1
Introdução

Gramáticas de Expressões de Parsing (*Parsing Expression Grammars* — PEGs) foram propostas em 2002 por Ford [Ford, 2002, 2004] como um formalismo para descrever linguagens. Segundo Ford, o uso de PEGs para descrever linguagens livres de contexto não ambíguas é mais adequado do que o uso de Gramáticas Livres de Contexto (*Context-Free Grammars* — CFGs), uma vez que PEGs não permitem expressar ambiguidade. Ainda segundo Ford, outra vantagem de PEGs em relação a CFGs é a possibilidade de usarmos uma única gramática para descrever os elementos léxicos e sintáticos de uma linguagem.

PEGs são fortemente baseadas em dois formalismos desenvolvidos anteriormente por Alexander Birman [Birman, 1970, Birman e Ullman, 1973] e que foram depois denominados de *Top-Down Parsing Language* (TDPL) e *Generalized Top-Down Parsing Language* (GTDPL) por Aho e Ullman [Aho e Ullman, 1972].

A característica distintiva de PEGs em relação a outros formalismos usados para descrever linguagens, tais como expressões regulares e CFGs, é o uso de um operador de escolha ordenada.

Com PEGs podemos descrever todas as linguagens $LR(k)$, isto é, a classe de linguagens livres de contexto determinísticas. Além disso, através de PEGs podemos descrever também linguagens que não são livres de contexto, como $a^n b^n c^n$, embora seja um problema em aberto se PEGs descrevem todas as linguagens que podem ser descritas por CFGs, isto é, se PEGs descrevem a classe de linguagens livres de contexto.

Apesar de PEGs descreverem todas as linguagens $LR(k)$, não existe uma abordagem formal para obter a partir de uma CFG determinística uma PEG equivalente, isto é, uma PEG que descreve a mesma linguagem. O método usual de obter uma PEG a partir de uma CFG consiste em fazer uma tradução manual da CFG em uma PEG, ou seja, é um processo de tentativa e erro [Redziejowski, 2008].

Acreditamos que uma abordagem mais rigorosa para tratar da correspondência entre CFGs e PEGs é necessária e apresenta algumas vantagens.

Em primeiro lugar, nos ajudaria a formalizar uma transformação entre CFGs e PEGs equivalentes. Em segundo lugar, poderíamos aplicar parte do conhecimento atual sobre CFGs (lemas, ferramentas, etc.) para o âmbito de PEGs. E por fim, teríamos um melhor entendimento da classe de linguagens descrita por PEGs.

Para estudar a correspondência entre classe de CFGs e PEGs, iremos usar novas formalizações de PEGs e CFGs. Essas novas formalizações são baseadas em semântica natural [Kahn, 1987, Winskel, 1993], também conhecida como semântica operacional *big step* [Turbak e Gifford, 2008].

Adotamos essa abordagem para tentar diminuir a distância entre as definições de CFGs e PEGs, pois uma formalização de CFGs mais próxima da formalização de PEGs nos permite ver de maneira mais clara as similaridades e as diferenças entre as semânticas desses formalismos.

Nesta tese, vamos estabelecer uma correspondência de PEGs com linguagens regulares e linguagens $LL(k)$-forte.

Uma maneira de representar linguagens regulares é através de expressões regulares, cuja forma sucinta é muito usada em bibliotecas de casamento de padrões (*pattern matching*) [Friedl, 2006, Goyvaerts e Levithan, 2009], também conhecidas como bibliotecas *regex*. Assim, vamos definir a equivalência entre expressões regulares e PEGs, e apresentar uma transformação entre expressões regulares e PEGs equivalentes. Para alcançar esse objetivo, usaremos uma nova formalização de expressões regulares que, assim como as novas formalizações de PEGs e CFGs que iremos apresentar, é baseada em semântica natural. A transformação entre expressões regulares e PEGs equivalentes que mostraremos pode ser facilmente adaptada de modo a acomodar diversas extensões usadas por bibliotecas de casamento de padrões, tais como repetição preguiçosa e subpadrões independentes, como mostrado em Oikawa et al. [2010].

Outra maneira de representar linguagens regulares é através de CFGs lineares à direita. A correspondência entre CFGs lineares à direita e PEGs foi apontada por Ierusalimschy [Ierusalimschy, 2009], porém uma prova detalhada não foi apresentada. Apresentamos aqui uma investigação mais detalhada desse problema, e estabelecemos a correspondência entre CFGs lineares à direita e PEGs equivalentes.

Apresentamos também um estudo da correspondência entre CFGs $LL(1)$ e PEGs. Uma motivação para esse estudo é que há uma intuição geral de que gramáticas $LL(1)$ definem a mesma linguagem quando interpretadas como CFGs e quando interpretadas como PEGs [1]. Embora essa intuição geral exista,

[1] Veja esta discussão de 2005 a respeito de CFGs $LL(1)$ e PEGs na lista *comp.compilers*: http://compilers.iecc.com/comparch/article/05-08-115.

nenhuma análise mais profunda sobre a veracidade dessa correspondência foi realizada.

O nosso objetivo é provar que gramáticas $LL(1)$, com uma pequena restrição na ordem das alternativas de uma escolha, realmente definem a mesma linguagem quando interpretadas como CFGs e quando interpretadas como PEGs. Além disso, vamos estudar a correspondência entre CFGs $LL(k)$-forte e PEGs, e mostrar como a partir de uma CFG $LL(k)$-forte é possível obter uma PEG equivalente que possui a mesma estrutura da CFG original.

1.1 Visão Geral de PEGs

Nesta seção, iremos apresentar uma visão geral de PEGs. Durante essa apresentação, vamos usar como exemplo uma PEG que descreve a própria sintaxe de PEGs. Esse exemplo, embora seja um pouco complexo, usa várias das construções de PEGs e é próximo de PEGs usadas em aplicações práticas.

Na figura 1.1, podemos ver o exemplo da PEG que descreve a sintaxe de PEGs. Essa figura é uma adaptação de uma figura que aparece no trabalho de Ford [Ford, 2004].

Uma PEG consiste de um conjunto de definições da forma A `->` p, onde A é um não terminal (ou variável), e p, o lado direito de uma definição, é uma *expressão de parsing*, que discutiremos no próximo capítulo[2]. Podemos ver que a descrição de uma PEG é muito parecida com uma descrição na Forma de Backus-Naur Estendida (*Extended Backus-Naur Form* — EBNF) [Wirth, 1977, ISO].

Nas linhas 1–12 da figura 1.1 temos as definições dos elementos sintáticos, e nas linhas 13–36 dessa figura temos as definições dos elementos léxicos. Vamos discutir primeiro as definições dos elementos léxicos.

Na linha 13, temos um comentário. Comentários são indicados pelo símbolo `#` e estendem-se até o final da linha corrente.

Nas linhas 14–23, definimos variáveis que representam um símbolo seguido de espaços. De modo geral, as definições de elementos léxicos usam a variável *Spacing*, definida na linha 25, para casar os espaços à direita de um elemento léxico. Em virtude disso, ao descrever os elementos sintáticos de uma PEG precisamos casar apenas os espaços que aparecem no começo da gramática.

Na linha 14, definimos a variável $RIGHTARROW$, que consiste do símbolo `->` seguido de espaços.

Na linha 15, temos o símbolo /, que é o operador de escolha em PEGs.

[2]Em definições, Ford usa o símbolo `<-` ao invés do símbolo `->`.

```
01  # Sintaxe hierárquica
02  Grammar    -> Spacing Definition+ EndOfFile
03  Definition -> Identifier RIGHTARROW Expression
04
05  Expression -> Sequence (SLASH Sequence)*
06  Sequence   -> Prefix*
07  Prefix     -> (AND / NOT)? Suffix
08  Suffix     -> Primary (QUESTION / STAR / PLUS)?
09  Primary    -> Identifier !RIGHTARROW
10              / OPEN Expression CLOSE
11              / Literal / Class / DOT
12
13  # Sintaxe léxica
14  RIGHTARROW -> '->' Spacing
15  SLASH      -> '/' Spacing
16  AND        -> '&' Spacing
17  NOT        -> '!' Spacing
18  QUESTION   -> '?' Spacing
19  STAR       -> '*' Spacing
20  PLUS       -> '+' Spacing
21  OPEN       -> '(' Spacing
22  CLOSE      -> ')' Spacing
23  DOT        -> '.' Spacing
24
25  Spacing   -> (Space / Comment)*
26  Comment   -> '#' (!EndOfLine .)* EndOfLine
27  Space     -> ' ' / '\t' / EndOfLine
28  EndOfLine -> '\r\n' / '\n' / '\r'
29  EndOfFile -> !.
30
31  Identifier -> [a-zA-Z_] ([a-zA-Z_] / [0-9])* Spacing
32  Literal    -> "'" (!"'" Char)* "'" Spacing
33              / '"' (!'"' Char)* '"' Spacing
34  Class      -> '[' (!']' Range)* ']' Spacing
35  Range      -> Char '-' Char / Char
36  Char       -> '\\'. / !'\\'.
```

Figura 1.1: PEG que Descreve a Sintaxe de PEGs

Nas linhas 16 e 17, aparecem, respectivamente, os símbolos & e !. O símbolo & é o operador de um predicado de afirmação, enquanto que o símbolo ! é o operador de um predicado de negação.

Nas linhas 18–20, temos os operadores de repetição ?, $*$, e $+$. Assim como em bibliotecas regex, o operador ? indica um casamento opcional (zero ou uma repetição). O operador $*$ indica zero ou mais repetições, enquanto que $+$ indica uma ou mais repetições.

Fechando esse bloco de definições, na linha 23 temos o símbolo `.` que é usado para casar qualquer caractere.

Em seguida, nas linhas 25–29, definimos espaços, comentários, fim de linha e fim de arquivo. Note que na linha 29 usamos o predicado `!.` para definir o fim de arquivo. Como a expressão `.` casa qualquer caractere, a negação dessa expressão só casa quando a entrada não possui mais nenhum caractere.

Na linha 31, temos a definição de um identificador. Um identificador é uma letra ou sublinhado seguido por zero ou mais letras, sublinhados ou dígitos. Note o uso da variável *Spacing* para casar os espaços à direita de um identificador.

Nas linhas 32–33, temos a definição de um literal. Um literal consiste de zero ou mais caracteres delimitados por aspas simples ou duplas. Nessa definição usamos o predicado de negação. O casamento da expressão `!"'"` é bem sucedido quando o próximo caractere da entrada é diferente de `'`. De modo análogo, o casamento de `!'"'` é bem sucedido quando o próximo caractere da entrada é diferente de `"`. Assim, temos que a expressão $(!\text{'"'}\ Char)*$ casa enquanto o próximo caractere da entrada é diferente de `"`.

A linha 34 apresenta a definição de classes de caracteres, uma construção que também está presente em bibliotecas regex. Assim como nessas bibliotecas, classes de caracteres são delimitadas por colchetes e podem conter zero ou mais intervalos, que são definidos na linha 35. Exemplos de intervalos são `a-z` e `0-9`. Ao contrário de classes de caracteres usadas em bibliotecas regex, as classes de caracteres de PEGs não possuem um operador de complemento. Apesar dessa limitação, é possível obter o complemento de uma classe de caracteres em PEGs usando o predicado de negação. Por exemplo, como o símbolo `.` casa qualquer caractere, temos que a expressão (`![0-9].`) casa qualquer caractere que não é um dígito.

Na linha 36, temos a definição de um caractere. Nessa linha, usamos o literal `'\\'` para casar o símbolo \, que é usado em PEGs para indicar um caractere de escape. Exemplos de caracteres de escape são `\n` e `\t`.

Agora, vamos discutir as definições dos elementos sintáticos nas linhas 1–12 da figura 1.1.

Na linha 1 temos um comentário, e na linha 2 temos a definição de uma gramática. Uma gramática consiste de espaços seguidos por uma ou mais definições seguidas pelo fim de arquivo.

A linha 3 nos diz que uma definição consiste de um identificador seguido por `->` seguido por uma expressão.

A linha 5 possui a definição de uma expressão. Uma expressão é uma sequência seguida por zero ou mais sequências separadas pelo operador de escolha.

Na linha 6 definimos uma sequência como zero ou mais prefixos, e na linha 7 temos a definição de um prefixo. Um prefixo consiste de um sufixo opcionalmente precedido por um dos operadores de predicado.

Na linha 8, definimos um sufixo como uma expressão primária opcionalmente seguida por um dos operadores de repetição.

As linhas 9–11 possuem a definição de uma expressão primária. Na linha 9, usamos o predicado de negação `!` para indicar que quando uma expressão primária é um identificador, esse identificador não pode ser seguido por `->`. Essa restrição evita que a expressão no lado direito de uma definição case o identificador da próxima definição. Sem essa restrição, no exemplo a seguir o identificador D seria parte da definição de A:

```
A -> B C          D -> 'd'
```

Continuando a definição de expressão primária, temos que uma expressão primária pode ser uma expressão entre parênteses, um literal, uma classe de caracteres, ou DOT, que casa qualquer caractere.

Dada essa visão geral da sintaxe de PEGs, vamos discutir agora um pouco mais da semântica de PEGs, e quais são as vantagens e desvantagens de PEGs em relação a CFGs e expressões regulares.

Como vimos no exemplo da figura 1.1, PEGs usam o símbolo / para representar uma escolha. ao invés do símbolo | usado em descrições EBNF de CFGs. Essa mudança busca enfatizar que em PEGs a ordem das alternativas de uma escolha é importante, pois tentamos casar a primeira alternativa de uma escolha antes da segunda. Se o casamento da primeira alternativa de uma escolha é bem sucedido, o casamento da segunda alternativa não é realizado. Em virtude disso, o parser obtido a partir de uma PEG faz menos backtracking do que o parser obtido a partir de uma CFG. Contudo, isso implica em um maior cuidado quando uma alternativa de uma escolha casa prefixos de outra alternativa. Na PEG da figura 1.1, esse é o caso da escolha na linha 35, no lado direito da definição de $EndOfLine$. Nessa escolha, temos que a alternativa `'\r'` casa um prefixo da alternativa `'\r\n'`, e por causa disso deve ser listada após esta alternativa.

Uma consequência da escolha ordenada é que em PEGs, ao contrário de expressões regulares, as repetições são possessivas. Isso quer dizer que uma repetição casa a maior porção possível da entrada, sem levar em conta a expressão que segue a repetição. Com o uso da escolha ordenada, podemos expressar a semântica usual de p^* através da regra abaixo, onde `''` é uma expressão que casa a cadeia vazia:

```
A -> p A / ''
```

A regra anterior nos diz que primeiro tentamos casar a expressão p, e só quando esse casamento não é bem sucedido é que casamos a cadeia vazia, cujo casamento sempre é bem sucedido. Assim, o casamento de uma expressão como $'a'*'a'$ nunca é bem sucedido, como mostramos a seguir. Essa expressão pode ser reescrita como abaixo:

```
A -> B 'a'          B -> 'a' B / ''
```

Dada a PEG anterior, para qualquer entrada da forma a^n, temos que todos os `a`s da entrada serão casados pela variável B, uma vez que a primeira alternativa da escolha ordenada de B sempre casa uma entrada que começa com `a`. Assim, a cadeia de entrada estará vazia ao tentarmos casar a expressão $'a'$ que segue a variável B, e portanto o casamento da variável A irá falhar.

Apesar de termos somente repetições possessivas em PEGs, é possível definir outros tipos de repetições, como a gulosa e a preguiçosa, através de construções apropriadas, como discutimos a seguir. Na discussão abaixo, vamos considerar que $p1$ é a expressão que está sendo repetida, e que $p2$ é a expressão que segue $p1$.

Para definir uma repetição gulosa, devemos construir uma PEG onde a expressão $p1$ casa o maior número possível de vezes, desde que a expressão $p2$ também case. A seguinte PEG define esse tipo de repetição:

```
A -> p1 A / p2
```

Já para definir uma repetição preguiçosa, devemos construir uma PEG onde a expressão $p1$ casa o menor número possível de vezes, desde que a expressão $p2$ também case. A PEG a seguir define essa repetição:

```
A -> p2 / p1 A
```

Podemos ver que a definição de repetição preguiçosa é bastante parecida com a definição de repetição gulosa. A única diferença entre as duas definições é a ordem das alternativas da escolha ordenada. Na repetição gulosa, tentamos casar primeiro a expressão $p1$ que está sendo repetida, enquanto que na

repetição preguiçosa tentamos casar primeiro a expressão $p2$ que segue a repetição.

Dado que em PEGs os operadores de repetição são possessivos, não precisamos de uma regra adicional de casamento mais longo, como a usada por analisadores léxicos baseados em expressões regulares, para descrever elementos léxicos tais como identificadores. Da mesma forma, como a escolha é ordenada em PEGs, não precisamos da regra adicional de definição mais acima usada por esses analisadores léxicos quando mais de uma definição de um elemento léxico casa uma dada entrada.

Como vimos anteriormente, PEGs possuem os operadores de predicado & e !. O casamento de uma expressão da forma $\&p$ é bem sucedido quando o casamento de p é bem sucedido, enquanto que o casamento de uma expressão da forma $!p$ é bem sucedido quando o casamento de p não é bem sucedido. O casamento de um predicado não consome caracteres da cadeia de entrada. Assim, os operadores & e ! oferecem um *lookahead* ilimitado, o que facilita a descrição de algumas linguagens.

Podemos ver o operador & como um açúcar sintático, uma vez que a expressão $\&p$ pode ser definida como $!!p$.

A seguir, mostramos o exemplo de uma PEG que usa predicados e descreve a linguagem $a^n b^n c^n$, que não é livre de contexto, onde $n \geq 1$:

```
S -> &(A 'c') 'a'* B !.
A -> 'a' A 'b' / 'ab'
B -> 'b' B 'c' / 'bc'
```

Na PEG anterior, ao definirmos S usamos o predicado $\&A$ para testar se o começo da cadeia de entrada possui n `a`s seguidos por n `b`s seguidos por um `c`. Em caso afirmativo, casamos os `a`s da entrada, e depois tentamos casar uma sequência de n `b`s seguidos por n `c`s. Por fim, usamos o predicado `!.` para testar se todos os caracteres da entrada foram consumidos, e portanto todos os `c`s foram casados.

Devido principalmente ao uso de predicados e de repetições possessivas, PEGs permitem descrever com a mesma facilidade os elementos léxicos e sintáticos de uma linguagem, como vimos no exemplo da figura 1.1. No caso de CFGs, embora teoricamente seja possível descrever os elementos léxicos de uma linguagem, essa não é uma abordagem prática, como mostrado por Ford [Ford, 2002].

Por fim, uma vantagem de PEGs em relação a CFGs é a facilidade para obtermos um parser a partir de uma gramática, pois ao descrevermos uma linguagem através de PEGs estamos também descrevendo um parser para a

mesma. No caso de CFGs, dada uma CFG G, nem sempre é possível obter um parser para a linguagem de G através de ferramentas como Yacc [Levine et al., 1992] e ANTLR [Parr e Quong, 1995, Parr, 2007], uma vez que G pode não pertencer à classe de gramáticas para a qual a ferramenta oferece suporte, como $LALR(1)$, $LL(1)$, ou $LL(k)$-forte. Em resumo, temos que é trivial obter um parser a partir de uma PEG, ao passo que pode ser uma tarefa difícil obter um parser a partir de uma CFG.

1.2 Organização da Tese

O próximo capítulo mostra a formalização original de Gramáticas de Expressões de Parsing e apresenta uma nova formalização de PEGs usando semântica natural. No capítulo 3, discutimos a correspondência entre expressões regulares e PEGs, apresentamos uma formalização de expressão regulares usando semântica natural, e mostramos uma transformação para obter uma PEG equivalente a partir de uma dada expressão regular. No capítulo 4, apresentamos uma nova formalização de CFGs, baseada em semântica natural, e discutimos a correspondência entre PEGs e CFGs lineares à direita, $LL(1)$ e $LL(k)$-forte. Por fim, no capítulo 5, discutimos alguns trabalhos relacionados e apresentamos as nossas conclusões.