# Parsing Expression Grammars

Construção de compiladores I

## Objetivos

### Objetivos

- Apresentar o formalismo de Parsing Expression Grammars (PEGs).
- Apresentar a sintaxe, semântica e boa-formação de PEGs.

### Objetivos

- Apresentar um sistema de tipos para PEGs.
- Apresentar uma implementação de PEGs em Haskell.

## Introdução

### Introdução

- Gramáticas livres de contexto são o **padrão** para especificar linguagens.
- Problemas: **ambiguidade**

### Introdução

- Apesar de existirem artifícios para eliminar problemas de ambiguidade, no geral este é um problema **indecidível**.

### Introdução

- GLCs são um formalismo **gerativo**, isto é, gramáticas geram palavras.
- Usamos GLCs para especificar **reconhecedores**.

## Introdução

- Essa diferença parece mínima mas é fundamental!

## Introdução

- Para entender a diferença, vamos considerar a linguagem de palavras com uma quantidade par de zeros.

## Introdução

- Especificação baseada em geração.
  - Palavras são construídas acrescentando um número finito de 00.

$\{s \in 0^* \mid \exists n \in \mathbb{N}.s = (00)^n\}$

## Introdução

- Especificação baseada em reconhecimento
  - Palavras são aceitas se tem tamanho par.

$\{s \in 0^* \mid |s| \texttt{mod} 2 = 0\}$

## Introdução

- Uma alternativa recente são as chamadas Parsing Expression Grammars (PEGs).
- Permitem a especificação de **reconhecedores**.

## Introdução

- Vantagem: não temos mais o problema de ambiguidade.
- Expressividade: PEGs podem expressar linguagens sensíveis ao contexto.

# Parsing Expression Grammars

## Parsing Expression Grammars

- Uma PEG G = (V, Σ, R, e) é tal que:
  - V: conjunto de não terminais da gramática.
  - Σ: alfabeto.
  - R: função total que associa uma *parsing expression* a cada não terminal.
  - e: expressão inicial da gramática.

## Parsing Expression Grammars

- Sintaxe de expressões

$$\begin{array}{rcl} e & \rightarrow & \lambda \mid a \in \Sigma \mid A \in V \\ & \mid & e\, e \mid e \,/\, e \mid e^* \mid !\, e \end{array}$$

## Parsing Expression Grammars

- Qual o significado de uma parsing expression?
- Definido em termos de uma relação $\Rightarrow_G$, em que $G = (V, \Sigma, R, e)$.

## Parsing Expression Grammars

- Relação $(e, s) \Rightarrow (s_p, s_s)$ denota:
  - Expressão $e$ processa o prefixo $s_p$ de $s$, deixando o sufixo $s_s$.
  - Caso exista um erro de parsing, utilizamos o símbolo $\perp$.

## Parsing Expression Grammars

- Regra para $e = \lambda$

$$\frac{}{(\lambda, s) \Rightarrow_G (\lambda, s)}$$

## Parsing Expression Grammars

- Regras para $e = a$

$$\frac{}{(a, as) \Rightarrow_G (a, s)} \quad \frac{a \neq b}{(b, as) \Rightarrow_G \perp} \quad \frac{}{(a, \lambda) \Rightarrow_G \perp}$$

## Parsing Expression Grammars

- Regras para $e = A$

$$\frac{A \leftarrow e \in R \quad (e, s) \Rightarrow_G r}{(A, s) \Rightarrow_G r}$$

## Parsing Expression Grammars

- Regras para $e = e_1\ e_2$

$$\frac{(e_1, s) \Rightarrow_G (s_{p1}, s_f) \quad (e_2, s_f) \Rightarrow_G (s_{p2}, s_s)}{(e_1\ e_2, s) \Rightarrow_G (s_{p1}\ s_{p2}, s_s)}$$

$$\frac{(e_1, s) \Rightarrow_G (s_{p1}, s_f) \quad (e_2, s_f) \Rightarrow_G \bot}{(e_1\ e_2, s) \Rightarrow_G \bot}$$

$$\frac{(e_1, s) \Rightarrow_G \bot}{(e_1\ e_2, s) \Rightarrow_G (s_{p1}\ s_{p2}, s_s)}$$

## Parsing Expression Grammars

- Regras para $e = e_1 \,/\, e_2$

$$\frac{(e_1, s) \Rightarrow_G (s_p, s_s)}{(e_1 \,/\, e_2, s) \Rightarrow_G (s_p, s_s)} \quad \frac{(e_1, s) \Rightarrow_G \bot \quad (e_2, s) \Rightarrow_G r}{(e_1 \,/\, e_2, s) \Rightarrow_G r}$$

## Parsing Expression Grammars

- Regras para $e = e_1^*$

$$\frac{(e_1, s) \Rightarrow_G (s_{p1}, s_f) \quad (e_1^*, s_f) \Rightarrow_G (s_{p2}, s_s)}{(e_1^*, s) \Rightarrow_G (s_{p1}\ s_{p2}, s_s)} \quad \frac{(e_1, s) \Rightarrow_G \bot}{(e_1^*, s) \Rightarrow_G (\lambda, s)}$$

## Parsing Expression Grammars

- Regras para $e = !\, e_1$

$$\frac{(e_1, s) \Rightarrow_G (s_p, s_s)}{(!\, e_1, s) \Rightarrow_G \bot} \quad \frac{(e_1, s) \Rightarrow_G \bot}{(!\, e_1, s) \Rightarrow_G (\lambda, s)}$$

### Parsing Expression Grammars

- Exemplo

$A \leftarrow aAb \,/\, ab$

- Expressão inicial: $A/\,!\,(a\,/\,b)\,/\,\lambda$
- Exemplos da semântica: aabb, a, $\lambda$

### Parsing Expression Grammars

- Qual o problema com a seguinte gramática?
  - Expressão inicial: A

$A \leftarrow Aa \,/\, a$

### Parsing Expression Grammars

- Qual o problema com a seguinte expressão?
  - $(a^*)^*$

### Parsing Expression Grammars

- Assim como programas em geral, PEGs podem entrar em **loop**.
- Basicamente, PEGs são especificações de parsers descendentes recursivos.

## Terminação em PEGs

### Terminação em PEGs

- Dizemos que uma expressão $e$ sucede com uma string $s$ se:
  - $\exists s_p\, s_s.(e, s) \Rightarrow_G (s_p, s_s)$

### Terminação em PEGs

- Relação $e \rightharpoonup o$, $o \in \{0, 1, f\}$.
  - Caso $s_p = \lambda$, temos que $e \rightharpoonup 0$.
  - Caso $s_p = ay$, temos que $e \rightharpoonup 1$.
  - Em caso de falha, temos que $e \rightharpoonup f$.

### Terminação em PEGs

- Dizemos que uma expressão é bem formada se:
  - Não possui recursão à esquerda direta / indireta.
  - Não é da forma $e^*$, em que $e \rightharpoonup 0$.

### Terminação em PEGs

- Uma PEG é **bem formada** se todas as suas subexpressões são bem formadas.

### Terminação em PEGs

- Definição original de boa formação é feita em dois passos:
  - Primeiro, define-se quando uma expressão é bem formada.
  - Depois, computa-se o ponto fixo do conjunto de expressões bem formadas de uma gramática.

### Terminação em PEGs

- Problemas
  - Definição não composicional
  - Difícil compreensão.
- Solução: Uso de tipos.
  - Falaremos sobre isso quando estudarmos análise semântica.

## Implementação

### Implementação

- Vamos apresentar uma implementação de PEGs usando combinadores.
- Note que nossa implementação não irá garantir a terminação!
  - Problema de pesquisa em aberto!

## Implementação

- Representando o resultado de uma expressão

```
data Result s a
  = Pure a           -- did not consume anything. We can backtrack.
  | Commit s a       -- remaining input and result.
  | Fail String Bool -- true if consume any input
  deriving (Show, Functor)
```

## Implementação

- Definindo uma expressão.

```
newtype PExp s a
  = PExp {
      runPExp :: s -> Result s a
    } deriving Functor
```

## Implementação

- Representando a entrada

```
class Stream a where
  anyChar :: PExp a Char

instance Stream String where
  anyChar = PExp $ \ d ->
    case d of
      (x : xs) -> Commit xs x
      []       -> Fail "eof" False
```

## Implementação

- Expressões são applicatives

```
instance Applicative (PExp s) where
  pure x = PExp $ \ _ -> Pure x
  (PExp efun) <*> (PExp earg)
    = PExp $ \ d ->
        case efun d of
          Pure f   -> f <$> earg d
```

```
          Fail s c -> Fail s c
          Commit d' f ->
            case earg d' of
              Pure a -> Commit d' (f a)
              Fail s' _ -> Fail s' True
              Commit d'' a -> Commit d'' (f a)
```

## Implementação

- Expressões são alternatives

```
instance Alternative (PExp d) where
  (PExp e1) <|> (PExp e2) = PExp $ \ d ->
    case e1 d of
      Fail _ False -> e2 d
      x            -> x
  empty = PExp $ \ _ -> Fail "empty" False
```

## Implementação

- Controlando backtracking

```
try :: PExp d a -> PExp d a
try (PExp m) = PExp $ \ d ->
  case m d of
    Fail s _ -> Fail s False
    x        -> x
```

## Implementação

- Escolha ordenada

```
(</>) :: PExp d a -> PExp d a -> PExp d a
e1 </> e2 = try e1 <|> e2
```

## Implementação

- Símbolos e $\lambda$

```
satisfy :: Stream d => (Char -> Bool) -> PExp d Char
satisfy p = do
  x <- anyChar
```

```
  x <$ guard (p x)

symbol :: Stream d => Char -> PExp d Char
symbol c = satisfy (c ==)

lambda :: Stream d => a -> PExp d a
lambda v = PExp $ \ d -> Commit d v
```

## Implementação

- Operador star

```
star :: Stream d => PExp d a -> PExp d [a]
star e1 = PExp $ \ d ->
  case runPExp e1 d of
    Fail _ _ -> Commit d []
    Pure _ -> Fail "Nullable star" False
    Commit d' v ->
      case runPExp (star e1) d' of
        Fail _ _ -> Commit d []
        Commit d'' vs -> Commit d'' (v : vs)
        Pure _ -> Fail "Nullable star" False
```

## Implementação

- Operador not

```
not :: Stream d => PExp d a -> PExp d ()
not e = PExp $ \ d ->
  case runPExp e d of
    Fail _ _ -> Pure ()
    _        -> Fail "not" True

and :: Stream d => PExp d a -> PExp d ()
and e = not $ not e
```

## Implementação

- Exemplo: $\{a^n b^n | n \geq 1\}$

```
ab1 :: PExp String String
```

```
ab1 = (f <$> symbol 'a' <*> ab <*> symbol 'b') </>
      (g <$> symbol 'a' <*> symbol 'b')
  where
    f x s y = x : s ++ [y]
    g x y = [x,y]
```

### Implementação

- Exemplo: $\{b^n c^n | n \geq 1\}$

```
bc1 :: PExp String String
bc1 = (f <$> symbol 'b' <*> bc <*> symbol 'c') </>
      (g <$> symbol 'b' <*> symbol 'c')
  where
    f x s y = x : s ++ [y]
    g x y = [x,y]
```

### Implementação

- Exemplo $\{a^n b^n c^n | n \geq 1\}$

```
abc1 :: PExp String String
abc1 = f <$> and (ab1 *> not b) <*>
             star a             <*>
             and b              <*>
             bc1                <*>
             not anyChar
  where
    a = symbol 'a'
    b = symbol 'b'
    f _ as _ bcs _ = as ++ bcs
```

## Concluindo

### Concluindo

- Nesta aula apresentamos uma alternativa às GLCs: PEGs
- Apresentamos a sintaxe e semântica de PEGs
  - Discutimos o problema de terminação.

### Concluindo

- Apresentamos uma implementação de PEGs em Haskell
- PEGs não são equivalentes a GLCs
  - Não existe PEG para palíndromos
  - Existe PEG para $\{a^n b^n c^n \mid n \geq 1\}$

### Concluindo

- Próxima aula: análise sintática ascendente.

## Exercícios

### Exercícios

- Usando a implementação apresentada, construa uma PEG para expressões aritméticas envolvendo números, variáveis, adição, multiplicação e parêntesis.