Análise Léxica

Construção de compiladores I

Objetivos

Objetivos

- Apresentar a linguagem a ser utilizada para exposição dos conceitos da disciplina.
- Definir a etapa de análise léxica e apresentar um analisador léxico ad-hoc para a linguagem considerada.

Linguagem Imp

Linguagem Imp

- Linguagem simples utilizada para exposição dos conceitos da disciplina.
- Permite a análise de problemas recorrentes no projeto de compiladores.

Linguagem Imp

- Porém, como especificar uma linguagem de programação?

Linguagem Imp

- Sintaxe definida usando gramáticas livres de contexto.
  - Alguns elementos são descritos por Expressões regulares.

Linguagem Imp

- Elementos lxicos são representados utilizando pela string propriamente dita
  - Ex.: if, while e outras palavras reservadas.
- Outros elementos léxicos: identificadores e números.

Linguagem Imp

- Identificadores: letter(letter + digit)*
  - letter = a + b + c + ... + A + B + ...
  - digit = 0 + 1 + ... + 9

Linguagem Imp

- Comentários:
  - Linha: // este é um comentário.
  - Bloco: /* este é um comentário. */

Linguagem Imp

- Sintaxe de Imp

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Stmts} \\
\mathit{Stmts} & \rightarrow & \mathit{Statement}\ \mathit{Stmts} \mid \lambda
\end{array}
$$

Linguagem Imp

- Sintaxe de Imp

$$
\begin{array}{lcl}
\mathit{Statement} & \rightarrow & \texttt{skip}; \mid \mathit{Type}\ \text{id}\ \mathit{Init}; \\
& \mid & \text{id} := \mathit{Expr}; \\
& \mid & \texttt{read}\ \text{id}; \mid \texttt{print}\ \mathit{Expr}; \\
& \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Block} \\
& \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Block}\ \text{else}\ \mathit{Block} \\
& \mid & \texttt{while}\ \mathit{Expr}\ \mathit{Block}
\end{array}
$$

Linguagem Imp

- Sintaxe de Imp (Continuação)

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
& \mid & -\ \mathit{Expr} \mid (\mathit{Expr}) \\
& \mid & !\ \mathit{Expr} \\
& \mid & \text{number} \mid \text{id} \\
& \mid & \text{true} \mid \text{false}
\end{array}
$$

Linguagem Imp

- Sintaxe de Imp (Continuação)

$$
\begin{array}{lcl}
Op & \rightarrow & + \mid - \mid * \mid / \\
 & \mid & \&\& \mid < \mid == \\
Type & \rightarrow & \text{int} \mid \text{bool} \\
Block & \rightarrow & \{Statement^*\} \\
Init & \rightarrow & := Expr \mid \lambda
\end{array}
$$

Análise léxica

Análise léxica

- Primeira etapa do frontend de um compilador.
- Responsável por dividir a entrada em uma sequência de **tokens**.
  - Eliminar espaços em branco e comentários.

Análise léxica

- Token: string que pode ser considerada indivisível pela gramática de uma linguagem.

Análise léxica

- Como implementar um analisador léxico para uma linguagem?
- Intuitivamente, um analisador léxico é uma função de tipo

```
String -> [Token]
```

Análise léxica

- Representação de tokens

```
data Token
  = Id String | Number Int | LPAREN | RPAREN
  | PLUS | TIMES | MINUS | DIV | AND | NOT
  | ASSIGN | SEMI | LT | EQ ...
```

Análise léxica

- Definir uma função de tipo:

```
data Result = Begin | Error String | Success [Token]

lexer :: String -> Result
lexer = foldr step Begin . concatMap words . lines
```

Análise léxica

- Continuação...

```
step :: String -> Result -> Result
step _ (Error s) = Error s
step s Begin = case s of
                "if" -> Success [IF]
                "then" -> Success [THEN]
                ...
```

Análise léxica

- Apesar de possível, essa abordagem possui problemas.
  - Não é escalável.
  - Não é simples eliminar comentários com essa estratégia.

Análise léxica

- Dificuldade com comentários
  - Funções `lines` e `words` dividem strings em linhas e palavras.
  - Problema: comentários em bloco.

Análise léxica

- Seria possível realizar a análise léxica de forma:
  - Sistemática
  - Escalável
  - Eficiente

Análise léxica

- A resposta para as perguntas anteriores é **SIM**
- Para resolvermos o dilema da análise léxica usaremos:
  - Autômatos finitos determinísticos (AFDs)
  - Expressões regulares (ERs)

Análise léxica

- Intuitivamente:
  - Processamento da entrada usando AFDs.
  - Especificação de lexemas utilizando ERs.

Autômatos finitos

Autômatos finitos

- Um AFD é uma quíntupla $M = (E, \Sigma, \delta, i, F)$:
  - E: conjunto de estados.
  - Σ: alfabeto de entrada.
  - δ : E $\times$ Σ $\rightarrow$ E: função de transição.
  - $i \in E$: estado inicial.
  - $F \subseteq E$: conjunto de estados finais.

Autômatos finitos

- Representando um autômato em Haskell

```
data DFA a
  = DFA {
      start :: a
    , delta :: a -> Char -> a
    , finals :: a -> Bool
    }
```

Autômatos finitos

- Processando uma string em um DFA

```
deltaStar :: DFA a -> String -> a
deltaStar m = foldl (delta m) (start m)
```

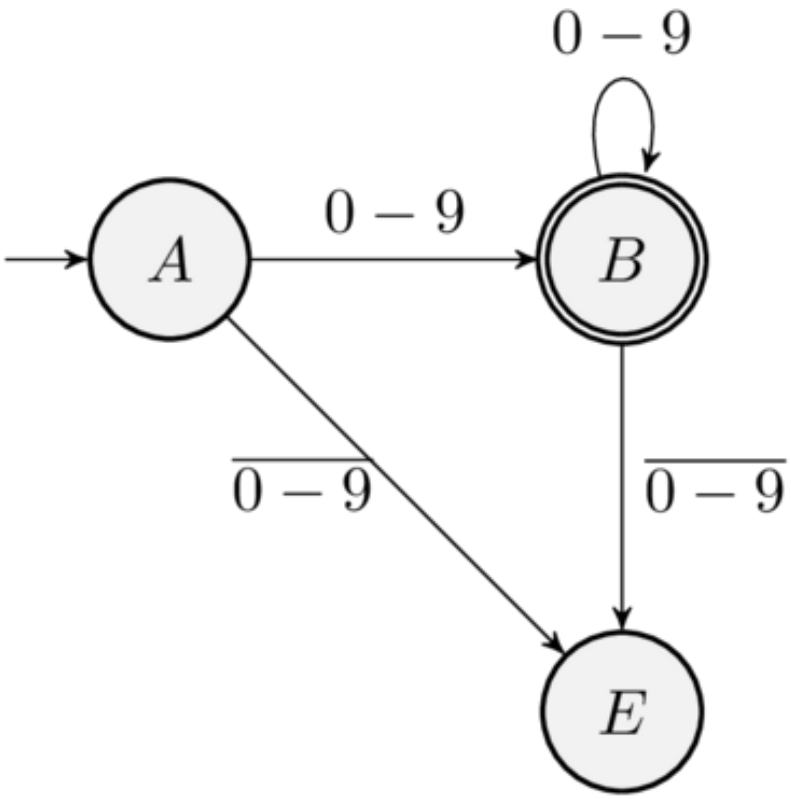
Autômatos finitos

- Relembrando:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ v []       = v
foldl f v (x : xs) = foldl (f v x) xs
```

Autômatos finitos

- Exemplo: Aceitando números.



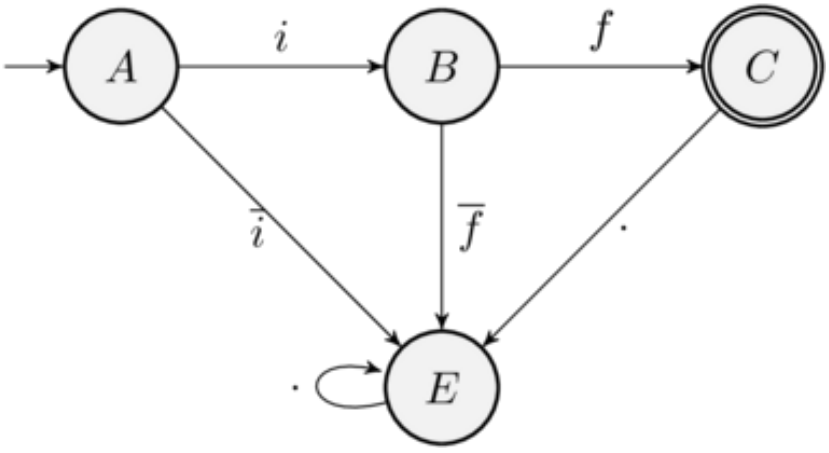
Autômatos finitos

- Exemplo em código Haskell

```
numberDFA :: DFA (Maybe Bool)
numberDFA
  = DFA {
      start = Just False
    , delta = numberTrans
    , finals = \ e -> e == Just True
    }
    where
      numberTrans (Just False) c
        | isDigit c = Just True
        | otherwise = Nothing
      numberTrans (Just True) c
        | isDigit c = Just True
        | otherwise = Nothing
      numberTrans _ _ = Nothing
```

Autômatos finitos

- Exemplo: aceitando palavras chave



Autômatos finitos

- Exemplo em código Haskell

```
ifDFA :: DFA (Maybe Int)
ifDFA
  = DFA {
      start = Just 0
    , delta = ifTrans
    , finals = \ e -> e == Just 2
    }
 where
   ifTrans (Just 0) 'i' = Just 1
   ifTrans (Just 1) 'f' = Just 2
   ifTrans _ _ = Nothing
```

Autômatos finitos

- Problema: lexemas da linguagem não são disjuntos.
  - Considere os tokens if e iftrue
  - Como um AFD deve lidar com essa situação?

Autômatos finitos

- O analisador léxico deve considerar como token o **maior prefixo consumido**.
- Logo, entre if e iftrue, deverá ser escolhido o segundo.
  - Como processar o maior prefixo possível?

Autômatos finitos

- Obtendo o maior casamento em um AFD
- Função: **go**
  - 1o argumento: Estado atual.
  - 2o argumento: Tripla formada por:
    - Último estado é final?
    - Prefixo processado e sufixo restante.

```
longest :: DFA a -> String -> Maybe (String, String)
longest m s
  = go (start m) (finals m (start m), Just "", Just  s)
```

Autômatos finitos

- Definição de `go`.
- Caso 1: String completamente processada.

```
go _ (True, Just pre, Just "") = Just (pre, "")
go _ (False, _, Just "") = Nothing
```

Autômatos finitos

- Definição de `go`
- Caso 2: Casamento já encontrado.

```
go e (True, Just pre, Just (c : cs))
  | finals m (delta m e c) = go (delta m e c) (True, Just (c : pre), Just cs)
  | otherwise   = Just (pre, (c : cs))
```

Autômatos finitos

- Definição de `go`
- Caso 3: Casamento ainda não encontrado

```
go e (False, Just pre, Just (c : cs))
  | finals m (delta m e c) = go (delta m e c)(True, Just (c : pre), Just cs)
  | otherwise = go (delta m e c) (False, Just (c : pre), Just cs)
```

Autômatos finitos

- Como processar todos os tokens de uma entrada?

```
lexer :: DFA a -> (String -> Maybe [b]) -> String -> Maybe [b]
lexer _ _ "" = return []
lexer m action s
  = do
      (pref, suf) <- longest m s
      token <- action (reverse pref)
      tokens <- lexer m action suf
      return (token ++ tokens)
```

Autômatos finitos

- Pergunta: como combinar os AFDs para...
  - Palavras chaves
  - Identificadores

Autômatos finitos

- Vamos utilizar a construção da união de AFDs.
  - União definida em termos de produto

Autômatos finitos

- Construção de produto

```
dfaProduct :: (Eq a, Eq b) => DFA a ->
                              DFA b ->
                              ((a,b) -> Bool) -> DFA (a, b)
dfaProduct m1 m2 fin
  = DFA {
      start = (start m1, start m2)
    , delta = delta'
    , finals = fin
    }
    where
      delta' (e1,e2) c = (delta m1 e1 c, delta m2 e2 c)
```

Autômatos finitos

- Definindo a união

```
unionDFA :: (Eq a, Eq b) => DFA a -> DFA b -> DFA (a,b)
unionDFA m1 m2 = dfaProduct m1 m2 g
  where
    g (e1, e2) = finals m1 e1 || finals m2 e2
```

Autômatos finitos

- Resolvendo o problema entre "if" e "ifblabla".

```
ifOrIdentDFA :: DFA (Maybe Int, Maybe Int)
ifOrIdentDFA = unionDFA ifDFA identDFA
```

Concluindo

Concluindo

- Apresentamos a linguagem IMP que será usada neste curso.
- Discutimos como implementar análise léxica.

Concluindo

- Mostramos que AFDs são um formalismo apropriado para denotar analisadores léxicos.
- Próxima aula: Especificando lexemas utilizando expressões regulares.

Exercícios

Exercícios

- Utilize as implementações de AFDs para criar um analisador léxico para a linguagem IMP. Seu programa deve processar um programa de entrada e imprimir a lista de tokens reconhecidos.