Derivadas e Análise Léxica

Construção de compiladores I

Objetivos

Objetivos

- Apresentar outra técnica para obtenção de AFDs a partir de ERs: derivadas.
- Apresentar o gerador de analisadores léxicos para Haskell: Alex.

Derivadas de ERs

Derivadas de ERs

- Definição de derivada de uma linguagem:

$$\partial_a(L) \quad = \quad \{y \in \Sigma^* \mid ay \in L\}$$

Derivadas de ERs

- Exemplo:

$$\partial_0(\{10, \lambda, 0, 01\}) \quad = \quad \{\lambda, 1\}$$

Derivadas de ERs

- A operação de derivada pode ser definida sobre ERs.
  - Definição da derivada é apenas uma função recursiva.

Derivadas de ERs

- Operação auxiliar: anulabilidade.

$$\begin{array}{lcl}
\nu(\emptyset) & = & \bot \\
\nu(\lambda) & = & \top \\
\nu(a) & = & \bot \\
\nu(e_1 + e_2) & = & \nu(e_1) \vee \nu(e_2) \\
\nu(e_1\, e_2) & = & \nu(e_1) \wedge \nu(e_2) \\
\nu(e_1^*) & = & \top
\end{array}$$

Derivadas de ERs

- Definição da derivada

$$\begin{array}{lcl}
\partial_a(\emptyset) & = & \emptyset \\
\partial_a(\lambda) & = & \emptyset \\
\partial_a(a) & = & \lambda \\
\partial_a(b) & = & \emptyset \text{ se } a \neq b
\end{array}$$

Derivadas de ERs

- Definição de derivada (continuação)

$$\begin{array}{lcl}
\partial_a(e_1 + e_2) & = & \partial_a(e_1) + \partial_a(e_2) \\
\partial_a(e_1\, e_2) & = & \partial_a(e_1)\, e_2 + \partial_a(e_2), \text{ se } \nu(e_1) = \top \\
\partial_a(e_1\, e_2) & = & \partial_a(e_1)\, e_2 \text{ se } \nu(e_1) = \bot \\
\partial_a(e_1^*) & = & \partial_a(e_1)\, e_1^*
\end{array}$$

Derivadas de ERs

- Número de derivadas é finito sobre uma ER e é finito.
  - Se considerarmos a equivalência de ER

Derivadas de ERs

- Equivalência de ERs

$$\begin{array}{cc}
e + (e' + e'') \approx (e + e') + e'' & e + e \approx e \\
e + e' \approx e' + e & (ee')e'' \approx e(e'e'') \\
\emptyset e \approx \emptyset & e\emptyset \approx \emptyset \\
e\lambda \approx e & \lambda e \approx e \\
e + \emptyset \approx e & \emptyset + e \approx e
\end{array}$$

Derivadas de ERs

- Exemplo: Cálculo da derivada de $(01)^*$.

$$\begin{array}{ll} \partial_0((01)^*) & = \\ \partial_0(01)(01)^* & = \\ \partial_0(0)1(01)^* & = \\ \lambda 1(01)^* & = \\ 1(01)^* & \end{array}$$

Derivadas de ERs

- Processando strings usando derivadas.

$$\begin{array}{lcl} \widehat{\partial}(e, \lambda) & = & \nu(e) \\ \widehat{\partial}(e, ay) & = & \widehat{\partial}(\partial_a(e), y) \end{array}$$

Derivadas de ERs

- Além de processar palavras diretamente, podemos construir AFDs diretamente a partir de uma ER.

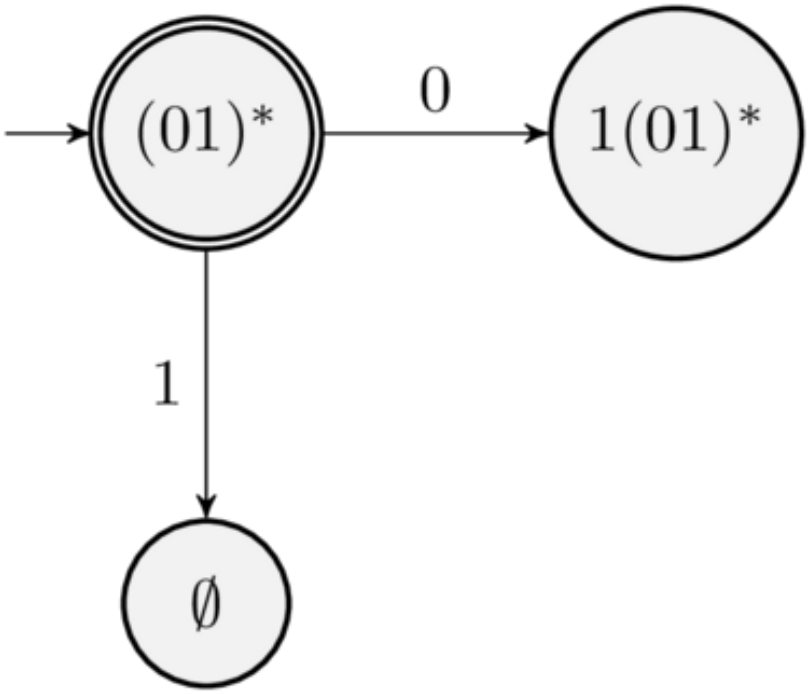
Derivadas de ERs

- O AFD correspondente a uma ER e é: $(E, \Sigma, \delta, e, F)$.
  - E: conjunto de derivadas
  - $\delta(e, a) = e'$ se $\partial_a(e) = e'$.
  - $F = \{e \,|\, \nu(e) = \top\}$.

Derivadas de ERs

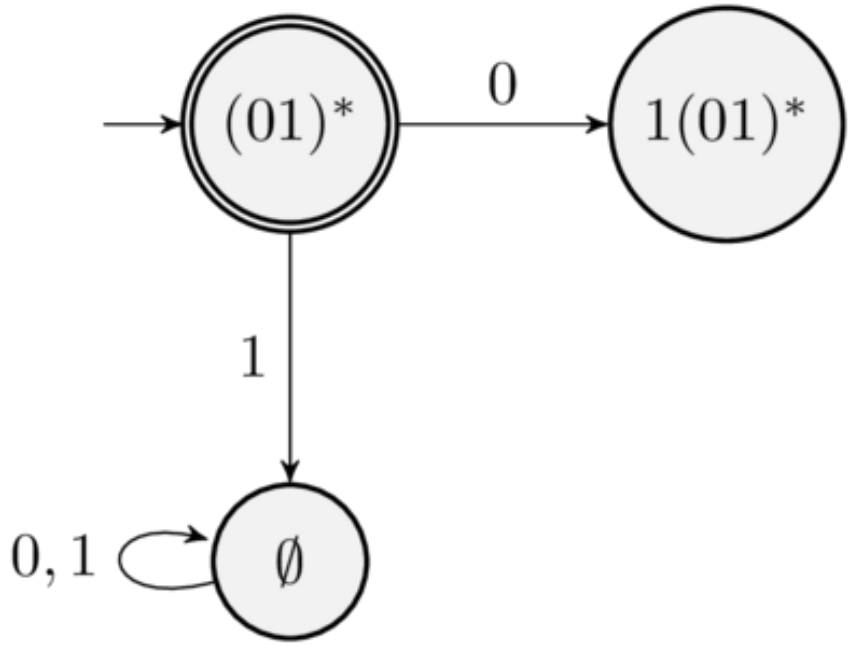
- Exemplo: Construir um AFD para $(01)^*$.
  - Já calculamos: $\partial_0((01)^*) = 1(01)^*$.
  - $\partial_1((01)^*) = \emptyset$.

Derivadas de ERs



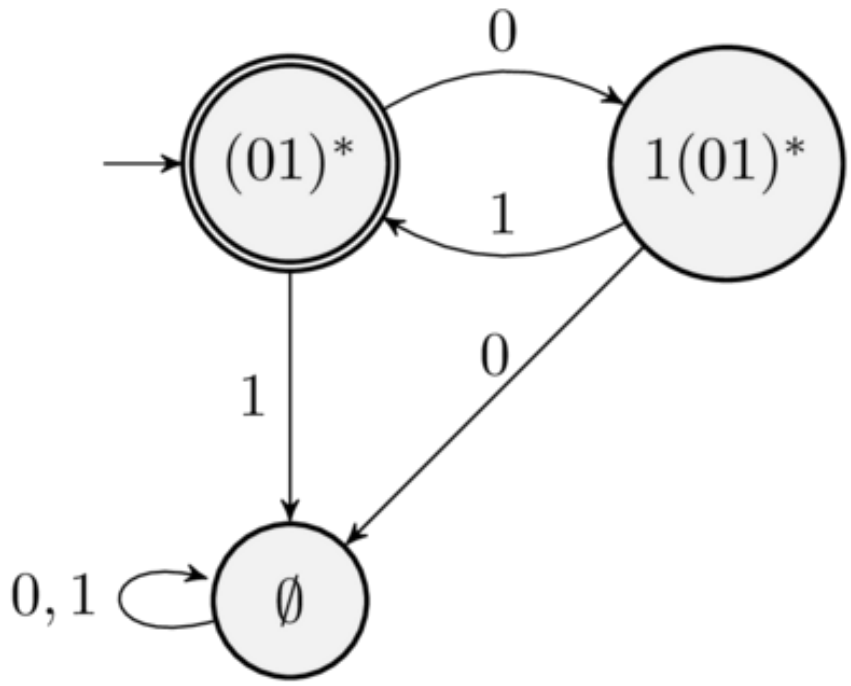
Derivadas de ERs

- Repetindo o processo para outras ERs.
  - $\partial_0(\emptyset) = \partial_1(\emptyset) = \emptyset$.



Derivadas de ERs

- Repetindo o processo para outras ERs.
  - $\partial_0(1(01)^*) = \emptyset$.
  - $\partial_1(1(01)^*) = (01)^*$.



Derivadas em Haskell

Derivadas em Haskell

- Derivadas são de implementação imediata em Haskell.

Derivadas em Haskell

- Teste de anulabilidade.

```
nullable :: Regex -> Bool
nullable Empty = False
nullable Lambda = True
nullable (Chr _) = False
nullable (e1 :+: e2)
  = nullable e1 || nullable e2
nullable (e1 :@: e2)
  = nullable e1 && nullable e2
nullable (Star e1) = True
```

Derivadas em Haskell

- Antes de apresentar a função de cálculo de derivadas, é interessante introduzir algumas funções para realizar simplificações.

Derivadas em Haskell

- Simplificando união.

```
(.+.) :: Regex -> Regex -> Regex
Empty .+. e' = e'
e .+. Empty  = e
e .+. e'     = e :+: e'
```

Derivadas em Haskell

- Simplificando concatenação

```
(.@.) :: Regex -> Regex -> Regex
Empty .@. _ = Empty
_ .@. Empty = Empty
Lambda .@. e' = e'
e .@. Lambda = e
e .@. e' = e :@: e'
```

Derivadas em Haskell

- Simplificando o fecho de Kleene.

```
star :: Regex -> Regex
star Empty = Lambda
star (Star e) = e
star e = Star e
```

Derivadas em Haskell

- Definição da derivada

```
deriv :: Char -> Regex -> Regex
deriv _ Empty  = Empty
deriv _ Lambda = Empty
deriv a (Chr b)
```

```
  | a == b = Lambda
  | otherwise = Empty
deriv a (e1 :+: e2)
  = deriv a e1 .+. deriv a e2
deriv a (e1 :@: e2)
  | nullable e1 = deriv a e1 .@. e2 .+. deriv a e2
  | otherwise   = deriv a e1 .@. e2
deriv a (Star e1)
  = deriv a e1 .@. (star e1)
```

Derivadas em Haskell

- Aceitando uma string.

```
match :: Regex -> String -> Bool
match e [] = nullable e
match e (c : cs) = match (deriv c e) cs
```

Uso do gerador Alex

Uso do gerador Alex

- A construção de um analisador léxico é uma tarefa automatizável.

Uso do gerador Alex

- Veremos como usar a ferramenta Alex para construir um analisador a partir de uma especificação.
  - Especificações Alex são apenas expressões regulares.

Uso do gerador Alex

- Para exemplificar o uso do Alex, vamos considerar uma linguagem de expressões.

$$e \quad \rightarrow \quad n \,|\, v \,|\, e + e \,|\, e * e \,|\, (e)$$

Uso do gerador Alex

- A sintaxe da linguagem de expressões aritméticas é formada pelos seguintes tokens:

  - Números: n
  - Variáveis: v
  - Símbolos: $+$, $*$, $($ e $)$.

Uso do gerador Alex

- Especificações Alex são formadas por:
  - Trechos de código Haskell
  - Definição do *wrapper* e de macros.
  - Declaração dos tokens.

Uso do gerador Alex

- Trechos de código Haskell
  - Definir cabeçalho do módulo
  - Tipos e funções.

Uso do gerador Alex

- Cabeçalho do módulo

```
{
module Arith.Lexer (lexer, testLexer) where
}
```

Uso do gerador Alex

- Definição do tipo `Token`

```
data Token
  = TNumber Int
  | TVar String
  | TAdd
  | TMul
  | TLParen
  | TRParen
  deriving (Eq, Show)
```

Uso do gerador Alex

- Funções auxiliares
  - Função `alexScanTokens` gerada automaticamente.

```
lexer :: String -> [Token]
lexer = alexScanTokens

testLexer :: IO ()
testLexer
  = do
      s <- getLine
      let tokens = lexer s
      mapM_ print tokens
```

Uso do gerador Alex

- Macros para conjuntos de caracteres.

```
$digit = 0-9            -- digits
$alpha = [a-zA-Z]       -- alphabetic characters
```

Uso do gerador Alex

- Macros para expressões regulares

```
@identifier = $alpha[$alpha $digit]* -- identifiers
@number     = $digit+
```

Uso do gerador Alex

- Especificação de tokens.

```
tokens :-
  $white+                 ; -- removing whitespace
  "//".*                  ; -- removing line comments
  @number                 { \ s -> TNumber (read s) }
  @identifier             { \ s -> TVar s }
  \+                      { \ _ -> TAdd }
  \*                      { \ _ -> TMul }
  \(                      { \ _ -> TLParen }
  \)                      { \ _ -> TRParen }
```

Uso de gerador Alex

- Arquivo `Arith/Lexer.x` contém a especificação.
- Produzindo o analisador léxico.
  - Produz o arquivo `Arith/Lexer.hs`

```
alex Lexer.x
```

Uso do gerador Alex

- A partir da especificação, o Alex:
  - Produz um analisador baseado em AFD mínimo para as ERs definidas.

Uso do gerador Alex

- Permite a definição rápida de analisadores para linguagens reais.

Uso do gerador Alex

- Problemas
  - Não há verificação de tipos em ações do analisador.
  - Necessidade de dominar outra linguagem.

Concluindo

Concluindo

- Com isso, concluímos o conteúdo de análise léxica.
- Próxima aula: análise sintática.

Exercícios

Exercícios

- Construa um analisador léxico, utilizando o gerador Alex, para a linguagem IMP. Seu analisador deve ler um arquivo, fornecido como argumento de linha de comando, e imprimir todos os tokens encontrados.