- $A \in V$.

First e Follow

- $\$ \in follow(P)$
- Se $A \to \alpha B \beta \in R$ então:

  $first(\beta) - \{\lambda\} \subseteq follow(B)$

First e Follow

- Se $A \to \alpha B$ ou $A \to \alpha B \beta$, em que $\lambda \in first(\beta)$ então:

  $follow(A) \subseteq follow(B)$

First e Follow

- Aplique estas regras enquanto possível.

First e Follow

- Calcular os conjuntos $follow$ para:

$$
\begin{array}{lcl}
E & \to & TE' \\
E' & \to & +TE' \mid \lambda \\
T & \to & FT' \\
T' & \to & \mathbf{*}FT' \mid \lambda \\
F & \to & (E) \mid \mathbf{id}
\end{array}
$$

First e Follow

- $\$ \in follow(E)$:
  - E é a variável inicial da gramática.

First e Follow

-) $\in follow(E)$:
  - Devido a produção $F \rightarrow (E)$
  - Regra: $A \rightarrow \alpha B \beta$ então $first(\beta) \subseteq follow(B)$.
    - Neste caso, $\beta =$)

First e Follow

- Logo, temos que $follow(E) = \{\$,)\}$.

Implementação em Haskell

Implementação em Haskell

- Representando terminais

```
data Terminal
  = T String
  | Dollar
  | Lambda
```

Implementação em Haskell

- Representando não terminais

```
data Nonterminal
  = NT String
```

Implementação em Haskell

- Representando símbolos

```
data Symbol
  = Var Nonterminal
  | Symb Terminal
```

Implementação em Haskell

- Representando produções

```
data Production
  = Prod {
      leftHand :: Nonterminal
    , rightHand :: [Symbol]
    }
```

Implementação em Haskell

- Representando gramáticas

```
data Grammar
  = Grammar {
      productions :: [Production]
    , start       :: Nonterminal
    }
```

Implementação em Haskell

- Representação de ponto fixo.

```
fixpoint :: Eq a => (a -> a) -> a -> a
fixpoint f x = let x' = f x
               in if x == x' then x
                  else fixpoint f x'
```

Implementação em Haskell

- Definição de tabela de first.

```
type First = Map Nonterminal [Terminal]

merge :: First -> Nonterminal -> [Terminal] -> First
merge m nt ts = Map.insertWith union nt ts m

firstSetFor :: Nonterminal -> First -> [Terminal]
firstSetFor nt m
  = case Map.lookup nt m of
      Nothing -> []
      Just ts -> ts
```

Implementação em Haskell

- Cálculo de first

```
first :: Grammar -> [(Nonterminal, [Terminal])]
first g = Map.toList m
  where
     m = fixpoint (stepFirst g) (first0 g)
```

Implementação em Haskell

- Cálculo de first0

```
first0 :: Grammar -> First
first0 g = Map.fromList $ map f (nonterminals g)
  where
    f nt = (nt , [])
```

Implementação em Haskell

- Iteração do conjunto first

```
stepFirst :: Grammar -> First -> First
stepFirst g current
  = step' (productions g) current
  where
    step' [] curr = curr
    step' (p:ps) curr
      = merge (step' ps curr)
              (leftHand p)
              (terminalsForRHS p)
```

Implementação em Haskell

- Definição de `terminalsForRHS`

```
terminalsForRHS :: Production -> [Terminal]
terminalsForRHS (Prod _ []) = [Lambda]
terminalsForRHS (Prod _ [Symb Lambda]) = [Lambda]
terminalsForRHS (Prod x (yj : ys))
  = case  yj of
      Symb terminal -> [terminal]
```

```
      Var nonterminal ->
         if Lambda `elem` first_iminus1 then
            terminalsForYj `union` terminalsForRHS (Prod x ys)
         else
            terminalsForYj
      where
         first_iminus1 = firstSetFor nonterminal current
         terminalsForYj = filter (/= Lambda) first_iminus1
```

Implementação em Haskell

- Extensão de first para palavras

```
firstForWord :: [Symbol] -> First -> [Terminal]
firstForWord [(Var nt)] ft = firstSetFor nt ft
firstForWord [(Symb t)] _ = [t]
firstForWord ((Var nt) : ss) ft =
  if Lambda `elem` firstSetFor nt ft then
        firstMinusLambda `union` firstForWord ss ft
  else firstMinusLambda
       where firstMinusLambda = [x | x <- firstSetFor nt ft, x /= Lambda]
firstForWord _ _ = []
```

Implementação em Haskell

- Implementação de follow

```
follow :: Grammar -> [(Nonterminal, [Terminal])]
follow g
  = Map.toList $ fixpoint (stepFollow g m) (follow0 g)
  where
    m = Map.fromList $ first g
```

Implementação em Haskell

- Inicialização de follow

```
follow0 :: Grammar -> Follow
follow0 g = Map.fromList $ map f (nonterminals g)
  where
    f nt = if nt == start g then (nt, [Dollar]) else (nt, [])
```

Implementação em Haskell

- Iteração do conjunto follow

```
stepFollow :: Grammar -> First -> Follow -> Follow
stepFollow g firstG current
  = step' (productions g) firstG current
    where
      step' [] _ curr = curr
      step' (p : ps) fG curr
        = mergeTerminals p (step' ps fG curr)
       where
         ...
```

Implementação em Haskell

- Iteração do conjunto follow

```
mergeTerminals (Prod _ []) = undefined
mergeTerminals (Prod l (s : ss)) = merge' l [] s ss
```

Implementação em Haskell

- Iteração do conjunto follow

```
merge' a _ (Var b) [] fMinus1
  = merge fMinus1 b (followSetFor a fMinus1)
merge' _ _  (Symb _) [] fMinus1 = fMinus1
merge' a w1 (Symb t) (w2 : w2s) fMinus1
 = merge' a (w1 ++ [Symb t]) w2 w2s fMinus1
merge' a w1 nt@(Var b) w2@(w21 : w2s) fMinus1
  = merge (merge' a (w1 ++ [nt]) w21 w2s fMinus1) b new
    where
       firstW2 = [x | x <- firstForWord w2 fG, x /= Lambda]
       new = if Lambda `elem` firstForWord w2 fG then
                 followSetFor a fMinus1 `union` firstW2
             else firstForWord w2 fG
```

Concluindo

Concluindo

- Nesta aula, apresentamos os conjuntos first e follow.

- Apresentamos uma implementação simples em Haskell de funções para calcular estes conjuntos.

Concluindo

- Próxima aula: Análise sintática LL(1)

Exercícios

Exercícios

- Utilizando a representação de gramáticas utilizada para obter os conjuntos first e follow, implemente uma função para obter o conjunto de não terminais anuláveis de uma gramática. Dizemos que um não terminal é anulável se ele deriva λ em um ou mais passos de derivação.