Análise Léxica

Construção de compiladores I

Objetivos

Objetivos

- Apresentar a importância da etapa de análise léxica em um compilador.
- Apresentar a implementação de um analisador léxico ad-hoc para uma linguagem simples.

Objetivos

- Mostrar como a teoria de expressões regulares e autômatos pode ser utilizada para automatizar a criação de analisadores léxicos.

Objetivos

- Apresentar o gerador de analisadore léxicos, Alex

Análise léxica

Análise léxica

- Primeira etapa do front-end de um compilador.
- Simplificar a entrada para análise sintática.

Análise léxica

- Simplificações:
  - Remoção de espaços em branco.
  - Remoção de comentários.
- Resultado: lista de **tokens**.

Análise léxica

- Token
  - Componente indivisível da sintaxe de uma linguagem.

Análise léxica

- Exemplos de tokens:
  - identificadores
  - palavras reservadas
  - separadores
  - literais

Análise léxica

- Como implementar a análise léxica?

Análise léxica ad-hoc

- Percorra a string:
  - Se for um dígito, guarde-o para formar um número.
  - Se for um operador, gere o token.
  - Se for um parêntesis, gere o token.
  - Se for um espaço, tente gerar um número e descarte o espaço.

Análise léxica ad-hoc

- Como representar tokens?

```
data Token
  = Number Int
  | Add
  | Minus
  | LParen
  | RParen
  deriving (Eq, Show)
```

Análise léxica ad-hoc

- Configuração do analisador léxico
  - Lista de tokens encontrados.
  - String de dígitos consecutivos encontrados.

```
type LexerState = Maybe ([Token], String)
```

Análise léxica ad-hoc

- Como finalizar um número?
  - Encontrando um espaço ou operador, criamos um token com os dígitos.

```
updateState :: LexerState -> LexerState
updateState Nothing = Nothing
updateState ac@(Just (ts, ns))
  | all isDigit ns && not (null ns)
    = let v = read (reverse ns)
      in Just (Number v : ts, [])
  | otherwise =ac
```

Análise léxica ad-hoc

- Iterando sobre a string de entrada.

```
lexer' :: LexerState -> String -> LexerState
lexer' ac [] = updateState ac
lexer' Nothing _ = Nothing
lexer' ac@(Just (ts, ns)) (c:cs)
  | isSpace c = lexer' (updateState ac) cs
  | isDigit c = lexer' (Just (ts, c : ns)) cs
  | c == '(' = lexer' (Just (LParen : ts, ns)) cs
  | c == ')' = lexer' (Just (RParen : ts, ns)) cs
  | c == '+' = lexer' (Just (Add : ts, ns)) cs
  | c == '*' = lexer' (Just (Mult : ts, ns)) cs
  | otherwise = Nothing
```

Análise léxica ad-hoc

- Interface principal do analisador

```
lexer :: String -> Maybe [Token]
lexer s
  = case lexer' (Just ([], "")) s of
      Nothing -> Nothing
      Just (ts, _) -> Just (reverse ts)
```

Análise léxica ad-hoc

- Algoritmo simples para análise léxica de uma linguagem.
- Problema: difícil de extender.
  - Como incluir números de ponto flutuante?
  - Como incluir identificadores e palavras reservadas?

Análise léxica ad-hoc

- Para acomodar essas mudanças, precisamos de uma abordagem sistemática para a análise léxica.
- Para isso, utilizaremos a teoria de expressões regulares e autômatos finitos.

Expressões regulares

Expressões regulares

- Formalismo algébrico para descrição de linguagens.
- Amplamente utilizado para representação de padrões em texto.
- Análise léxica: dividir texto em subpadrões de interesse.

Expressões regulares

- Qual a relação entre ERs e análise léxica?
  - Usando ERs podemos **automatizar** a construção de analisadores léxicos.

Expressões regulares

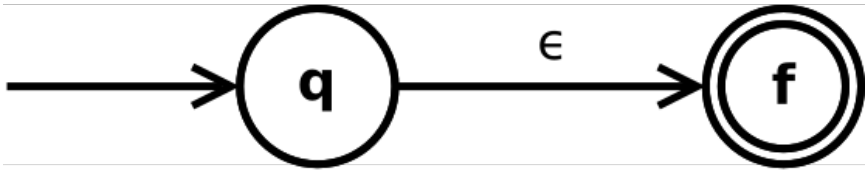
- Em essência, um analisador léxico é um AFD que produz uma lista de tokens.
- Em Teoria da computação, vimos que toda ER possui um AFD equivalente
  - Construção de Thompson / derivadas

Expressões regulares

- Construção de Thompson
  - Baseada em propriedades de fechamento de AFs.
  - Cria um AFN com transições lambda.

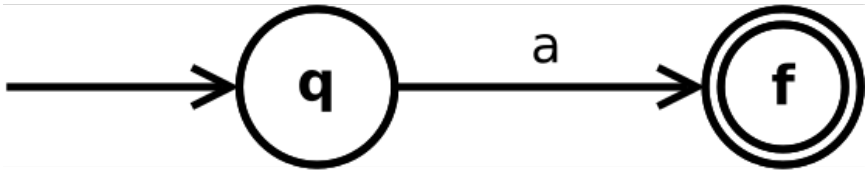
Expressões regulares

- Construção de Thompson para lambda.



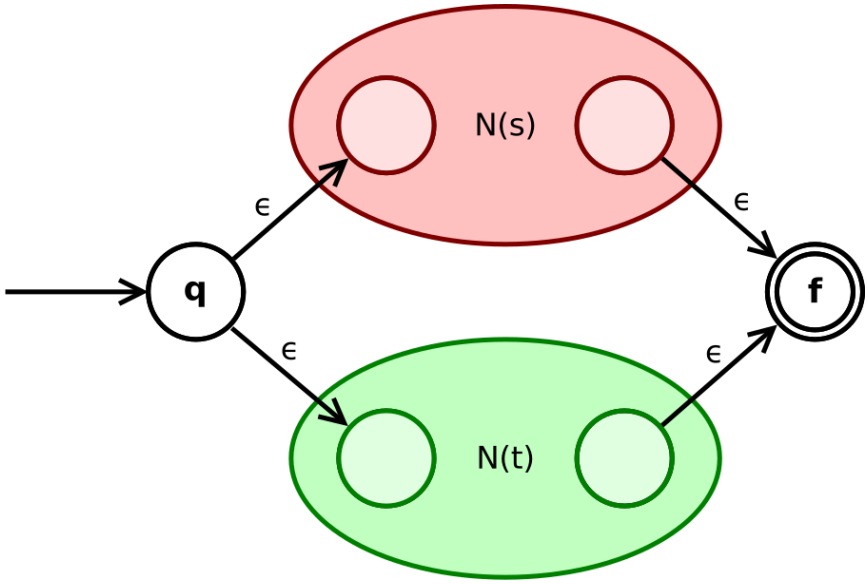
Expressões regulares

- Construção de Thompson para símbolo.



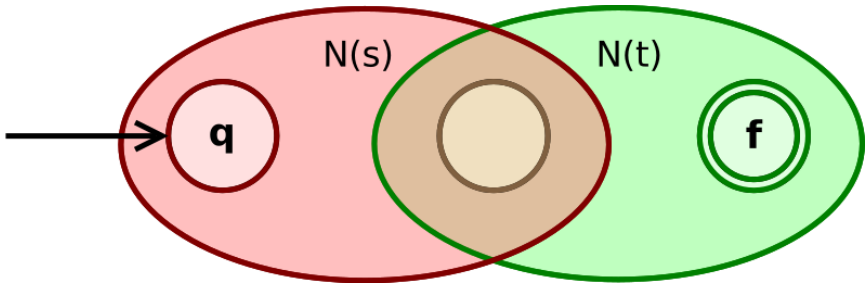
Expressões regulares

- Construção de Thompson para união.



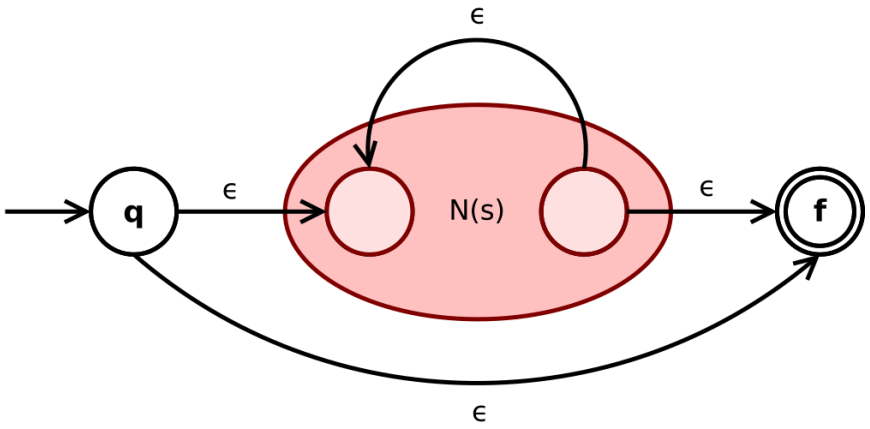
Expressões regulares

- Construção de Thompson para concatenação.



Expressões regulares

- Construção de Thompson para Kleene.



Expressões regulares

- Como representar AFD em código?
  - Normalmente, utilizamos uma matriz para representar a função de transição.

Expressões regulares

- Representando um AFD:

```
-- a: type for states
-- b: type for alphabet symbols

data DFA a b
  = DFA {
      start :: a
    , trans :: [((a,b), a)]
    , final :: [a]
    } deriving Show
```

Expressões regulares

- Processando palavras usando o AFD

```
delta :: (Eq a, Eq b) => DFA a b -> [b] -> Maybe a
delta m s = foldl step (Just (start m)) s
  where
```

```
    step (Just e) a
      = lookup (e,a) (trans m)
    step Nothing _ = Nothing
```

Expressões regulares

- Representando o AFD de números:

```
data State = S0 | S1 deriving (Eq, Show)

numberDFA :: DFA State Char
numberDFA
  = DFA {
      start = S0
    , trans = [((S0, c), S1) | c <- ['0'..'9']] ++
              [((S1, c), S1) | c <- ['0'..'9']]
    , final = [S1]
    }
```

Expressões regulares

- Como usar AFDs para produzir os tokens?
  - Crie o token usando o maior prefixo possível processado.

Expressões regulares

- Produzindo um token

```
extract :: DFA State Char -> String -> (String, String)
extract m s = go (start m) "" s
  where
    go _ token [] = (token, [])
    go e token (x : xs)
      | isSpace x = (token, x : xs)
      | otherwise = case lookup (e,x) (trans m) of
                      Just e' -> go e' (token ++ [x]) xs
                      Nothing -> (token, x : xs)
```

Expressões regulares

- Analisador léxico

```
dfaLexer :: DFA State Char -> String -> [Token]
dfaLexer m s = go s []
  where
    go [] ac = reverse ac
    go (x : xs) ac
      | isSpace x = go xs ac
      | otherwise =
        let (token, rest) = extract m (x : xs)
        in go rest (if null token then ac else Number (read token) : ac)
```

Expressões regulares

- Esse código simples funciona para apenas um AFD.
- A especificação de uma linguagem é formada por várias ERs.
  - Como combiná-las para produzir um AFD?

Expressões regulares

- Como combinar AFDs?
  - Propriedades de fechamento!
- Processo automatizável utilizando geradores de analisadores léxicos.

Analisadores léxicos

Analisadores léxicos

- Geradores de analisadores produzem a representação de AFDs mínimos a partir de uma especificação descrita como expressões regulares.
- Abordagens baseadas no teorema de Kleene / derivadas

Analisadores léxicos

- Para Haskell, podemos utilizar a ferramenta Alex.
- Produz o código Haskell correspondente ao analisador léxico a partir de uma especificação.

Analisadores léxicos

- Componentes de uma especificação Alex.
  - Código Haskell
  - Especificação de expressões regulares.
  - Definição de *wrapper*.

Analisadores léxicos

- Trechos de código Haskell
  - Definem funções utilizadas para criação de tokens
  - Definir o tipo do token
  - Definição de módulo e importações.

Analisadores léxicos

- Expressões regulares.

```
$digit = 0-9
@number = $digit+

tokens :-
      -- whitespace and comments
      <0> $white+       ;
      <0> "--" .*       ;
      -- other tokens
      <0> @number       {mkNumber}
      <0> "("           {simpleToken TLParen}
      <0> ")"           {simpleToken TRParen}
      <0> "+"           {simpleToken TPlus}
      <0> "*"           {simpleToken TTimes}
```

Analisadores léxicos

- Expressões regulares.
  - O "." representa qualquer caractere diferente da quebra de linha.

Analisadores léxicos

- Cada token é formado por:
  - Especificação do estado do analisador (<0>).
  - Expressão regular (@number).
  - Ação semântica executada quando do reconhecimento (mkNumber).

Analisadores léxicos

- Expressões regulares.
  - macros usando $: definem conjuntos de caracteres
  - macros usando @: definem expressões regulares.

Analisadores léxicos

- Exemplo: macro de caractere

```
$digit = 0-9
```

Analisadores léxicos

- Exemplo: macro de expressões regulares

```
@number = $digit+
```

Analisadores léxicos

- Especificando a criação de tokens

```
tokens :-
      -- whitespace and comments
      <0> $white+       ;
      -- other tokens
      <0> @number       {mkNumber}
      <0> "("           {simpleToken TLParen}
      <0> ")"           {simpleToken TRParen}
      <0> "+"           {simpleToken TPlus}
      <0> "*"           {simpleToken TTimes}
      <0> "-"           {simpleToken TMinus}
```

Analisadores léxicos

- Especificando a criação de tokens
  - Para cada ER, apresentamos código para construir o token correspondente
  - Deve ter tipo `AlexInput -> Int64 -> Alex Token`
- Tipo `AlexInput`

```
type AlexInput = (AlexPosn,    -- current position,
                  Char,        -- previous char
                  ByteString,  -- current input string
                  Int64)       -- bytes consumed so far
```

Analisadores léxicos

- Exemplo: criando token de número

```
mkNumber :: AlexAction Token
mkNumber (st, _, _, str) len
  = pure $ Token (position st) (TNumber $ read $ take len str)
```

Analisadores léxicos

- Exemplo: criando token de operadores e separadores

```
simpleToken :: Lexeme -> AlexAction Token
simpleToken lx (st, _, _, _) _
  = return $ Token (position st) lx
```

Analisadores léxicos

- Função top-level do analisador.

```
lexer :: String -> Either String [Token]
lexer s = runAlex s go
  where
    go = do
      output <- alexMonadScan
      if lexeme output == TEOF then
        pure [output]
      else (output :) <$> go
```

Analisadores léxicos

- Especificação de exemplo:
  - pasta `Alex/LexerExample.x`

Analisadores léxicos

- Produzindo o código Haskell do analisador.
  - Construído automaticamente pelo *cabal*.

```
alex LexerExample.x -o LexerExample.hs
```

Analisadores léxicos

- Outros detalhes da especificação.
  - wrapper do analisador.
  - definição do estado do analisador.
  - definição de outros estados e transições entre eles.

Analisadores léxicos

- Wrapper do analisador: define o "modelo" de código a ser produzido pelo gerador Alex.
  - No exemplo, usamos o mais geral dos templates.

```
%wrapper "monadUserState"
```

Analisadores léxicos

- Definição do estado do analisador
  - Qualquer tipo Haskell cujo nome deve ser `AlexUserState`.

```
data AlexUserState
  = AlexUserState {
      nestLevel :: Int -- comment nesting level
    }
```

Analisadores léxicos

- Estado inicial do analisador.
  - Deve possui o nome `alexInitUserState` de tipo `AlexUserState`.

```
alexInitUserState :: AlexUserState
alexInitUserState
  = AlexUserState 0
```

Analisadores léxicos

- Interface para manipular o estado.

```
get :: Alex AlexUserState
get = Alex $ \s -> Right (s, alex_ust s)

put :: AlexUserState -> Alex ()
put s' = Alex $ \s -> Right (s{alex_ust = s'}, ())

modify :: (AlexUserState -> AlexUserState) -> Alex ()
modify f
  = Alex $ \s -> Right (s{alex_ust = f (alex_ust s)}, ())
```

Analisadores léxicos

- Transições entre estados:

```
-- multi-line comment
<0> "\*"              { nestComment `andBegin` state_comment }
<0> "*/"              {\ _ _ -> alexError "Error! Unexpected close comment!" }
<state_comment> "\*"  { nestComment }
<state_comment> "*/"  { unnestComment }
<state_comment> .     ;
<state_comment> \n    ;
```

Conclusão

Conclusão

- Análise léxica é responsável por decompor o código em **tokens**.
- Eliminar comentários, espaços em branco do código.

Conclusão

- Análise léxica pode ser automatizada utilizando...
  - Expressões regulares e autômatos finitos.
- No contexto de Haskell, podemos utilizar o gerador Alex.

Conclusão

- Existem geradores de analisadores léxicos para outras linguagens?
  - Sim! O primeiro foi o **lex** para C.
  - Grande parte das linguagens possuem ferramentas similares.

Conclusão

- Vantagens de uso de geradores:
  - **Eficiência**: código gerado é bastante eficiente.
  - **Manutenção**: fácil de incluir / remover tokens da linguagem.

Conclusão

- Próxima aula: Análise sintática descendente recursiva.