3
Parsing Expression Grammars

Parsing Expression Grammars (PEG) são um formalismo que descreve reconhecedores de linguagens (For04). PEGs são uma alternativa para gramáticas livres de contexto (CFGs) e expressões regulares, e nos permite reconhecer diversas classes de linguagens. Por exemplo, PEGs reconhecem todas as linguagens livres de contexto *LR(k)* determinísticas e algumas linguagens sensíveis a contexto (For04).

A principal característica de PEGs é o *determinismo*. Isto é, dado uma PEG, esta casa apenas de uma forma com uma determinada entrada. O determinismo de PEGs é resultado de duas outras características fundamentais: falha e escolha ordenada. Dizemos que um padrão falha quando este não casa com um dado prefixo da entrada. Escolha ordenada é uma operação que define prioridade em uma sequência de alternativas através do operador barra (/). Em escolhas ordenadas, uma alternativa só será testada se a alternativa anterior tiver falhado.

Uma PEG é uma gramática G composta por uma 4-tupla (V_n, V_t, R, e_S) onde V_n é um conjunto finito de não-terminais; V_t é um conjunto finito de terminais sendo que $V_n \cap V_t = \emptyset$; R é um conjunto de regras $\texttt{A} \leftarrow \texttt{e}$ onde $\texttt{A} \in V_n$ e `e` é uma *expressão de parsing*; e e_S é a expressão de parsing usada como ponto de partida.

Os operadores que constroem expressões de parsing são listados na tabela abaixo.

Em PEGs, literais devem ser delimitados por aspas simples ou aspas duplas. Colchetes definem classes de caracteres semelhantes as classes de caracteres dos regexes. Estas podem conter intervalos como `[0-9]` e `[a-zA-Z]` ou uma lista de caracteres como `[abc123_]`. O operador ".", assim como nos regexes, casa com qualquer caracter.

A concatenação $\texttt{e}_1\,\texttt{e}_2$ tenta casar $\texttt{e}_2$ após o casamento de $\texttt{e}_1$; caso $\texttt{e}_1$ ou $\texttt{e}_2$ falhe, toda a concatenação falha.

A escolha ordenada define prioridade em uma lista de alternativas. Dessa forma, em um padrão $\texttt{e}_1/\texttt{e}_2$, o padrão $\texttt{e}_2$ só é testado se $\texttt{e}_1$ falhar. Se $\texttt{e}_1$ casar, $\texttt{e}_2$ nunca será testado.

Operadores	Descrição
`' '`	literal
`" "`	literal
`[...]`	classe de caracteres
`.`	qualquer caracter
`e?`	opcional
`e*`	zero-ou-mais
`e+`	um-ou-mais
`&e`	predicado de lookahead
`!e`	predicado de negação
$\texttt{e}_1\ \texttt{e}_2$	concatenação
$\texttt{e}_1\ /\ \texttt{e}_2$	escolha ordenada

Tabela 3.1: Operadores de PEGs

Uma consequência interessante da escolha ordenada é o backtracking restrito. Isso significa que PEGs permitem apenas backtrackings *locais*, isto é, uma PEG faz somente backtrackings para escolher uma opção de uma escolha ordenada. Assim que uma opção casar com a entrada, esta não pode ser desfeita mesmo que os padrões seguites falhem. Para ilustrar os backtrackings locais de PEGs, considere o exemplo abaixo:

$$\texttt{S} \leftarrow \texttt{A B}$$
$$\texttt{A} \leftarrow \texttt{p}_1\ /\ \texttt{p}_2\ /\ \ldots\ /\texttt{p}_n$$

Para casar `S` com a entrada, é necessário casar a expressão `AB`. Para casar `A`, tentamos casar a primeira alternativa $\texttt{p}_1$; se $\texttt{p}_1$ falhar, a PEG retrocede para a posição inicial da entrada (backtracking local) e tenta casar $\texttt{p}_2$ e assim por diante. Assim que um padrão $\texttt{p}_i$ casar, a PEG considera que `A` casou com um prefixo da entrada e tenta casar `B`. Caso `B` falhe, não haverá backtracking para `A`.

Os operadores `*`, `+` e `?` são repetições semelhantes às repetições de regexes, mas são cegas, isto é, em uma expressão $\texttt{e}_1\texttt{*}\ \texttt{e}_2$ o padrão $\texttt{e}_1$ repete o máximo de vezes possível (guloso) sem se importar se o restante da expressão, no caso $\texttt{e}_2$, irá casar (cego). O comportamento cego das repetições é uma consequência das escolhas ordenadas. Basicamente, a repetição `e*` é um açúcar sintático para a PEG abaixo:

$$\texttt{A} \leftarrow \texttt{e A / ""}$$

Para casar `A`, a PEG tenta casar o padrão `e` repetidas vezes e, assim que o casamento de uma expressão `e` falhar, a PEG tenta casar com a segunda opção `""`, que casa com o string vazia. Assim que a repetição do padrão `e` falhar, a segunda opção `""` irá casar com a entrada já que a string vazia casa com qualquer prefixo da entrada. Como os backtrackings de PEGs são locais, uma vez que o padrão `A` casa, os caracteres consumidos não estarão sujeitos a backtrackings, o que caracteriza uma repetição cega. De forma análoga, a repetição `+` também é cega por ser apenas um açúcar sintático para a expressão `e e*`.

A operação `?` é um açúcar sintático para a expressão `e/""` e, da mesma forma, é cega por dar prioridade ao casamento da expressão `e`.

PEGs possuem dois predicados sintáticos que não consomem porções da entrada. O predicado de negação, denotado por `!`, define que uma expressão `!e` casa somente se o padrão `e` falhar. Já o predicado de lookahead, denotado por `&`, define que uma expressão `&e` casa somente se `e` casar com a entrada. O predicado de lookahead `&e` é apenas um açúcar sintático para a PEG `!!e`. A PEG abaixo ilustra um exemplo prático que utiliza o predicado de negação para casar comentários de C:

```
C  ←  "/*" (!"*/".)* "*/"
```

Nesse exemplo, padrão `"/*"` casa com o início de um comentário. Logo após, a expressão `(!"*/".)*` consome todos os caracteres que **não** casam com o fim do comentário, isto é, caracteres que compõem o conteúdo do comentário. Assim que essa repetição falhar, a PEG tenta casar o padrão `"*/"` que delimita o fim do comentário.

Os predicados sintáticos são operadores que aumentam o poder de expressão das PEGs. Além de proporcionar um poderoso mecanismo de lookahead, PEGs não precisam de um operador específico para casar com o fim de entrada (denotado por `$` em regexes). O padrão `!.` casa apenas se . (que casa qualquer caracter) falhar; esta falha ocorre apenas no fim da entrada, quando não há mais caracteres a serem casados.

O predicado de negação também permite a construção de uma operação similar ao de complemento. Por exemplo, a PEG abaixo casa todas as palavras minúsculas excluindo as palavras reservadas (`"int"`, `"float"`, etc):

```
Identifier ← !Reserved [a-z]+
  Reserved ← ("int"/ "double"/ ... / "float") ![a-z]
```

Assim como BNF, uma PEG é um conjunto de uma ou mais regras. Abaixo temos a PEG completa que descreve sintaxe de PEGs:

```
     pattern ← grammar / simplepatt
     grammar ← (nonterminal "←" sp simplepatt)+
  simplepatt ← alternative ("/" sp alternative)*
 alternative ← ([!&]? sp suffix)+
      suffix ← primary ([*+?] sp)?
     primary ← "(" sp pattern ")" sp /
               "." sp /
               literal /
               charclass /
               nonterminal !"←"
     literal ← ['] (!['] .)* ['] sp
   charclass ← "["(!"]"(. "-". / .))* "]" sp
 nonterminal ← [a-zA-Z]+ sp
          sp ← [ \t\n]*
```

Uma PEG é composta por uma gramática (`grammar`) ou por um padrão simples (`simplepatt`). Gramáticas são compostas por um conjunto de regras na forma `nonterminal` ← `simplepatt`. Um padrão simples é uma sequência de alternativas separadas por barra (`/`) que correspônde a operação de escolha ordenada. Cada alternativa é uma sequência de `suffix` prefixados opcionalmente por predicados sintáticos e cada `suffix` é uma expressão seguida opcionamente por um quantificador. Por fim, uma expressão pode ser um literal demilitado por aspas simples ou aspas duplas, pode ser um ponto (`.`) que casa com qualquer caracter, uma referência para um não-terminal ou uma expressão agrupada por parênteses.

Abaixo, mostramos um exemplo simplificado de uma PEG que casa com elementos XML:

```
Elemento ← StartTag Value EndTag
StartTag ← "<"Name ">"
  EndTag ← "</" Name ">"
   Value ← Elemento+ / [a-zA-Z0-9_ ]+ / ""
    Name ← [a-zA-Z]+
```

Neste exemplo, um `Elemento` é composto pelo início de uma Tag (`StartTag`), um valor (`Value`) e pelo delimitador que indica o fim da Tag (`EndTag`). Tanto `StartTag` e `EndTag` possuem um nome que, basicamente, é composto por uma sequência de um ou mais caracteres alfabéticos. Cada valor pode ser uma sequência de elementos, ser um texto ou ser a string vazia (Tag sem valor).

PEGs casam em modo ancorado, isto é, PEGs não fazem busca do padrão na entrada. Porém, podemos contruir facilmente uma PEG que executa a busca de um dado padrão na entrada. Para ilustrar esse comportamento, considere a PEG abaixo, que reconhece expressões aritméticas em modo ancorado:

```
     Exp ← Factor (FactorOp Factor)*
  Factor ← Term (TermOp Term)*
    Term ← "-"? Number
FactorOp ← [+-]
  TermOp ← [*/]
  Number ← [0-9]+
```

Essa PEG reconhece a string `"2*3+4/4-1"`, porém falha ao casarmos com a string `"expressão 2*3+4/4-1"`. Porém, podemos reescrever esta PEG para buscar uma expressão aritmética na entrada. A PEG abaixo ilustra a gramática modificada:

```
S ← Exp / . S
```

`Exp` ← ... (igual à original)

Esta PEG tenta casar a expressão `Exp`; se essa alternativa falhar, a PEG consome um caracter da entrada e tenta casar a PEG inteira novamente. Dessa forma, a PEG consome os caracteres da entrada, um a um, buscando a primeira

ocorrência de uma expressão aritmética.

3.1 LPeg - PEGs em Lua

LPeg é uma implementação de PEGs focada em casamento de padrões (Ier09) e está disponível como um módulo Lua (Ier08). LPeg é uma biblioteca inspirada no casamento de padrão de SNOBOL4 (Gri71), onde os padrões são valores de primeira classe. Embora deixe a criação de padrões mais prolixa, essa abordagem possui diversas vantagens como, por exemplo, facilitar a inserção de comentários entre os padrões, armazenar padrões em variáveis auxiliares, reutilizar padrões e testar cada padrão independentemente.

A implementação de PEGs introduzida por Ford, chamada Packrat (For02), utiliza uma versão alterada do algoritmo que reconhece qualquer gramática *Generalized Top-Down Parsing Language* (GTDPL), um formalismo proposto por Aho & Ullman (AU72). Essa alteração consiste em usar *avaliação preguiçosa* presente em linguagens funcionais modernas como Haskell (Hut07) com o intuito de aumentar a eficiência do casamento. A implementação de LPeg, ao invés de adotar a estratégia de Packrat, utiliza uma *máquina virtual de parsing* onde cada PEG corresponde a um programa dessa máquina (MI08). A vantagem de usar essa abordagem é que a máquina de parsing possui um modelo de desempenho bem definido e uma implementação simples.

Nessa subseção vamos apresentar apenas as operações de LPeg fundamentais para o entendimento deste trabalho. Vamos descrever apenas algumas capturas que LPeg disponibiliza, são elas: capturas simples, capturas de posição e capturas constantes. A documentação completa de LPeg pode ser consultada online (Ier08).

Para explicar as capturas de LPeg, vamos utilizar a sintaxe de PEGs acrescida pelos metacaracteres {} que denotam início e fim de uma captura. Basicamente, uma captura simples consiste em obter o trecho que casou com uma determinada expressão. Utilizamos a sintaxe {`e`} para denotar a captura simples do padrão `e`.

Uma captura de posição consiste em obter a posição atual do casamento. Para denotar uma captura de posição, utilizamos a sintaxe {} (captura vazia).

Capturas constantes são capturas que produzem valores determinados. Para denotar essas capturas, utilizamos a sintaxe {$_c$"v_1", "v_2",..., "v_n"}, onde "v_1", "v_2" e "v_n" são os valores produzidos pela captura.

Como exemplo de captura simples, considere a PEG `"#"{[A-F0-9]+}` que casa com literais do sistema hexadecimal. O casamento dessa PEG com a string `"#0000FF"` resulta na captura do valor `"0000FF"`.

Para sabermos as posições de início e fim do valor hexadecimal, podemos utilizar capturas de posição. Se modificarmos o exemplo anterior para `"#"{}{[A-F0-9]+}{}` e casarmos com a mesma entrada, `"#0000FF"`, obtemos a lista de valores [`2, "0000FF", 8`], onde 2 é a posição capturada pela primeira captura de posição, {}, `"0000FF"` é o valor capturado pela expressão {`[A-F0-9]+`} e 8 é o valor capturado pela segunda captura de posição.

Para ilustrar um uso prático de capturas constantes, considere a PEG abaixo, que casa com expressão que possuem parênteses balanceados:

```
  Parenthesis  ←  "("(NonParenthesis / Parenthesis)* ")"
NonParenthesis ←  ![()] .
```

Podemos modificar a PEG acima de modo que, para cada parenteses `"("` casado, utilizamos uma captura constante que produz a string `"open"` seguida por uma captura de posição. De forma análoga, para cada parenteses `")"` criamos uma captura constante da string `"close"` seguida da captura de posição. Abaixo mostramos a PEG modificada para produzir essas capturas:

```
  Parenthesis  ←  Open (NonParenthesis / Parenthesis)* Close
         Open  ←  {c"open"} {} "("
        Close  ←  {c"close"} {} ")"
NonParenthesis ←  ![()] .
```

Ao casarmos essa PEG com uma expressão que possui parênteses balanceados teremos, como resultado, uma lista de valores onde cada `"open"` e `"close"` é seguido pela posição em que casaram na entrada. Por exemplo, o casamento desta PEG com a string `"(()())"` resulta na lista de valores [`"open", 1, "open", 2, "close", 3, "open", 4, "close", 5, "close", 6`].