Pontifícia Universidade Católica
do Rio de Janeiro

Sérgio Queiroz de Medeiros

Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto

Tese de Doutorado

Tese apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio como requisito parcial para obtenção do título de Doutor em Informática

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Agosto de 2010

Pontifícia Universidade Católica
do Rio de Janeiro

Sérgio Queiroz de Medeiros

Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto

Tese apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do título de Doutor em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática — PUC–Rio

Prof. Edward Hermann Haeusler
Departamento de Informática — PUC–Rio

Prof. Noemi de La Rocque Rodriguez
Departamento de Informática — PUC–Rio

Prof. Alex de Vasconcellos Garcia
Instituto Militar de Engenharia — IME

Prof. Anamaria Martins Moreira
Universidade Federal do Rio Grande do Norte — UFRN

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 18 de Agosto de 2010

Sérgio Queiroz de Medeiros

Graduou-se em Ciência da Computação na Universidade Federal do Rio Grande Norte (UFRN), onde também fez mestrado em Sistemas e Computação. Durante o doutorado, na PUC-Rio, trabalhou na área de Linguagens de Programação.

Ficha Catalográfica

Medeiros, Sérgio Queiroz de

Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto / Sérgio Queiroz de Medeiros; orientador: Roberto Ierusalimschy. — 2010.

86 f. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2010.

Inclui bibliografia.

1. Informática – Teses. 2. Gramáticas de Expressões de Parsing. 3. Expressões Regulares. 4. Gramáticas Livres de Contexto. 5. Semântica Natural. 6. $LL(1)$. 7. $LL(k)$-Forte. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Primeiro, gostaria de agradecer a Roberto pelos anos de orientação e por me motivar a tentar fazer um trabalho melhor.

Agradeço também a todo mundo do LabLua. Em especial, gostaria de agradecer a Fabio que esteve sempre por aqui e ajudou em vários momentos da pesquisa.

Gostaria de agradecer aos meu pais e às minhas irmãs, que mesmo longe estão comigo.

Agradeço a todos os companheiros de república, em especial a Bruno.

Gostaria de agradecer a todos os amigos, aos antigos e aos que fiz nesses anos de Rio.

Agradeço a Marcia pelo apoio, pelo carinho e pelo sorriso.

Finalmente, gostaria de agradecer ao CNPq, ao Tecgraf e à PUC–Rio pela ajuda financeira e pelo bom ambiente de trabalho.

Resumo

Medeiros, Sérgio Queiroz de; Ierusalimschy, Roberto. **Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto**. Rio de Janeiro, 2010. 86p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Gramáticas de Expressões de Parsing (PEGs) são um formalismo que permite descrever linguagens e que possui como característica distintiva o uso de um operador de escolha ordenada. A classe de linguagens descrita por PEGs contém propriamente todas as linguagens livres de contexto determinísticas. Nesta tese discutimos a correspondência de PEGs com dois outros formalismos usados para descrever linguagens: expressões regulares e Gramáticas Livres de Contexto (CFGs). Apresentamos uma formalização de expressões regulares usando semântica natural e mostramos uma transformação para converter expressões regulares em PEGs que descrevem a mesma linguagem; essa transformação pode ser facilmente adaptada para acomodar diversas extensões usadas por bibliotecas de expressões regulares (e.g., repetição preguiçosa e subpadrões independentes). Também apresentamos uma nova formalização de CFGs usando semântica natural e mostramos a correspondência entre CFGs lineares à direita e PEGs equivalentes. Além disso, mostramos que gramáticas $LL(1)$ com uma pequena restrição descrevem a mesma linguagem quando interpretadas como CFGs e quando interpretadas como PEGs. Por fim, mostramos como transformar CFGs $LL(k)$-forte em PEGs equivalentes.

Palavras–chave

Gramáticas de Expressões de Parsing; Expressões Regulares; Gramáticas Livres de Contexto; Semântica Natural; $LL(1)$; $LL(k)$-Forte.

Abstract

Medeiros, Sérgio Queiroz de; Ierusalimschy, Roberto. **Correspondence between PEGs and Classes of Context-Free Grammars**. Rio de Janeiro, 2010. 86p. DSc Thesis — Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Parsing Expression Grammars (PEGs) are a formalism that allow us to describe languages and that has as its distinguishing feature the use of an ordered choice operator. The class of languages described by PEGs properly contains all deterministic context-free languages. In this thesis we discuss the correspondence between PEGs and two other formalisms used to describe languages: regular expressions and Context-Free Grammars (CFGs). We present a new formalization of regular expressions that uses natural semantics and we show a transformation to convert a regular expression into a PEG that describes the same language; this transformation can be easily adapted to accommodate several extensions used by regular expression libraries (e.g., lazy repetition and independent subpatterns). We also present a new formalization of CFGs that uses natural semantics and we show the correspondence between right linear CFGs and equivalent PEGs. Moreover, we show that $LL(1)$ grammars with a minor restriction define the same language when interpreted as a CFG and when interpreted as a PEG. Finally, we show how to transform strong-$LL(k)$ CFGs into PEGs that are equivalent.

Keywords

Parsing Expression Grammars; Regular Expressions; Context-Free Grammars; Natural Semantics; $LL(1)$; Strong-$LL(k)$.

Sumário

Lista de figuras

Lista de tabelas

Soneto do desmantelo azul

Então, pintei de azul os meus sapatos
por não poder de azul pintar as ruas,
depois, vesti meus gestos insensatos
e colori as minhas mãos e as tuas.

Para extinguir em nós o azul ausente
e aprisionar no azul as coisas gratas,
enfim, nós derramamos simplesmente
azul sobre os vestidos e as gravatas.

E afogados em nós, nem nos lembramos
que no excesso que havia em nosso espaço
pudesse haver de azul também cansaço.

E perdidos de azul nos contemplamos
e vimos que entre nós nascia um sul
vertiginosamente azul. Azul.

Carlos Pena Filho, *Livro Geral.*

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.

2
Gramáticas de Expressões de Parsing

Neste capítulo vamos rever a definição de Gramáticas de Expressões de Parsing (*Parsing Expression Grammars* — PEGs). Além disso, apresentaremos uma nova formalização de PEGs, baseada em semântica natural, que usaremos ao longo da tese.

Na seção 2.1 mostramos a formalização original de PEGs dada por Ford, e na seção 2.2 mostramos a relação usada por Ford para interpretar PEGs. Finalmente, na seção 2.3, apresentamos a nossa formalização de PEGs baseada em semântica natural.

2.1
Definição de PEGs

Segundo Ford [Ford, 2004], uma PEG é uma tupla $G = (V, T, P, p_S)$, onde V é um conjunto finito de não terminais (também chamados de variáveis), T é um conjunto finito de terminais, P é uma função de não terminais em expressões de parsing, e p_S é a expressão de parsing inicial.

Ao longo do texto, iremos usar também o termo *gramática* ao nos referirmos a uma PEG.

Uma expressão de parsing p pode ser definida indutivamente como mostrado a seguir, onde $a \in T$, $A \in V$, e p_1 e p_2 também são expressões de parsing:

$$p \;=\; \varepsilon \;\mid\; a \;\mid\; A \;\mid\; p_1\, p_2 \;\mid\; p_1 \,/\, p_2 \;\mid\; p_1^* \;\mid\; !p_1$$

Na definição acima, temos que ε é uma expressão que casa a cadeia vazia, a é um terminal, A é um não terminal, $p_1\, p_2$ é uma concatenação, $p_1 \,/\, p_2$ é uma escolha ordenada, p_1^* é uma repetição, e $!p_1$ é um predicado de negação.

A prioridade do operador * é a mais alta, seguida pela prioridade do operador !. A seguir, temos a prioridade da concatenação, e por último temos a prioridade da escolha ordenada. A concatenação e a escolha são operações associativas, de modo que podemos interpretar a escolha $p_1 \,/\, p_2 \,/\, p_3$ tanto como $p_1 \,/\, (p_2 \,/\, p_3)$ quanto como $(p_1 \,/\, p_2) \,/\, p_3$ sem alterar o seu resultado. Por motivos que veremos mais adiante, iremos considerar uma associatividade mais à direita para a concatenação e para a escolha.

Seguindo Ford [Ford, 2004], esse conjunto básico de expressões de parsing que apresentamos não contém algumas expressões que vimos na figura 1.1, tais como $\mathbf{.}$, $p?$, $p+$, e $\&p$, pois estas expressões podem ser vistas como um açúcar sintático. A seguir discutimos como podemos reescrever alguns dos açúcares sintáticos que aparecem na figura 1.1.

A expressão de parsing $\mathbf{.}$ casa qualquer terminal da entrada e pode ser reescrita como uma escolha ordenada de todos os elementos do conjunto de terminais.

Uma expressão da forma $\texttt{"}a_1 a_2 \cdots a_n\texttt{"}$, que casa cada terminal a_i em sequência, pode ser reescrita como uma concatenação $a_1\, a_2\, \cdots\, a_n$, enquanto que uma classe de caracteres $[a_1 a_2 \cdots a_n]$, que casa algum caractere a_i, pode ser reescrita como uma escolha ordenada $a_1 \,/\, a_2 \,/\, \cdots \,/\, a_n$.

A expressão de parsing $p?$ representa um casamento opcional e pode ser reescrita como $p \,/\, \varepsilon$. Já a expressão $p+$, que casa p uma ou mais vezes, pode ser reescrita como $p\, p^*$.

Por fim, a expressão de parsing $\&p$ é um predicado que casa quando p casa e falha quando o casamento de p falha. A expressão $\&p$ pode ser reescrita como $!!p$.

Ao longo do texto, usaremos G para representar uma gramática e p para representar uma expressão de parsing. Letras minúsculas do início do alfabeto, como a e b, serão usadas para representar terminais; letras minúsculas do final do alfabeto, como x e y, para representar cadeias (possivelmente vazias) de terminais; e letras maiúsculas do início do alfabeto, como A e B, para representar não terminais.

Dada uma PEG $G = (V, T, P, p_S)$, usaremos o termo produção ao nos referirmos a um par $(A, p) \in P$. Iremos representar uma produção (A, p) como $A \rightarrow p$.

Dada uma gramática $G = (V, T, P, p_S)$, usaremos a notação $G[p'_S]$ para representar uma nova gramática $G' = (V, T, P, p'_S)$, ou seja, G' é uma gramática com o mesmo conjunto de terminais, não terminais e produções de G, mas que possui p'_S como expressão de parsing inicial.

2.2 Interpretação de PEGs Usando a Relação $\Rightarrow_G$

Na formalização original de PEGs [Ford, 2004], o significado de uma gramática G é dado pela relação $\Rightarrow_G$, que é baseada no trabalho anterior de Birman [Birman, 1970, Birman e Ullman, 1973]. Dada uma gramática $G = (V, T, P, p_S)$, a relação $\Rightarrow_G$ associa pares da forma (p, x) com pares da forma (n, o), onde p é uma expressão de parsing, x é uma cadeia de entrada,

$n \geq 0$ é um contador de passos, e $o \in (T^* \cup \{\texttt{fail}\})$ representa o resultado de um casamento. Quando um casamento é bem sucedido, a saída o é um prefixo da entrada x. Caso contrário, a saída é `fail`.

Ford define a relação $\Rightarrow_G$ indutivamente da seguinte maneira:

1. **Cadeia Vazia:** $(\varepsilon,\, x) \Rightarrow_G (1,\, \varepsilon)$, para qualquer x.
2. **Terminal (caso de sucesso):** $(a,\, ax) \Rightarrow_G (1,\, a)$.
3. **Terminal (caso de falha):** $(a,\, bx) \Rightarrow_G (1,\, \texttt{fail})$ se $a \neq b$, e $(a,\, \varepsilon) \Rightarrow_G (1,\, \texttt{fail})$.
4. **Variável:** $(A,\, x) \Rightarrow_G (n+1,\, o)$ se $A \rightarrow p \in P$ e $(p,\, x) \Rightarrow_G (n,\, o)$.
5. **Concatenação (caso de sucesso):** se $(p_1,\, xyz) \Rightarrow_G (n_1,\, x)$ e $(p_2,\, yz) \Rightarrow_G (n_2,\, y)$, então $(p_1\, p_2,\, xyz) \Rightarrow_G (n_1 + n_2 + 1,\, xy)$.
6. **Concatenação (caso de falha 1):** se $(p_1,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$, então $(p_1\, p_2,\, x) \Rightarrow_G (n_1 + 1,\, \texttt{fail})$.
7. **Concatenação (caso de falha 2):** se $(p_1,\, xy) \Rightarrow_G (n_1,\, x)$ e $(p_2,\, y) \Rightarrow_G (n_2,\, \texttt{fail})$, então $(p_1\, p_2,\, xy) \Rightarrow_G (n_1 + n_2 + 1,\, \texttt{fail})$.
8. **Escolha Ordenada (caso 1):** se $(p_1,\, xy) \Rightarrow_G (n_1,\, x)$, então $(p_1 \,/\, p_2,\, xy) \Rightarrow_G (n_1 + 1,\, x)$.
9. **Escolha Ordenada (caso 2):** se $(p_1,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$ e $(p_2,\, x) \Rightarrow_G (n_2,\, o)$, então $(p_1 \,/\, p_2,\, x) \Rightarrow_G (n_1 + n_2 + 1,\, o)$.
10. **Repetição (caso de repetição):** se $(p,\, xyz) \Rightarrow_G (n_1,\, x)$ e $(p^*,\, yz) \Rightarrow_G (n_2,\, y)$, então $(p^*,\, xyz) \Rightarrow_G (n_1 + n_2 + 1,\, xy)$.
11. **Repetição (caso de terminação):** se $(p,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$, então $(p^*,\, x) \Rightarrow_G (n_1 + 1,\, \varepsilon)$.
12. **Predicado de Negação (caso 1):** se $(p,\, xy) \Rightarrow_G (n,\, x)$, então $(!p,\, xy) \Rightarrow_G (n+1,\, \texttt{fail})$.
13. **Predicado de Negação (caso 2):** se $(p,\, x) \Rightarrow_G (n,\, \texttt{fail})$, então $(!p,\, x) \Rightarrow_G (n+1,\, \varepsilon)$.

A relação $\Rightarrow_G$ é então usada para definir uma nova relação $\Rightarrow_G^+$, que associa pares da forma $(p,\, x)$ com uma saída o, onde a definição de $\Rightarrow_G^+$ é dada a seguir:

$(p,\, x) \Rightarrow_G^+ o$ se e somente se existe um n tal que $(p,\, x) \Rightarrow_G (n,\, o)$

De acordo com Ford, podemos definir a linguagem de uma PEG da seguinte maneira:

Definição 2.2.1. *Dada uma PEG* $G = (V, T, P, p_S)$*, a sua linguagem consiste das cadeias* xy *tais que* $(p_S, xy) \Rightarrow_G^+ x$.

Ao longo do texto, usaremos $L(G)$ para representar a linguagem de uma PEG G.

Pela definição anterior, dada uma PEG G, se o casamento de w é bem sucedido em G, então $w \in L(G)$, mesmo que somente um prefixo de w tenha sido consumido nesse casamento.

2.3 Formalização de PEGs Usando Semântica Natural

Ao contrário de Ford, que usou as relações $\Rightarrow_G$ e $\Rightarrow_G^+$, usaremos a relação $\overset{\text{PEG}}{\rightsquigarrow}$ para dar significado a uma PEG. A definição de $\overset{\text{PEG}}{\rightsquigarrow}$ usa semântica natural, que é mais adequada para estudar a correspondência de PEGs com expressões regulares e CFGs pois torna mais explícitas as diferenças e semelhanças entre esses formalismos. Nos próximos capítulos da tese, iremos apresentar formalizações de expressões regulares e CFGs que também usam semântica natural e iremos estabelecer a correspondência de PEGs com expressões regulares e CFGs $LL(k)$-forte.

Definimos $\overset{\text{PEG}}{\rightsquigarrow}$ como uma relação $(G \times T^*) \times (T^* \cup \{\texttt{fail}\})$, onde $\overset{\text{PEG}}{\rightsquigarrow}$ relaciona uma gramática G e uma entrada xy ou com um sufixo y da entrada ou com `fail`. Usamos a notação $G\ xy \overset{\text{PEG}}{\rightsquigarrow} X$, onde $X \in (T^* \cup \{\texttt{fail}\})$, para indicar que $((G, xy), X) \in \overset{\text{PEG}}{\rightsquigarrow}$. A figura 2.1 apresenta a definição da relação $\overset{\text{PEG}}{\rightsquigarrow}$ usando semântica natural.

Uma diferença entre as relações $\Rightarrow_G$ e $\overset{\text{PEG}}{\rightsquigarrow}$ é que a relação $\Rightarrow_G$ nos dá como resultado de um casamento bem sucedido um prefixo da entrada e um contador de passos, enquanto que a relação $\overset{\text{PEG}}{\rightsquigarrow}$ nos dá um sufixo da entrada e uma árvore de prova. O fato da relação $\overset{\text{PEG}}{\rightsquigarrow}$ nos dar um prefixo da entrada ao invés de um sufixo não é relevante, pois ela poderia ser facilmente modificada para nos dar um sufixo da entrada. Já a ausência do contador de passos em $\overset{\text{PEG}}{\rightsquigarrow}$ se deve ao fato de que a altura da árvore de prova dada por essa relação pode ser usada como uma medida da complexidade de um casamento, o que torna o contador de passos desnecessário.

A seguir, discutimos as regras de $\overset{\text{PEG}}{\rightsquigarrow}$ e a sua correspondência com as regras da relação $\Rightarrow_G$.

A regra *empty*.1 trata do caso em que a expressão de parsing inicial da gramática representa a cadeia vazia. O casamento dessa expressão de parsing

Cadeia Vazia $\dfrac{}{G[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\,(\textbf{empty.1})$ **Variável** $\dfrac{G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} X}{G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} X}\,(\textbf{var.1})$

Terminal $\dfrac{}{G[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x}\,(\textbf{char.1})$ $\dfrac{}{G[b]\ ax \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\,, b \neq a\,(\textbf{char.2})$ $\dfrac{}{G[a]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\,(\textbf{char.3})$

Concatenação $\dfrac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y \quad G[p_2]\ y \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\,p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} X}\,(\textbf{con.1})$ $\dfrac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p_1\,p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\,(\textbf{con.2})$

Escolha Ordenada $\dfrac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[p_1\,/\,p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}\,(\textbf{ord.1})$ $\dfrac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail} \quad G[p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\,/\,p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X}\,(\textbf{ord.2})$

Predicado de Negação $\dfrac{G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[!p]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\,(\textbf{not.1})$ $\dfrac{G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[!p]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\,(\textbf{not.2})$

Repetição $\dfrac{G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p^*]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\,(\textbf{rep.1})$ $\dfrac{G[p]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz \quad G[p^*]\ yz \overset{\text{PEG}}{\rightsquigarrow} z}{G[p^*]\ xyz \overset{\text{PEG}}{\rightsquigarrow} z}\,(\textbf{rep.2})$

Figura 2.1: Definição da Relação $\overset{\text{PEG}}{\rightsquigarrow}$ Usando Semântica Natural

não consome nenhum terminal da entrada e sempre é bem sucedido. Essa regra corresponde à regra 1 da semântica de $\Rightarrow_G$.

As regras *char*.1, *char*.2 e *char*.3 tratam do caso em que a expressão de parsing inicial de G é um terminal. A regra *char*.1 corresponde à regra 2 da semântica de $\Rightarrow_G$, e as regras *char*.2 e *char*.3 correspondem à regra 3 de $\Rightarrow_G$.

Quando discutirmos a correspondência de PEGs com expressões regulares e CFGs lineares à direita usaremos **.** para representar uma expressão de parsing que casa qualquer terminal.

A regra *var*.1 trata do caso em que a expressão de parsing inicial é um não terminal A. Essa regra faz com que a expressão de parsing associada com A seja usada como expressão de parsing inicial da gramática. O resultado do casamento dessa expressão de parsing é então o resultado do casamento de A. A regra correspondente a *var*.1 na semântica de $\Rightarrow_G$ é a regra 4.

As regras *con*.1 e *con*.2 tratam da concatenação de expressões de parsing. No caso de uma concatenação $p_1\,p_2$, se o casamento de p_1 falha, então o casamento da concatenação falha (regra *con*.2). Se o casamento de p_1 é bem sucedido, o resultado da concatenação é o resultado do casamento de p_2 (regra *con*.1). A regra *con*.1 corresponde às regras 5 e 7 da semântica de Ford, e a regra *con*.2 corresponde à regra 6 da semântica de Ford.

$$\dfrac{\dfrac{\dfrac{}{G[a]\ bcd \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}(\textbf{char.2}) \quad \dfrac{}{G[b]\ bcd \overset{\text{PEG}}{\rightsquigarrow} cd}(\textbf{char.1})}{G[a\ /\ b]\ bcd \overset{\text{PEG}}{\rightsquigarrow} cd}(\textbf{ord.2}) \quad \dfrac{}{G[c]\ cd \overset{\text{PEG}}{\rightsquigarrow} d}(\textbf{char.1})}{G[(a\ /\ b)\ c]\ bcd \overset{\text{PEG}}{\rightsquigarrow} d}(\textbf{con.1})$$

Figura 2.2: Exemplo de Árvore de Prova Usando a Relação $\overset{\text{PEG}}{\rightsquigarrow}$

As regras *ord*.1 e *ord*.2 tratam da escolha ordenada. No caso de uma escolha ordenada $p_1 \,/\, p_2$, se o casamento de p_1 é bem sucedido, então o casamento da escolha ordenada é bem sucedido (regra *ord*.1). Se o casamento da alternativa p_1 falha, então o resultado da escolha ordenada é o resultado do casamento da alternativa p_2 (regra *ord*.2). Temos que *ord*.1 é a regra correspondente da regra 8 de $\Rightarrow_G$, e que *ord*.2 corresponde à regra 9 de $\Rightarrow_G$.

As regras *not*.1 e *not*.2 tratam do predicado de negação. Se o casamento de uma expressão de parsing p falha, então o casamento de $!p$ é bem sucedido (regra *not*.1). De modo análogo, se o casamento de p é bem sucedido, então o casamento de $!p$ falha (regra *not*.2). O casamento de $!p$ não consome nenhum terminal da entrada. A regra *not*.1 corresponde à regra 13 de $\Rightarrow_G$, enquanto que *not*.2 corresponde à regra 12 de $\Rightarrow_G$.

Na seção 4.6.1, ao discutir a correspondência entre CFGs $LL(k)$-forte e PEGs, usaremos $\&p$ como um açúcar sintático para $!!p$. O casamento da expressão de parsing $\&p$ é bem sucedido quando o casamento de p é bem sucedido e falha quando o casamento de p falha. O casamento de $\&p$ não consome nenhum prefixo da entrada.

Por fim, as regras *rep*.1 e *rep*.2 tratam da repetição de uma expressão de parsing p. A regra *rep*.1 trata do caso em que o casamento de p falha. Nesse caso, o resultado do casamento de p^* é a entrada corrente. A regra *rep*.2 trata do caso em que p casa um prefixo da entrada. Nesse caso, o resultado do casamento de p^* é o resultado do casamento de p^* levando-se em conta o restante da entrada. A regra correspondente a *rep*.1 em $\Rightarrow_G$ é a regra 11, enquanto que a regra 10 de $\Rightarrow_G$ corresponde à regra *rep*.2.

Na figura 2.2 podemos ver o exemplo de uma árvore de prova usando as regras de relação $\overset{\text{PEG}}{\rightsquigarrow}$. Nesse exemplo, a cadeia de entrada é `bcd`, e a expressão de parsing inicial da gramática é $(a\ /\ b)\ c$. O casamento dessa expressão de parsing é bem sucedido para a entrada `bcd`, e o prefixo `bc` dessa entrada é casado.

No exemplo da figura 2.2, a gramática G não é relevante para o resultado do casamento, uma vez que a expressão de parsing inicial não possui não terminais. Quando uma expressão de parsing p não possui não terminais,

podemos omitir a gramática e representar o casamento de p simplesmente como $p\ xy \overset{\text{PEG}}{\rightsquigarrow} X$, dado que o resultado desse casamento não depende da gramática.

Dadas as definições das relações $\Rightarrow_G$ e $\overset{\text{PEG}}{\rightsquigarrow}$, podemos ver que todas as regras de $\Rightarrow_G$ possuem regras correspondentes em $\overset{\text{PEG}}{\rightsquigarrow}$ e vice versa. É possível então estabelecer uma correspondência entre um casamento em $\Rightarrow_G$ e um casamento em $\overset{\text{PEG}}{\rightsquigarrow}$, como dito a seguir:

Lema 2.3.1. *Dada uma PEG G e uma expressão de parsing p, temos que $\exists\, n \cdot (p,\, xy) \Rightarrow_G (n,\, o)$ se e somente se $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} X$, onde $o = x \Leftrightarrow X = y$ e $o = \texttt{fail} \Leftrightarrow X = \texttt{fail}$.*

Demonstração. ($\Rightarrow$): A prova desta parte é por indução no número n dado pela relação $\Rightarrow_G$. Vamos estruturar nossa prova com base nas possíveis regras da relação $\Rightarrow_G$ que podem ter sido usadas.

Se a regra 1 foi usada, então $(\varepsilon,\, x) \Rightarrow_G (1,\, \varepsilon)$, e por *empty*.1 temos que $G[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x$.

Se a regra 2 foi usada, então $(a,\, ax) \Rightarrow_G (1,\, a)$, e por *char*.1 temos que $G[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x$.

Se a regra 3 foi usada, há dois subcasos dependendo se a entrada é ou não vazia. No primeiro subcaso, o casamento foi $(a,\, bx) \Rightarrow_G (1,\, \texttt{fail})$, onde $a \neq b$, e por *char*.2 concluímos que $G[a]\ bx \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. No segundo subcaso, o casamento foi $(a,\, \varepsilon) \Rightarrow_G (1,\, \texttt{fail})$, e por *char*.3 concluímos que $G[a]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

Se a regra 4 foi usada, temos que $(P(A),\, x) \Rightarrow_G (n,\, o)$. Pela hipótese de indução temos que $G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} X$, e por *var*.1 concluímos que $G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} X$.

Se a regra 5 foi usada, temos que $(p_1,\, xyz) \Rightarrow_G (n_1,\, x)$ e que $(p_2,\, yz) \Rightarrow_G (n_2,\, y)$. Pela hipótese de indução temos que $G[p_1]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz$ e que $G[p_2]\ yz \overset{\text{PEG}}{\rightsquigarrow} z$, e por *con*.1 concluímos que $G[p_1\, p_2]\ xyz \overset{\text{PEG}}{\rightsquigarrow} z$.

Se a regra 6 foi usada, temos que $(p_1,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$. Pela hipótese de indução temos que $G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, e por *con*.2 concluímos que $G[p_1\, p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

Se a regra 7 foi usada, temos que $(p_1,\, xy) \Rightarrow_G (n_1,\, x)$ e que $(p_2,\, y) \Rightarrow_G (n_2,\, \texttt{fail})$. Pela hipótese de indução temos que $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ e que $G[p_2]\ y \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Assim, por *con*.2 concluímos que $G[p_1\, p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

Se a regra 8 foi usada, temos que $(p_1,\, xy) \Rightarrow_G (n_1,\, x)$. Pela hipótese de indução temos que $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e por *ord*.1 concluímos que $G[p_1\, /\, p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$.

Se a regra 9 foi usada, temos que $(p_1,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$ e que $(p_2,\, x) \Rightarrow_G (n_2,\, o)$. Pela hipótese de indução temos que $G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ e que $G[p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X$, e por *ord*.2 concluímos que $G[p_1\, /\, p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X$.

Se a regra 10 foi usada, temos que $(p,\, xyz) \Rightarrow_G (n_1,\, x)$ e que $(p^*,\, yz) \Rightarrow_G (n_2,\, y)$. Pela hipótese de indução temos que $G[p]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz$ e que $G[p^*]\ yz \overset{\text{PEG}}{\rightsquigarrow} z$. Assim, por *rep*.2 concluímos que $G[p^*]\ xyz \overset{\text{PEG}}{\rightsquigarrow} z$.

Se a regra 11 foi usada, temos que $(p,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$. Pela hipótese de indução temos que $G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, e por *rep*.1 concluímos que $G[p^*]\ x \overset{\text{PEG}}{\rightsquigarrow} x$.

Se a regra 12 foi usada, temos que $(p,\, xy) \Rightarrow_G (n,\, x)$. Pela hipótese de indução temos que $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e por *not*.2 concluímos que $G[!p]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

Se a regra 13 foi usada, temos que $(p,\, x) \Rightarrow_G (n,\, \texttt{fail})$. Pela hipótese de indução temos que $G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, e por *not*.1 concluímos que $G[!p]\ x \overset{\text{PEG}}{\rightsquigarrow} x$.

($\Leftarrow$): A prova desta parte é por indução na altura da árvore de prova dada pela relação $\overset{\text{PEG}}{\rightsquigarrow}$. Vamos estruturar nossa prova com base nas possíveis regras da relação $\overset{\text{PEG}}{\rightsquigarrow}$ que podem ter sido usadas para concluir essa árvore de prova.

Se a regra *empty*.1 foi usada, então $p = \varepsilon$. Assim, pela regra 1 temos que $(\varepsilon,\, x) \Rightarrow_G (1,\, \varepsilon)$.

Se a regra *char*.1 foi usada, então $p = a$ e a entrada é da forma ax. Assim, pela regra 2 temos que $(a,\, ax) \Rightarrow_G (1,\, a)$.

Se a regra *char*.2 foi usada, então $p = a$ e a entrada é forma bx, onde $a \neq b$. Assim, pela regra 3 temos que $G[a]\ bx \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

Se a regra *char*.3 foi usada, então $p = a$ e a entrada é da forma ε. Assim, pela regra 3 temos que $(a,\, \varepsilon) \Rightarrow_G (1,\, \texttt{fail})$.

Se a regra *var*.1 foi usada, temos que $G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} X$. Pela hipótese de indução temos que $(P(A),\, x) \Rightarrow_G (n,\, o)$, e pela regra 4 concluímos que $(A,\, x) \Rightarrow_G (n+1,\, o)$.

Se a regra *con*.1 foi usada, temos que $G[p_1]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz$ e que $G[p_2]\ yz \overset{\text{PEG}}{\rightsquigarrow} X$. Pela hipótese de indução temos que $(p_1,\, xyz) \Rightarrow_G (n_1,\, x)$ e que $(p_2,\, yz) \Rightarrow_G (n_2,\, o)$. Há dois subcasos dependendo se o casamento de p_2 foi ou não bem sucedido. No primeiro subcaso, pela regra 5 concluímos que $(p_1\, p_2,\, xyz) \Rightarrow_G (n_1+n_2+1,\, xy)$. No segundo subcaso, pela regra 7 concluímos que $(p_1\, p_2,\, x) \Rightarrow_G (n_1 + n_2 + 1,\, \texttt{fail})$.

Se a regra *con*.2 foi usada, temos que $G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Pela hipótese de indução temos que $(p_1,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$, e pela regra 6 concluímos que $(p_1\, p_2,\, x) \Rightarrow_G (n_1 + 1,\, \texttt{fail})$.

Se a regra *ord*.1 foi usada, temos que $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Pela hipótese de indução temos que $(p_1,\, xy) \Rightarrow_G (n_1,\, x)$, e pela regra 8 concluímos que $(p_1\, /\, p_2,\, x) \Rightarrow_G (n_1 + 1,\, x)$.

Se a regra *ord*.2 foi usada, temos que $G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ e que $G[p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X$. Pela hipótese de indução temos que $(p_1,\, x) \Rightarrow_G (n_1,\, \texttt{fail})$

e que $(p_2, x) \Rightarrow_G (n_2, o)$, e pela regra 9 concluímos que $(p_1 \,/\, p_2, x) \Rightarrow_G (n_1 + n_2 + 1, o)$.

Se a regra *rep*.1 foi usada, temos que $G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Pela hipótese de indução temos que $(p, x) \Rightarrow_G (n_1, \texttt{fail})$, e pela regra 11 concluímos que $(p^*, x) \Rightarrow_G (n_1 + 1, \varepsilon)$.

Se a regra *rep*.2 foi usada, temos que $G[p]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz$ e que $G[p^*]\ yz \overset{\text{PEG}}{\rightsquigarrow} z$. Pela hipótese de indução sabemos que $(p, xyz) \Rightarrow_G (n_1, x)$ e que $(p^*, yz) \Rightarrow_G (n_2, y)$, e pela regra 10 concluímos que $(p^*, xyz) \Rightarrow_G (n_1 + n_2 + 1, xy)$.

Se a regra *not*.1 foi usada, temos que $G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Pela hipótese de indução temos que $(p, x) \Rightarrow_G (n, \texttt{fail})$, e pela regra 13 concluímos que $(!p, x) \Rightarrow_G (n + 1, \varepsilon)$.

Finalmente, se a regra *not*.2 foi usada, temos que $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Pela hipótese de indução temos que $(p, xy) \Rightarrow_G (n, x)$, e pela regra 12 concluímos que $(!p, xy) \Rightarrow_G (n + 1, \texttt{fail})$. □

Após provar a correspondência entre as relações $\Rightarrow_G$ e $\overset{\text{PEG}}{\rightsquigarrow}$, vamos definir alguns conceitos de PEGs que são adaptações das definições dadas por Ford [Ford, 2004] e enunciar alguns lemas.

Primeiro, vamos definir os conceitos de gramática *livre de predicado* e *livre de repetição*.

Definição 2.3.1. *Uma gramática é livre de repetição se ela não possui nenhuma expressão de parsing* p^*.

Definição 2.3.2. *Uma gramática é livre de predicado se ela não possui nenhuma expressão de parsing* $!p$.

Agora, vamos definir quando uma escolha ordenada é disjunta:

Definição 2.3.3. *Dada uma PEG G, uma escolha* $p_1 \,/\, p_2$ *de G é disjunta se temos que* $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y \Rightarrow G[p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, *e que* $G[p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y \Rightarrow G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

Como podemos ver na definição acima, quando uma escolha é disjunta o casamento de uma alternativa é bem sucedido para uma dada entrada somente quando o casamento da outra alternativa não é bem sucedido para essa entrada. Se uma escolha é disjunta, a ordem das alternativas não é relevante.

A seguir, apresentamos a nossa definição da linguagem de uma PEG:

Definição 2.3.4. *Dada uma PEG G, temos que uma cadeia* $x \in L(G)$ *se e somente se existe uma cadeia y tal que* $G\ xy \overset{\text{PEG}}{\rightsquigarrow} y$.

Dada uma PEG G, seja $L(G)$ a nossa definição da linguagem de G e seja $L^F(G)$ a definição correspondente dada por Ford, temos que as cadeias de $L(G)$ são prefixos das cadeias de $L^F(G)$.

Como podemos ver, a nossa definição de linguagem enfatiza o prefixo x da entrada que foi casado, enquanto que a definição de Ford dá uma importância maior ao sufixo y da entrada que não foi casado.

Geralmente, se mudarmos o sufixo y da entrada que não foi casado, podemos casar um prefixo diferente da entrada ou obtermos um casamento que não é bem sucedido. Por exemplo, dada uma PEG G cuja expressão de parsing inicial é $a\,!b$, temos que essa expressão casa apenas o primeiro terminal da entrada, mas o segundo terminal da entrada também é importante para determinar o resultado do seu casamento. Segundo a nossa definição, $L(G)$ seria apenas $\{\texttt{a}\}$, ao passo que $L^F(G)$ teria, além de `a`, cadeias como `aa` e `acaca`.

Há casos em que o sufixo y não é importante para determinar o resultado de um casamento. Por exemplo, dada uma PEG G cuja expressão de parsing inicial é a escolha disjunta $a \,/\, b$, podemos ver que apenas o primeiro terminal da entrada é importante para determinar o resultado do casamento dessa expressão. Nesse caso, temos que $L(G) = \{\texttt{a}, \texttt{b}\}$, enquanto que $L^F(G)$ teria cadeias como `acc` e `bcc`, onde o sufixo `cc` não é importante para determinar o resultado do casamento dessas cadeias.

A nossa definição da linguagem descrita por uma PEG é mais próxima da definição de linguagem que usaremos para expressões regulares e CFGs. No caso destes formalismos, o sufixo y não é importante para determinar o resultado de um casamento. Como veremos, quando uma CFG ou expressão regular casa um prefixo x de uma entrada xy, isso implica que para qualquer sufixo y' é possível casar o prefixo x de uma entrada da forma xy'.

Um outro conceito definido por Ford que iremos adaptar é o de gramática *completa*, que é dado a seguir:

Definição 2.3.5. *Dada uma PEG G, dizemos que G é completa se para toda cadeia w temos que G $w \overset{PEG}{\rightsquigarrow} X$, e portanto existe uma árvore de prova associada ao casamento de w em G através da relação $\overset{PEG}{\rightsquigarrow}$.*

Abaixo, temos o exemplo de uma PEG que não é completa:

$$A \rightarrow A\,a \,/\, b$$

Essa PEG não é completa pois durante o casamento de uma cadeia w, se substituirmos o não terminal A pela sua expressão de parsing associada iremos obter novamente esse mesmo não terminal A e iremos tentar casar a mesma

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{G[A]\ \ w \overset{\text{PEG}}{\rightsquigarrow}}}{G[A\,a]\ \ w \overset{\text{PEG}}{\rightsquigarrow}}}{G[A\,a\,/\,b]\ \ w \overset{\text{PEG}}{\rightsquigarrow}}}{G[A]\ \ w \overset{\text{PEG}}{\rightsquigarrow}}$$

Figura 2.3: Exemplo de Casamento Quando uma PEG não é Completa

entrada w. Dado que a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$ é determinística, então esse processo se repetirá infinitas vezes e não irá resultar em uma árvore de prova, como podemos ver na figura 2.3:

No caso anterior, onde tentamos casar infinitas vezes o mesmo não terminal e a mesma entrada, dizemos que a gramática é *recursiva à esquerda*, ou possui produções recursivas à esquerda.

Outro exemplo de PEGs que não são completas são gramáticas que possuem expressões de parsing da forma p^*, onde p casa a cadeia vazia. Nesse caso, a expressão de parsing p pode casar a cadeia vazia infinitas vezes.

No seu artigo sobre PEGs [Ford, 2004], Ford apresenta uma relação que pode ser usada para determinar se uma gramática é ou não completa de acordo com a sua estrutura. Com base nessa relação, Ford mostrou que uma gramática é completa se ela não é recursiva à esquerda nem possui expressões p^* onde p casa a cadeia vazia.

A seguir, vamos enunciar um lema que relaciona a cadeia de entrada na parte de baixo de uma árvore de prova com a cadeia de entrada de uma subárvore:

Lema 2.3.2. *Dada uma PEG G, se existe um casamento $G\ x \overset{PEG}{\rightsquigarrow} X$ então para toda subárvore $G\ y \overset{PEG}{\rightsquigarrow} X$ desse casamento temos que y é um sufixo de x.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$. Podemos ver que em todas as regras de $\overset{\text{PEG}}{\rightsquigarrow}$ este lema é verdadeiro. □

Por sua vez, o lema a seguir relaciona a cadeia resultante na parte de baixo de uma árvore de prova com a cadeia resultante de uma subárvore:

Lema 2.3.3. *Dada uma PEG G, se existe um casamento $G\ x \overset{PEG}{\rightsquigarrow} x'$ então para toda subárvore $G\ y \overset{PEG}{\rightsquigarrow} y'$ desse casamento temos que x' é um sufixo de y'.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$. Podemos ver que em todas as regras de $\overset{\text{PEG}}{\rightsquigarrow}$ este lema é verdadeiro. □

No próximo capítulo, discutiremos a correspondência entre expressões regulares e PEGs, e no capítulo 4 abordaremos a correspondência entre CFGs lineares à direita e $LL(k)$-forte e PEGs.

3
Expressões Regulares e PEGs

Este capítulo apresenta uma formalização de expressões regulares usando semântica natural, discute a correspondência entre expressões regulares e PEGs, e define uma transformação entre expressões regulares e PEGs equivalentes. Além disso, mostramos aqui como modificar uma expressão regular de modo a obter uma expressão sem subexpressões da forma e_1^* onde e_1 casa a cadeia vazia. Essa modificação nos permite transformar expressões regulares em PEGs sem produções recursivas à esquerda.

Na próxima seção revisamos alguns conceitos de expressões regulares e na seção 3.2 apresentamos nossa formalização de expressões regulares baseada em semântica natural. Na seção 3.3, discutimos a equivalência entre expressões regulares e PEGs. A seção 3.4 descreve como podemos transformar uma expressão regular em uma PEG equivalente. A seção 3.5 mostra como podemos reescrever expressões regulares da forma e_1^* onde e_1 casa a cadeia vazia. Por fim, na seção 3.6, provamos a equivalência entre expressões regulares e PEGs obtidas a partir da transformação apresentada na seção 3.4.

3.1
Expressões Regulares

Dado um alfabeto finito Σ, podemos definir uma expressão regular e_0 indutivamente como a seguir, onde $\mathsf{a} \in \Sigma$, e e_1 e e_2 também são expressões regulares:

$$e_0 \;=\; \emptyset \;\mid\; \varepsilon \;\mid\; a \;\mid\; e_1\, e_2 \;\mid\; e_1 \mid e_2 \;\mid\; e_1^*$$

Geralmente a linguagem definida por uma expressão regular e_0, denotada por $L(e_0)$, é especificada através de operações sobre conjuntos [Hopcroft e Ullman, 1979, Sipser, 1996], conforme a tabela 3.1.

A linguagem definida pela expressão regular $\emptyset$ é o conjunto vazio. Quando não queremos definir uma expressão regular cuja linguagem é vazia o uso de $\emptyset$ é desnecessário, uma vez que qualquer expressão regular e_0 pode ser reduzida a uma expressão regular e_1 que define a mesma linguagem, onde $e_1 = \emptyset$ ou e_1 não possui $\emptyset$ como uma subexpressão. Essa transformação é

Expressão Regular	Linguagem Correspondente
$\emptyset$	$\emptyset$
ε	$\{\ \varepsilon\ \}$
a	$\{\ \mathtt{a}\ \}$
$e_1\, e_2$	$L(e_1)\, L(e_2)$
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$
e_1^*	$L(e_1)^*$

Tabela 3.1: Expressões Regulares e suas Linguagens Correspondentes

baseada nas seguintes igualdades:

$$L(e_0\, \emptyset) \;=\; L(\emptyset) \qquad L(e_0 \mid \emptyset) \;=\; L(e_0) \qquad L(\emptyset^*) \;=\; L(\varepsilon)$$

Como o uso da expressão regular $\emptyset$ é bastante restrito, em algumas discussões não iremos considerar essa forma de expressão regular. Em particular, iremos assumir que $\emptyset$ nunca é uma subexpressão de uma expressão regular $e_0 \neq \emptyset$.

3.2 Definição de Expressões Regulares Usando Semântica Natural

Como nosso objetivo principal é estabelecer a correspondência entre expressões regulares e PEGs, apresentaremos uma formalização diferente de expressões regulares. Nessa nova formalização, usamos a relação $\overset{\text{RE}}{\rightsquigarrow}$ para dar significado a uma expressão regular.

Definimos $\overset{\text{RE}}{\rightsquigarrow}$ como uma relação $(e_0 \times \Sigma^*) \times \Sigma^*$, onde $\overset{\text{RE}}{\rightsquigarrow}$ relaciona uma expressão regular e_0 e uma entrada xy com um sufixo y da entrada. Usamos a notação $e_0\ \ xy \overset{\text{RE}}{\rightsquigarrow} y$ para indicar que $((e_0,\, xy),\, y) \in \overset{\text{RE}}{\rightsquigarrow}$. A figura 3.1 apresenta a definição de $\overset{\text{RE}}{\rightsquigarrow}$ usando semântica natural.

A relação $\overset{\text{RE}}{\rightsquigarrow}$ não é uma função, uma vez que um par $(e_0,\, x)$ pode não se relacionar com nenhum sufixo de x, ou se relacionar com diferentes sufixos. A seguir, discutimos as regras de $\overset{\text{RE}}{\rightsquigarrow}$.

A regra *empty*.1 trata do caso em que a expressão regular representa a cadeia vazia. A regra *char*.1 trata de expressões regulares da forma a, e a regra *con*.1 trata de expressões regulares da forma $e_1\, e_2$.

As regras *choice*.1 e *choice*.2 tratam de expressões regulares da forma $e_1 \mid e_2$. Usaremos o termo *escolha* ao nos referirmos à união das linguagens de duas expressões regulares. O resultado do casamento de uma escolha pode ser tanto o resultado do casamento de e_1 (regra *choice*.1), como o resultado do casamento de e_2 (regra *choice*.2).

Finalmente, as regras *rep*.1 e *rep*.2 tratam de expressões regulares da

Cadeia Vazia $\dfrac{}{\varepsilon\ x \overset{\text{RE}}{\rightsquigarrow} x}\,(\textbf{empty.1})$ **Caractere** $\dfrac{}{a\ ax \overset{\text{RE}}{\rightsquigarrow} x}\,(\textbf{char.1})$

Concatenação $\dfrac{e_1\ xyz \overset{\text{RE}}{\rightsquigarrow} yz \quad e_2\ yz \overset{\text{RE}}{\rightsquigarrow} z}{e_1\, e_2\ xyz \overset{\text{RE}}{\rightsquigarrow} z}\,(\textbf{con.1})$

Escolha $\dfrac{e_1\ xy \overset{\text{RE}}{\rightsquigarrow} y}{e_1 \mid e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y}\,(\textbf{choice.1})$ $\dfrac{e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y}{e_1 \mid e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y}\,(\textbf{choice.2})$

Repetição $\dfrac{}{e_0^*\ x \overset{\text{RE}}{\rightsquigarrow} x}\,(\textbf{rep.1})$ $\dfrac{e_0\ xyz \overset{\text{RE}}{\rightsquigarrow} yz \quad e_0^*\ yz \overset{\text{RE}}{\rightsquigarrow} z}{e_0^*\ xyz \overset{\text{RE}}{\rightsquigarrow} z}\ ,\ x \neq \varepsilon\,(\textbf{rep.2})$

Figura 3.1: Definição da Relação $\overset{\text{RE}}{\rightsquigarrow}$ Usando Semântica Natural

forma e_0^*. Usaremos o termo *repetição* ao nos referirmos ao fecho de Kleene de uma expressão regular. A regra *rep*.1 trata do caso em que nenhum caractere da entrada é consumido, enquanto a regra *rep*.2 trata do caso em que algum caractere da entrada é consumido. Caso a regra *rep*.2 não tivesse a condição $x \neq \varepsilon$, poderíamos casar a cadeia vazia de modo não determinístico através das regras *rep*.1 e *rep*.2.

Podemos notar que nenhuma regra de $\overset{\text{RE}}{\rightsquigarrow}$ trata da expressão regular $\emptyset$, de modo que essa expressão não se relaciona com nenhuma cadeia, e portanto a linguagem que ela define é vazia.

3.2.1 Correspondência com a Definição Usual de Expressões Regulares

Dada a nova definição de expressões regulares baseada na relação $\overset{\text{RE}}{\rightsquigarrow}$, vamos estabelecer a sua correspondência com a definição usual de expressões regulares que é baseada em operações sobre conjuntos. Para isso, vamos precisar do seguinte lema:

Lema 3.2.1. *Dada uma expressão regular* e_0 *e uma cadeia* x*, temos que* $x \in L(e_0)^*$ *se e somente se* $x = \varepsilon$ *ou* $x = x_1x_2$*, onde* $x_1 \in L(e_0)$*,* $x_2 \in L(e_0)^*$ *e* $x_1 \neq \varepsilon$*.*

Demonstração. Por definição, temos que $L(e_0)^* = \bigcup_{i=0}^{\infty} L(e_0)^i$. Portanto, $x \in L(e_0)^* \Leftrightarrow \exists i \cdot x \in L(e_0)^i$. Seja j o menor número natural tal que $x \in L(e_0)^j$, temos então dois casos dependendo se j é ou não maior que zero.

Se $j = 0$, temos que $L(e_0)^0 = \{\varepsilon\}$, e portanto $x = \varepsilon$.

Se $j > 0$, temos que $L(e_0)^j = L(e_0)\,L(e_0)^{j-1}$. Como $x \in L(e_0)\,L(e_0)^{j-1}$, então $x = x_1x_2$, com $x_1 \in L(e_0)$ e $x_2 \in L(e_0)^{j-1}$. Nesse caso, a cadeia x_1 é diferente de ε, pois se $x_1 = \varepsilon$ então teríamos que $x = x_2 \in L(e_0)^{j-1}$, o que contradiz o fato de que escolhemos o menor natural j tal que $x \in L(e_0)^j$. □

A proposição a seguir estabelece a correspondência entre a formalização usual de expressões regulares e a nossa formalização:

Proposição 3.2.2. *Dada uma expressão regular e_0 e uma cadeia x, para qualquer cadeia y temos que $x \in L(e_0)$ se e somente se $e_0\ xy \overset{RE}{\rightsquigarrow} y$.*

Demonstração. A prova das duas partes é por indução na complexidade do par $(e_0,\, x)$, que é dada pela estrutura de e_0 e pelo comprimento de x. Dados os pares $(e_1,\, x_1)$ e $(e_2,\, x_2)$, temos que a complexidade do primeiro par é maior do que a do segundo se a estrutura de e_1 é maior do que a estrutura de e_2, ou se a estrutura de e_1 é igual à estrutura de e_2 e $|x_1| > |x_2|$.

($\Rightarrow$): Quando $e_0 = \varepsilon$, somente $\varepsilon \in L(\varepsilon)$. Nesse caso, pela regra *empty*.1 temos que $\varepsilon\ xy \overset{\text{RE}}{\rightsquigarrow} y$, onde $x = \varepsilon$.

Quando $e_0 = a$, somente $\mathtt{a} \in L(a)$. Nesse caso, pela regra *char*.1 temos que $a\ ay \overset{\text{RE}}{\rightsquigarrow} y$.

Quando $e_0 = e_1\,e_2$, temos que $L(e_1\,e_2) = L(e_1)\,L(e_2)$. Seja $x_1x_2 \in L(e_1\,e_2)$, onde $x_1 \in L(e_1)$ e $x_2 \in L(e_2)$. Pela hipótese de indução temos que $e_1\ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} x_2y$ e que $e_2\ x_2y \overset{\text{RE}}{\rightsquigarrow} y$. Assim, pela regra *con*.1 concluímos que $e_1\,e_2\ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} y$.

Quando $e_0 = e_1 \mid e_2$, temos que $L(e_1 \mid e_2) = L(e_1) \cup L(e_2)$. Vamos considerar dois casos: $x \in L(e_1)$ e $x \in L(e_2)$.

Se $x \in L(e_1)$, pela hipótese de indução temos que $e_1\ xy \overset{\text{RE}}{\rightsquigarrow} y$, e pela regra *choice*.1 concluímos que $e_1 \mid e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y$.

Se $x \in L(e_2)$, pela hipótese de indução temos que $e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y$, e pela regra *choice*.2 concluímos que $e_1 \mid e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y$.

Finalmente, quando $e_0 = e_1^*$, seja $x \in L(e_1^*)$, pelo lema 3.2.1 sabemos que $x = \varepsilon$ ou $x = x_1x_2$, onde $x_1 \in L(e_1)$, $x_2 \in L(e_1^*)$, e $x_1 \neq \varepsilon$.

Se $x = \varepsilon$, pela regra *rep*.1 concluímos que $e_1^*\ y \overset{\text{RE}}{\rightsquigarrow} y$.

Se $x = x_1x_2$, dado que a estrutura de e_1 é menor do que a estrutura de e_1^*, pela hipótese de indução temos que $e_1\ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} x_2y$. Dado que $x_1 \neq \varepsilon$, sabemos que $|x_2| < |x_1x_2|$. Assim, pela hipótese de indução temos que $e_1^*\ x_2y \overset{\text{RE}}{\rightsquigarrow} y$, e pela regra *rep*.2 concluímos que $e_1^*\ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} y$.

($\Leftarrow$): Quando $e_0 = \varepsilon$, temos o casamento $\varepsilon\ xy \overset{\text{RE}}{\rightsquigarrow} y$, e pela definição usual de expressões regulares temos que $\varepsilon \in L(\varepsilon)$.

Quando $e_0 = a$, temos o casamento $a\ ay \overset{\text{RE}}{\rightsquigarrow} y$, e pela definição usual de expressões regulares temos que $\mathtt{a} \in L(a)$.

Quando $e_0 = e_1\, e_2$, temos o casamento $e_1\, e_2\ \ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} y$, onde pela regra *con*.1 sabemos que $e_1\ \ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} x_2y$ e que $e_2\ \ x_2y \overset{\text{RE}}{\rightsquigarrow} y$. Assim, pela hipótese de indução temos que $x_1 \in L(e_1)$ e que $x_2 \in L(e_2)$, e portanto $x_1x_2 \in L(e_1\, e_2)$.

Quando $e_0 = e_1 \,|\, e_2$, temos o casamento $e_1 \,|\, e_2\ \ xy \overset{\text{RE}}{\rightsquigarrow} y$. Vamos considerar dois casos: no primeiro caso a regra *choice*.1 foi usada, e no segundo caso a regra *choice*.2 foi usada.

Se *choice*.1 foi usada, então $e_1\ \ xy \overset{\text{RE}}{\rightsquigarrow} y$, e pela hipótese de indução temos que $x \in L(e_1)$. Assim, concluímos que $x \in L(e_1 \,|\, e_2)$.

Se *choice*.2 foi usada, então $e_2\ \ xy \overset{\text{RE}}{\rightsquigarrow} y$, e pela hipótese de indução temos que $x \in L(e_2)$. Assim, concluímos que $x \in L(e_1 \,|\, e_2)$.

Quando $e_0 = e_1^*$, temos o casamento $e_1^*\ \ xy \overset{\text{RE}}{\rightsquigarrow} xy$. Vamos considerar dois casos: no primeiro caso a regra *rep*.1 foi usada, e no segundo caso a regra *rep*.2 foi usada.

Se *rep*.1 foi usada, então $e_1^*\ \ xy \overset{\text{RE}}{\rightsquigarrow} y$, onde $x = \varepsilon$, e pela definição usual de expressões regulares sabemos que $\varepsilon \in L(e_1^*)$.

Se *rep*.2 foi usada, seja $x = x_1x_2$, temos que $e_1\ \ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} x_2y$ e que $e_1^*\ \ x_2y \overset{\text{RE}}{\rightsquigarrow} y$, onde $x_1 \neq \varepsilon$. Dado que a estrutura de e_1 é menor do que a estrutura de e_1^*, pela hipótese de indução temos que $x_1 \in L(e_1)$, e como $|x_2| < |x_1x_2|$ pela hipótese de indução temos que $x_2 \in L(e_1^*)$. Como $L(e_1\, e_1^*) \subset L(e_1^*)$, concluímos que $x_1x_2 \in L(e_1^*)$. □

3.2.2 Propriedades de Expressões Regulares

A seguir, definimos um lema a respeito do casamento de expressões regulares quando mudamos o sufixo da entrada que não foi casado:

Lema 3.2.3. *Dada uma expressão regular e_0, temos que se $e_0\ \ xy \overset{RE}{\rightsquigarrow} y$ então $\forall y' \cdot e_0\ \ xy' \overset{RE}{\rightsquigarrow} y'$.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{RE}}{\rightsquigarrow}$.

No caso de uma repetição e_0^*, temos que as regras *rep*.1 e *rep*.2 podem ter sido usadas.

Se a regra *rep*.1 foi usada, sabemos que $e_0^*\ \ xy \overset{\text{RE}}{\rightsquigarrow} y$, onde $x = \varepsilon$, e pela própria regra *rep*.1 concluímos que $e_0^*\ \ xy' \overset{\text{RE}}{\rightsquigarrow} y'$.

Se a regra *rep*.2 foi usada, sabemos que $e_0\ \ x_1x_2y \overset{\text{RE}}{\rightsquigarrow} x_2y$ e que $e_0^*\ \ x_2y \overset{\text{RE}}{\rightsquigarrow} y$, onde $x = x_1x_2$. Pela hipótese de indução temos que $e_0\ \ x_1x_2y' \overset{\text{RE}}{\rightsquigarrow} x_2y'$ e que $e_0^*\ \ x_2y' \overset{\text{RE}}{\rightsquigarrow} y'$, e pela regra *rep*.2 concluímos que $e_0^*\ \ x_1x_2y' \overset{\text{RE}}{\rightsquigarrow} y'$.

A prova dos outros casos é similar. □

No próximo capítumo, iremos mostrar um lema análogo para o casamento de CFGs e discutiremos porque no caso de PEGs não é possível definir tal lema.

3.3 Equivalência Entre Expressões Regulares e PEGs

Nesta seção vamos discutir a correspondência entre expressões regulares e PEGs, e definir quando uma expressão regular é equivalente a uma PEG.

Vamos usar ao longo do texto a notação $x' \preceq x$ para denotar que a cadeia x' é um sufixo da cadeia x, e a notação $x' \prec x$ para denotar que x' é um sufixo próprio de x.

A seguir, apresentamos a nossa definição de equivalência entre expressões regulares e PEGs:

Definição 3.3.1. *Dada uma PEG $G = (V, T, P, p_S)$ e uma expressão regular e_0 sobre o alfabeto $\Sigma = T$, dizemos que elas são equivalentes se para toda cadeia x temos que:*

1. $\forall y \preceq x \cdot G \;\; x \overset{PEG}{\rightsquigarrow} y \Rightarrow e_0 \;\; x \overset{RE}{\rightsquigarrow} y.$
2. $\forall y \preceq x \cdot e_0 \;\; x \overset{RE}{\rightsquigarrow} y \Rightarrow \exists y' \preceq x \cdot G \;\; x \overset{PEG}{\rightsquigarrow} y'.$

Pela definição anterior, uma PEG G é equivalente a uma expressão regular e_0 se para toda cadeia x a cadeia y resultante do casamento de G pode ser obtida através do casamento de e_0, e se para toda cadeia x em que o casamento de e_0 é bem sucedido o casamento de G também é bem sucedido.

Dada uma cadeia x, uma expressão regular e_0 e uma PEG G, vamos usar a notação $G \overset{\preceq x}{\Rightarrow} e_0$ para denotar que para todo $x' \preceq x$ temos que $G \;\; x' \overset{\text{PEG}}{\rightsquigarrow} y \Rightarrow e_0 \;\; x' \overset{\text{RE}}{\rightsquigarrow} y$, e a notação $G \overset{\prec x}{\Rightarrow} e_0$ para denotar que para todo $x' \prec x$ temos que $G \;\; x' \overset{\text{PEG}}{\rightsquigarrow} y \Rightarrow e_0 \;\; x' \overset{\text{RE}}{\rightsquigarrow} y$.

De maneira análoga, dada uma cadeia x, uma expressão regular e_0 e uma PEG G, vamos usar a notação $G \overset{\preceq x}{\Leftarrow} e_0$ para denotar que para todo $x' \preceq x$ temos que $e_0 \;\; x' \overset{\text{RE}}{\rightsquigarrow} y \Rightarrow G \;\; x' \overset{\text{PEG}}{\rightsquigarrow} y'$, e a notação $G \overset{\prec x}{\Leftarrow} e_0$ para denotar que para todo $x' \prec x$ temos que $e_0 \;\; x' \overset{\text{RE}}{\rightsquigarrow} y \Rightarrow G \;\; x' \overset{\text{PEG}}{\rightsquigarrow} y'$.

Como uma expressão regular pode se relacionar com diferentes sufixos de uma cadeia x, pois o casamento usando $\overset{\text{RE}}{\rightsquigarrow}$ é não determinístico, e uma PEG pode se relacionar com apenas um único sufixo de uma cadeia x, pois o casamento usando $\overset{\text{PEG}}{\rightsquigarrow}$ é determinístico, quando uma PEG G é equivalente a uma expressão regular e_0 temos que a linguagem definida por e_0 pode ter cadeias que não pertencem à linguagem de G.

Vamos discutir alguns exemplos de expressões regulares e PEGs e analisar se elas são equivalentes ou não.

Se a expressão regular é da forma a, é trivial obter uma PEG equivalente cuja expressão de parsing inicial também é da forma a.

No caso da expressão regular $a \mid ab$, ela é equivalente a uma PEG cuja expressão de parsing inicial é $a \,/\, ab$, embora a expressão regular defina a linguagem {`a`, `ab`}, enquanto que a PEG define a linguagem {`a`}.

A expressão regular $a\,(b \mid bb)$ é equivalente a uma PEG cuja expressão de parsing inicial é $a\,(b \,/\, bb)$. Contudo, a expressão regular $(a \mid aa)\,b$ não é equivalente a uma PEG cuja expressão de parsing inicial é $(a \,/\, aa)\,b$, pois o casamento dessa expressão regular é bem sucedido para a entrada `aab`, ao passo que o casamento dessa PEG não.

Para obter uma PEG equivalente a uma expressão regular da forma $(e_1 \mid e_2)\,e_3$, precisamos criar uma PEG na qual a expressão de parsing p_2, equivalente a e_2, pode casar quando o casamento da expressão de parsing p_1, equivalente a e_1, é bem sucedido, mas o casamento da expressão de parsing p_3, equivalente a e_3, falha. Podemos conseguir isso distribuindo a expressão regular e_3 entre as alternativas da escolha $e_1 \mid e_2$, o que nos daria uma expressão de parsing da forma $p_1\,p_3 \,/\, p_2\,p_3$. Voltando ao exemplo anterior, temos que a expressão regular $(a \mid aa)\,b$ é equivalente a uma PEG cuja expressão de parsing inicial é $ab \,/\, aab$.

No caso da expressão regular $b^*\,b$, ela não é equivalente a uma PEG cuja expressão de parsing inicial é $b^*\,b$. Como o casamento usando $\overset{\text{RE}}{\rightsquigarrow}$ é não determinístico, dada a entrada `bb`, a expressão b^* pode casar ε, `b` ou `bb`, e portanto o casamento da expressão regular $b^*\,b$ pode ser bem sucedido. Por outro lado, como um casamento usando $\overset{\text{PEG}}{\rightsquigarrow}$ é determinístico, dada a entrada `bb` temos que b^* sempre casa `bb`, e o resultado do casamento da expressão de parsing $b^*\,b$ sempre é `fail`. Este problema é análogo ao discutido anteriormente, uma vez que expressões regulares da forma e^* podem ser reescritas como $e\,e^* \mid \varepsilon$.

Para obter uma PEG equivalente a uma expressão regular da forma $e_1^*\,e_2$, vamos reescrever essa expressão regular como a seguir:

$$e_1^*\,e_2 \;\equiv\; (e_1\,e_1^* \;\mid\; \varepsilon)\;e_2 \;\equiv\; e_1\,e_1^*\,e_2 \;\mid\; e_2$$

Como podemos ver, a expressão regular $e_1^*\,e_2$ aparece novamente depois de a reescrevermos. Em PEGs, podemos expressar definições recursivas através de não terminais, de modo que a PEG a seguir é equivalente à expressão regular $e_1\,e_1^*\,e_2 \;\mid\; e_2$, onde p_1 é equivalente a e_1, e p_2 é equivalente a e_2:

$$A \;\rightarrow\; p_1\,A \;\;/\;\; p_2$$

Quando o casamento da expressão regular anterior é bem sucedido através da *choice*.1, sabemos que o casamento da expressão de parsing $p_1\,A$ é bem sucedido. Já quando o casamento da expressão regular anterior usa a regra *choice*.2, sabemos que o casamento da expressão de parsing p_2 é bem sucedido.

Seguindo nosso exemplo, a gramática a seguir é equivalente à expressão regular $b^*\,b$:

$$A \rightarrow b\,A \;/\; b$$

Como vimos anteriormente, essa gramática define uma repetição gulosa. Dada a PEG acima e a entrada `bb`, o primeiro `b` da entrada é casado pela subexpressão b da concatenação $b\,A$ e em seguida temos o casamento do não terminal A, onde a entrada restante é `b`. Novamente, a subexpressão b da concatenação $b\,A$ casa um `b` e em seguida temos o casamento de A, onde a entrada restante agora é ε. Como os casamentos de $b\,A$ e de b falham para a entrada ε, o casamento de A falha para a entrada ε, e portanto o casamento da concatenação $b\,A$ falha para a entrada `b`. Como o casamento da segunda alternativa da escolha associada ao não terminal A é bem sucedido para essa entrada, temos que o casamento de A é bem sucedido para a entrada `b` e que o casamento de $b\,A$ é bem sucedido para a entrada `bb`. Assim, o não terminal A casa a entrada `bb`.

Como o casamento em PEGs é determinístico, dada a entrada `bb`, o não terminal A sempre casa `bb`, ao passo que a expressão regular $b^*\,b$, dada a mesma entrada, pode casar `b` ou `bb`.

3.4 Transformação de uma Expressão Regular em uma PEG Equivalente

Nesta seção vamos apresentar a função Π, que transforma uma dada expressão regular em uma PEG equivalente.

A função Π, cuja definição aparece na figura 3.2, recebe uma expressão regular e_0 e uma PEG $G_k = (V_k, T, P_k, p_k)$, que é equivalente a uma expressão regular e_k, e nos dá uma PEG que é equivalente à expressão regular $e_0\,e_k$.

Podemos ver a gramática G_k como uma continuação, que determina como deve ser o casamento na PEG depois que o casamento da expressão de parsing equivalente a e_0 ocorre. Quando G_k é equivalente a um expressão regular $e_k = \varepsilon$, a transformação Π nos dá uma PEG equivalente à expressão regular $e_0\,\varepsilon \equiv e_0$.

Se a expressão regular é a constante $\emptyset$, o resultado de $\Pi(\emptyset, G_k)$ é uma gramática cuja expressão de parsing inicial é $!\varepsilon$. O resultado do casamento

$$
\begin{aligned}
&\Pi(\emptyset,\, G_k) \;=\; G_k[!\varepsilon] \\
&\Pi(\varepsilon,\, G_k) \;=\; G_k \\
&\Pi(a,\, G_k) \;=\; G_k[a\, p_k] \\
&\Pi(e_1\, e_2,\, G_k) \;=\; \Pi(e_1,\, \Pi(e_2,\, G_k)) \\
&\Pi(e_1 \mid e_2,\, G_k) \;=\; G_2[p_1 \,/\, p_2] \text{ , onde } G_2 = (V_2,\, T,\, P_2,\, p_2) \;=\; \Pi(e_2,\, (V_1,\, T,\, P_1,\, p_k)) \text{ e} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (V_1,\, T,\, P_1,\, p_1) \;=\; \Pi(e_1,\, G_k) \\
&\Pi(e_1^*,\, G_k) \;=\; G \text{ , onde } G \;=\; (V_1,\, T,\, P_1 \cup \{A \to p_1 \,/\, p_k\},\, A) \text{ ,} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (V_1,\, T,\, P_1,\, p_1) \;=\; \Pi(e_1,\, (V_k \cup \{A\},\, T,\, P_k,\, A)) \text{ e} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad A \notin V_k
\end{aligned}
$$

Figura 3.2: Definição da Função Π, onde $G_k = (V_k,\, T,\, P_k,\, p_k)$

dessa expressão de parsing é sempre `fail`, de modo que $L(G_k[!\varepsilon]) = \emptyset$.

No caso de uma expressão regular ε, o resultado da função Π é a mesma gramática G_k que fornecemos a ela.

Quando a expressão regular é da forma a, a função Π nos dá uma gramática com o mesmo conjunto de terminais, não terminais e produções de G_k, e que possui $a\, p_k$ como sua expressão de parsing inicial.

Quando a expressão regular fornecida a Π é uma concatenação $e_1\, e_2$, primeiro criamos uma gramática $\Pi(e_2,\, G_k)$, que é equivalente a $e_2\, e_k$, e depois a usamos como continuação para obter uma PEG que é equivalente à expressão regular $e_1\, (e_2\, e_k)$.

Se a expressão regular é uma escolha $e_1 \mid e_2$, criamos uma gramática que possui $p_1 \,/\, p_2$ como sua expressão de parsing inicial, e que é equivalente à expressão regular $e_1\, e_k \mid e_2\, e_k$. Ao criarmos essa gramática, usamos uma gramática intermediária, pois a transformação de e_1 pode introduzir novos não terminais, que não devem ser usados durante a transformação de e_2, e vice versa. Ao invés de transformarmos primeiro e_1 e então e_2, poderíamos ter transformado primeiro e_2 e então e_1.

Finalmente, quando a expressão regular é uma repetição e_1^*, criamos uma gramática G que é equivalente à expressão regular $e_1\, e_1^*\, e_k \mid e_k$. Ao criarmos G adicionamos um novo não terminal A à gramática G_k, de modo que $A \notin V_k$. A produção associada a esse não terminal é $A \to p_1 \,/\, p_k$, onde temos que $G[p_1]$ é equivalente a $e_1\, e_1^*\, e_k$, e que $G[p_k]$ é equivalente a e_k.

A PEG resultante da transformação Π é linear à direita. No próximo capítulo, na seção 4.4, discutiremos a correspondência entre CFGs lineares à direita e PEGs.

3.4.1
Exemplos de Uso da Transformação Π

Vamos ver agora alguns exemplos de uso da transformação Π. Na discussão a seguir, usaremos o alfabeto $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, e a gramática $G_k = (\emptyset, \Sigma, \emptyset, \varepsilon)$.

No primeiro exemplo, vamos usar a expressão regular e_0 a seguir, que casa cadeias que possuem pelo menos um `a`:

$$(a \mid b \mid c)^* \, a \, (a \mid b \mid c)^*$$

O resultado da transformação $\Pi(e_0, G_k)$ é uma PEG que possui as seguintes produções, onde A é a expressão de parsing inicial:

$$A \rightarrow a\,A \;/\; b\,A \;/\; c\,A \;/\; a\,B \qquad B \rightarrow a\,B \;/\; b\,B \;/\; c\,B \;/\; \varepsilon$$

Quando a expressão regular e_0 casa uma dada entrada, não sabemos quantos `a`'s a primeira repetição de e_0 casa, pois o casamento de uma expressão regular é não determinístico. Por outro lado, como o casamento de uma PEG é determinístico, temos que o casamento da alternativa $a\,B$ do não terminal A só é bem sucedido quando o casamento da alternativa $a\,A$ falha, de modo que A casa um prefixo da entrada que se estende até o último caractere `a`, e portanto o não terminal B não casa nenhum `a`.

A expressão regular a seguir define a mesma linguagem da expressão regular anterior:

$$(b \mid c)^* \, a \, (a \mid b \mid c)^*$$

Para esta expressão regular, obtemos a seguinte gramática G' como resultado da transformação Π, onde A é a expressão de parsing inicial:

$$A \rightarrow b\,A \;/\; c\,A \;/\; a\,B \qquad B \rightarrow a\,B \;/\; b\,B \;/\; c\,B \;/\; \varepsilon$$

Embora a gramática G' seja similar à gramática G, um reconhecedor para G' baseado em backtracking apresenta um desempenho melhor do que um reconhecedor correspondente para G. A razão para isso é que o reconhecedor de G analisa mais de uma vez a porção da entrada que segue o último `a`, enquanto que o reconhecedor de G' não.

No próximo exemplo, vamos definir uma expressão regular e_0 que casa cadeias onde todo `a` é seguido por no mínimo um `b` ou um `c`. A definição de e_0 é apresenta a seguir:

$$(b \mid c)^* \, (a \, (b \mid c) \, (b \mid c)^*)^*$$

O resultado da transformação $\Pi(e_0, G_k)$ é uma gramática G que possui as seguintes produções, onde A é a expressão de parsing inicial:

$$A \rightarrow b\,A \;/\; c\,A \;/\; B \qquad B \rightarrow a\,(b\,C \;/\; c\,C) \;/\; \varepsilon \qquad C \rightarrow b\,C \;/\; c\,C \;/\; B$$

Como a expressão regular e_0 possui três repetições, a PEG G obtida a partir da transformação Π possui três não terminais. Dada a entrada `abaca`, o resultado do casamento de e_0 pode ser ε, `ab`, ou `abac`. Dada a mesma entrada, o resultado do casamento da gramática G é `abac`, pois o resultado do casamento de uma PEG para uma dada entrada é único.

3.5 Transformação de Repetições e_1^* onde e_1 Casa a Cadeia Vazia

Dada uma expressão regular e_0, essa expressão pode ter subexpressões da forma e_1^* onde e_1 casa a cadeia vazia. Nesse caso, como o casamento de e_1 pode ser bem sucedido e não consumir nenhum caractere da entrada, temos que a expressão e_1^* pode casar um número infinito de vezes.

Para evitar que uma repetição infinita ocorra, bibliotecas de casamento de padrões baseadas em backtracking, como a de Perl [Wall, 2000], implementam políticas para interromper um casamento quando a expressão regular que está sendo repetida casa a cadeia vazia [perldoc].

Dada uma expressão regular e_0, iremos dizer que e_0 é uma expressão regular bem formada se ela não possui subexpressões e_1^* onde $\varepsilon \in L(e_1)$.

Quando uma expressão regular não é bem formada, a PEG obtida através da transformação Π não é completa. Para ver um exemplo disso, vamos transformar a seguinte expressão regular e_0, que é mal formada, em uma PEG:

$$(a \mid \varepsilon)^* b$$

A gramática G resultante da transformação $\Pi(e_0, \varepsilon)$, possui a seguinte produção, que é recursiva à esquerda:

$$A \rightarrow a\,A \;/\; A \;/\; b$$

Como G possui produções recursivas à esquerda, G não é uma gramática completa. Na próxima seção, veremos que quando uma expressão regular é bem formada, a gramática obtida através da transformação Π é completa.

No restante desta seção, vamos discutir como podemos reescrever uma expressão regular para torná-la bem formada, e assim evitar o casamento infinito de uma repetição e a geração de uma PEG que não é completa.

$$
\begin{aligned}
isNull(\emptyset) &= \textbf{false} \\
isNull(\varepsilon) &= \textbf{true} \\
isNull(a) &= \textbf{false} \\
isNull(e_1\, e_2) &= isNull(e_1) \wedge isNull(e_2) \\
isNull(e_1 \mid e_2) &= isNull(e_1) \wedge isNull(e_2) \\
isNull(e_1^*) &= isNull(e_1)
\end{aligned}
$$

Figura 3.3: Definição da Função $isNull$

Uma abordagem que poderíamos adotar para obter uma expressão regular bem formada seria definir uma função f que reescreve uma expressão regular de maneira *bottom-up* com base nas seguintes igualdades, onde e_1 não casa a cadeia vazia:

$$L(\varepsilon^*) = L(\varepsilon) \qquad L((e_1 \mid \varepsilon)^*) = L(e_1^*) \qquad L((e_1^*)^*) = L(e_1^*)$$

Nessa abordagem, restringiríamos as possíveis formas de uma expressão regular, de modo que o resultado de f seria uma expressão em que todas as subexpressões teriam uma das seguintes formas, onde e_1 não casa a cadeia vazia:

$$\varepsilon \qquad e_1 \mid \varepsilon \qquad e_1^* \qquad e_1$$

Dado que uma expressão regular poderia estar em uma das quatro formas acima, a função f teria 16 casos para tratar da concatenação, e mais 16 casos para tratar da escolha. Além do grande número de casos necessário para definir a função f, um outro ponto negativo dessa abordagem é que muitas vezes a função f nos daria uma expressão regular bem formada maior do que a expressão regular original.

Como a definição da função f teria muitos casos e o seu resultado às vezes seria uma expressão regular maior do que a original, vamos seguir uma outra abordagem para transformar uma expressão regular e_0 em uma expressão bem formada. Nesta abordagem, vamos usar as funções auxiliares $isNull$, cuja definição aparece na figura 3.3, e $hasEmpty$, cuja definição aparece na figura 3.4.

A função $isNull$ nos diz se a linguagem associada a uma expressão regular contém somente a cadeia vazia. Por sua vez, a função $hasEmpty$ nos diz se a linguagem associada a uma expressão regular contém a cadeia vazia.

Com a ajuda das funções $isNull$ e $hasEmpty$, iremos definir as funções f_{out} e f_{in}, que reescrevem uma expressão regular de maneira *top-down*. O uso

$$
\begin{aligned}
hasEmpty(\emptyset) &= \textbf{false} \\
hasEmpty(\varepsilon) &= \textbf{true} \\
hasEmpty(a) &= \textbf{false} \\
hasEmpty(e_1\, e_2) &= hasEmpty(e_1) \ \wedge \ hasEmpty(e_2) \\
hasEmpty(e_1 \mid e_2) &= hasEmpty(e_1) \ \vee \ hasEmpty(e_2) \\
hasEmpty(e_1^*) &= \textbf{true}
\end{aligned}
$$

Figura 3.4: Definição da Função $hasEmpty$

$$
\begin{aligned}
f_{out}(\emptyset) &= \emptyset \\
f_{out}(\varepsilon) &= \varepsilon \\
f_{out}(a) &= a \\
f_{out}(e_1\, e_2) &= f_{out}(e_1)\, f_{out}(e_2) \\
f_{out}(e_1 \mid e_2) &= f_{out}(e_1) \ \mid \ f_{out}(e_2) \\
f_{out}(e_1^*) &= \begin{cases} f_{out}(e_1)^* & \text{se } \neg hasEmpty(e_1) \\ \varepsilon & \text{se } isNull(e_1) \\ f_{in}(e_1)^* & \text{caso contrário} \end{cases}
\end{aligned}
$$

Figura 3.5: Definição da Função f_{out}

de uma abordagem top-down resultará em uma expressão regular bem formada mais simples do que a expressão original.

Dada uma expressão regular e_0, vamos usar a função f_{out}, cuja definição é apresentada na figura 3.5, para encontrar subexpressões e_1^* onde e_1 casa a cadeia vazia. Quando encontra uma dessas subexpressões, f_{out} usa a função f_{in}, definida na figura 3.6, para reescrever a expressão e_1 de modo que $f_{in}(e_1)$ não case a cadeia vazia e que $f_{in}(e_1)^*$ defina a mesma linguagem que e_1^*.

Com base na definição de f_{out}, dada uma expressão regular e_0, temos que:

- $L(e_0) \ = \ L(f_{out}(e_0))$
- $f_{out}(e_0)$ é bem formada

Já com base na definição de f_{in}, dada uma expressão regular e_0 que casa a cadeia vazia, onde $L(e_0) \neq \{\varepsilon\}$, temos que:

- $\varepsilon \notin \ L(f_{in})$
- $L(e_0^*) \ = \ L(f_{in}(e_0)^*)$
- $f_{in}(e_0)$ é bem formada

$$
\begin{aligned}
f_{in}(e_1\, e_2) &= f_{in}(e_1 \mid e_2) \\
f_{in}(e_1 \mid e_2) &= \begin{cases}
f_{in}(e_2) & \text{se } isNull(e_1) \text{ e } hasEmpty(e_2) \\
f_{out}(e_2) & \text{se } isNull(e_1) \text{ e } \neg hasEmpty(e_2) \\
f_{in}(e_1) & \text{se } hasEmpty(e_1) \text{ e } isNull(e_2) \\
f_{out}(e_1) & \text{se } \neg hasEmpty(e_1) \text{ e } isNull(e_2) \\
f_{out}(e_1) \mid f_{in}(e_2) & \text{se } \neg hasEmpty(e_1) \text{ e } \neg isNull(e_2) \\
f_{in}(e_1) \mid f_{out}(e_2) & \text{se } \neg isNull(e_1) \text{ e } \neg hasEmpty(e_2) \\
f_{in}(e_1) \mid f_{in}(e_2) & \text{caso contrário}
\end{cases} \\
f_{in}(e_1^*) &= \begin{cases}
f_{in}(e_1) & \text{se } hasEmpty(e_1) \\
f_{out}(e_1) & \text{caso contrário}
\end{cases}
\end{aligned}
$$

Figura 3.6: Definição da Função $f_{in}(e_0)$, onde $\neg isNull(e_0)$ e $hasEmpty(e_0)$

É fácil mostrar que as afirmações anteriores sobre f_{out} e f_{in} são verdadeiras. A parte não óbvia da prova é quando a função f_{in} recebe uma expressão da forma $e_1\, e_2$.

Nesse caso, podemos ver na definição de f_{in} que o resultado de $f_{in}(e_1\, e_2)$ é dado por $f_{in}(e_1 \mid e_2)$. Essa definição usa o fato de que a expressão regular recebida por f_{in} sempre casa a cadeia vazia. Desse modo, temos que $\varepsilon \in e_1\, e_2$, e portanto sabemos que $\varepsilon \in (e_1)$ e que $\varepsilon \in (e_2)$. Assim, a seguinte igualdade é verdadeira:

$$L((e_1\, e_2)^*) \;=\; L((e_1 \mid e_2)^*) \;\text{, onde } \varepsilon \in L(e_1\, e_2)$$

Como um exemplo, vamos usar as funções f_{out} e f_{in} para transformar a expressão regular $((a \mid \varepsilon)\, b^*)^*$ em uma expressão bem formada. A seguir, mostramos a sequência de passos dessa transformação:

$$
\begin{aligned}
f_{out}(((a \mid \varepsilon)\, b^*)^*) &= f_{in}((a \mid \varepsilon)\, b^*)^* \\
&= f_{in}((a \mid \varepsilon) \mid b^*)^* \\
&= (f_{in}(a \mid \varepsilon) \mid f_{in}(b^*))^* \\
&= (f_{out}(a) \mid f_{out}(b))^* \\
&= (a \mid b)^*
\end{aligned}
$$

Como podemos ver, as funções f_{out} e f_{in} nos deram expressões bem formadas que estão de acordo com as condições anteriores.

3.6 Corretude da Transformação Π

Nesta seção iremos provar que dada uma expressão regular e_0 e uma PEG $G_k = (V_k, T, P_k, p_k)$, que é equivalente a uma expressão regular e_k, a transformação $\Pi(e_0, G_k)$ nos dá uma PEG que é equivalente a $e_0\, e_k$. Primeiro iremos definir um lema auxiliar, e depois vamos mostrar que dada uma cadeia x temos que $\Pi(e_0, G_k) \overset{\preceq x}{\Rightarrow} e_0\, e_k$ e também que $\Pi(e_0, G_k) \overset{\preceq x}{\Leftarrow} e_0\, e_k$. Em seguida, vamos apresentar uma proposição sobre a equivalência entre expressões regulares e PEGs. Por fim, iremos discutir quando uma expressão regular e uma PEG equivalente definem a mesma linguagem.

Nas provas a seguir, usaremos indução na complexidade de um par (e_0, x), onde e_0 é uma expressão regular e x é uma cadeia de terminais. A complexidade de um par (e_0, x) é dada pela estrutura de e_0 e pelo comprimento de x. Assim, dados os pares (e_1, x_1) e (e_2, x_2), temos que a complexidade do primeiro par é maior do que a do segundo se a estrutura de e_1 é maior do que a estrutura de e_2, ou se a estrutura de e_1 é igual à estrutura de e_2 e $|x_1| > |x_2|$.

Nas provas a seguir, usaremos o fato de que todas as produções da gramática G_k estão presentes na gramática $\Pi(e_0, G_k)$, independentemente da expressão regular e_0.

Vamos começar definindo o seguinte lema, que nos diz que se $\Pi(e_0, G_k)$ casa uma cadeia, então G_k casa um sufixo dessa cadeia:

Lema 3.6.1. *Dada uma expressão regular e_0 bem formada, uma gramática G_k, e uma cadeia x, onde $\Pi(e_0, G_k)\ x \overset{PEG}{\rightsquigarrow} y$, então existe $x' \preceq x$ tal que $G_k\ x' \overset{PEG}{\rightsquigarrow} y$.*

Demonstração. A prova é por indução na complexidade do par (e_0, x).

Quando $e_0 = \varepsilon$, o resultado de $\Pi(\varepsilon, G_k)$ é a própria gramática G_k. Assim, temos que $G_k\ x' \overset{\text{PEG}}{\rightsquigarrow} y$, onde $x = x'$.

Quando $e_0 = a$, o resultado de $\Pi(a, G_k)$ é a gramática $G_k[a\, p_k]$. Desse modo, dada uma cadeia $x = ax'$, quando $G_k[a\, p_k]\ ax' \overset{\text{PEG}}{\rightsquigarrow} y$, a regra $con.1$ nos diz que $G_k[a]\ ax' \overset{\text{PEG}}{\rightsquigarrow} x'$ e que $G_k[p_k]\ x' \overset{\text{PEG}}{\rightsquigarrow} y$.

Quando a expressão regular e_0 é da forma $e_1\, e_2$, temos a transformação $\Pi(e_1\, e_2, G_k) = \Pi(e_1, \Pi(e_2, G_k))$. Pela hipótese de indução temos que $\Pi(e_2, G_k)\ x' \overset{\text{PEG}}{\rightsquigarrow} y$, e novamente pela hipótese de indução concluímos que $G_k\ x'' \overset{\text{PEG}}{\rightsquigarrow} y$, onde x'' é um sufixo de x'.

Quando $e_0 = e_1 \mid e_2$, a transformação Π nos dá uma gramática G_2 cuja expressão de parsing inicial é $p_1 \,/\, p_2$. Há duas regras na semântica de $\overset{\text{PEG}}{\rightsquigarrow}$ que podem ter sido usadas no casamento de $p_1 \,/\, p_2$: $ord.1$ e $ord.2$.

Se a regra *ord*.1 foi usada, então o casamento de $G_2[p_1]$ foi bem sucedido, e portanto sabemos que o casamento de p_1 também é bem sucedido na gramática $(V_1, T, P_1, p_1) = \Pi(e_1, G_k)$. Assim, pela hipótese de indução, concluímos que $G_k\ x' \overset{\text{PEG}}{\rightsquigarrow} y$. A prova é similar se a regra *ord*.2 foi usada.

Finalmente, quando temos uma repetição e_1^*, o resultado de $\Pi(e_1^*, G_k)$ é uma gramática $G = (V_1, T, P, A)$, onde $A \rightarrow p_1 \,/\, p_k$. Pela regra *var*.1 sabemos que se $G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} y$ então temos que $G[p_1 \,/\, p_k]\ x \overset{\text{PEG}}{\rightsquigarrow} y$. Há duas regras em $\overset{\text{PEG}}{\rightsquigarrow}$ que podem ter sido usadas no casamento dessa escolha: *ord*.1 e *ord*.2.

Se a regra *ord*.2 foi usada, então $G[p_k]\ x' \overset{\text{PEG}}{\rightsquigarrow} y$, onde $x = x'$. Assim, temos que o casamento de p_k também é bem sucedido em G_k, e podemos concluir que $G_k[p_k]\ x' \overset{\text{PEG}}{\rightsquigarrow} y$.

Se a regra *ord*.1 foi usada, então $\Pi(e_1, \Pi(e_1^*, G_k))\ x \overset{\text{PEG}}{\rightsquigarrow} y$, e pela hipótese de indução temos que $\Pi(e_1^*, G_k)\ x' \overset{\text{PEG}}{\rightsquigarrow} y$. Nesse caso, não podemos fazer indução na estrutura de e_1^*, então vamos fazer indução no comprimento de x. Dado que e_1 casa um prefixo não vazio da entrada, temos que $x' \prec x$, e pela hipótese de indução concluímos que $G_k\ x'' \overset{\text{PEG}}{\rightsquigarrow} y$, onde x'' é um sufixo de x'. □

O próximo lema nos diz quando uma expressão regular da forma $e_0\, e_k$ casa como a gramática $\Pi(e_0, G_k)$:

Lema 3.6.2. *Seja x uma cadeia de caracteres, e_k uma expressão regular, e G_k uma gramática, onde $G_k \overset{\preceq x}{\Rightarrow} e_k$. Então para qualquer expressão e_0 bem formada temos que $\Pi(e_0, G_k) \overset{\preceq x}{\Rightarrow} e_0\, e_k$.*

Demonstração. A prova é por indução na complexidade do par (e_0, x).

Quando $e_0 = \varepsilon$, temos a transformação $\Pi(\varepsilon, G_k) = G_k$. Dado que $\varepsilon\, e_k \equiv e_k$, concluímos que $G_k \overset{\preceq x}{\Rightarrow} \varepsilon\, e_k$.

Quando $e_0 = a$, a transformação correspondente é $\Pi(a, G_k) = G_k[a\, p_k]$. Dado que $G_k[a] \overset{\preceq x}{\Rightarrow} a$ e que $G_k[p_k] \overset{\preceq x}{\Rightarrow} e_k$, pela regra *con*.1 concluímos que $G_k[a\, p_k] \overset{\preceq x}{\Rightarrow} a\, e_k$.

Quando $e_0 = e_1\, e_2$, a transformação correspondente é $\Pi(e_1\, e_2, G_k) = \Pi(e_1, \Pi(e_2, G_k))$. Pela hipótese de indução $\Pi(e_2, G_k) \overset{\preceq x}{\Rightarrow} e_2\, e_k$, e novamente pela hipótese de indução temos que $\Pi(e_1, \Pi(e_2, G_k)) \overset{\preceq x}{\Rightarrow} e_1\, (e_2\, e_k)$. Dado que $(e_1\, e_2)\, e_k \equiv e_1\, (e_2\, e_k)$, podemos concluir que $\Pi(e_1\, e_2, G_k) \overset{\preceq x}{\Rightarrow} (e_1\, e_2)\, e_k$.

Quando $e_0 = e_1 \mid e_2$, a transformação Π nos dá uma gramática G_2 cuja expressão de parsing inicial é $p_1 \,/\, p_2$. Há duas regras que podem ter sido usadas no casamento desta escolha: *ord*.1 e *ord*.2.

Se a regra *ord*.1 foi usada, então o casamento de $G_2[p_1]$ foi bem sucedido, e portanto o casamento de p_1 também é bem sucedido na gramática $(V_1, T, P_1, p_1) = \Pi(e_1, G_k)$. Pela hipótese de indução temos que $\Pi(e_1, G_k) \overset{\preceq x}{\Rightarrow} e_1\, e_k$, e pela regra *choice*.1 temos que $\Pi(e_1 \mid e_2, G_k) \overset{\preceq x}{\Rightarrow} e_1\, e_k \mid e_2\, e_k$. Dado que

$e_1\, e_k \mid e_2\, e_k \equiv (e_1 \mid e_2)\, e_k$, podemos concluir que $\Pi(e_1 \mid e_2,\, G_k) \overset{\prec x}{\Rightarrow} (e_1 \mid e_2)\, e_k$. A prova é similar se a regra *ord*.2 foi usada.

Finalmente, quando temos uma repetição e_1^*, a transformação $\Pi(e_1^*,\, G_k)$ nos dá uma gramática $G = (V_1,\, T,\, P,\, A)$, onde $A \to p_1 \,/\, p_k$. Pela regra *var*.1 sabemos que se $G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} y$, então temos que $G[p_1 \,/\, p_k]\ x \overset{\text{PEG}}{\rightsquigarrow} y$. Há duas regras que podem ter sido usadas no casamento da escolha anterior: *ord*.1 e *ord*.2.

Se a regra *ord*.2 foi usada, então o casamento de $G[p_k]$ foi bem sucedido, e portanto o casamento de $G_k[p_k]$ também é bem sucedido. Como $G_k[p_k] \overset{\prec x}{\Rightarrow} e_k$, pela regra *choice*.2 temos que $\Pi(e_1^*,\, G_k) \overset{\prec x}{\Rightarrow} e_1\, e_1^*\, e_k \mid e_k$. Dado que $e_1\, e_1^*\, e_k \mid e_k \equiv e_1^*\, e_k$, podemos concluir que $\Pi(e_1^*,\, G_k) \overset{\prec x}{\Rightarrow} e_1^*\, e_k$.

Se a regra *ord*.1 foi usada, temos que $\Pi(e_1,\, \Pi(e_1^*,\, G_k))\ x \overset{\text{PEG}}{\rightsquigarrow} y$, e pelo lema 3.6.1 sabemos que $\Pi(e_1^*,\, G_k)\ x' \overset{\text{PEG}}{\rightsquigarrow} y$. Dado que e_1 casa um prefixo não vazio da entrada, então $x' \prec x$, e podemos fazer indução no comprimento da cadeia de entrada. Assim, pela hipótese de indução temos que $\Pi(e_1^*,\, G_k) \overset{\prec x}{\Rightarrow} e_1^*\, e_k$, e novamente pela hipótese de indução temos que $\Pi(e_1,\, \Pi(e_1^*,\, G_k)) \overset{\prec x}{\Rightarrow} e_1\, (e_1^*\, e_k)$. Portanto, pela regra *choice*.1 temos que $\Pi(e_1^*,\, G_k) \overset{\prec x}{\Rightarrow} e_1\, e_1^*\, e_k \mid e_k$. Dado que $e_1\, e_1^*\, e_k \mid e_k \equiv e_1^*\, e_k$, podemos concluir que $\Pi(e_1^*,\, G_k) \overset{\prec x}{\Rightarrow} e_1^*\, e_k$. □

Agora vamos provar um lema que complementa o lema anterior. O lema a seguir nos diz que podemos usar a função Π para obter uma PEG que casa quando uma dada expressão regular casa:

Lema 3.6.3. *Seja x uma cadeia de caracteres, e_k uma expressão regular, e G_k uma gramática, onde $G_k \overset{\prec x}{\Leftarrow} e_k$. Então para qualquer expressão e_0 bem formada temos que $\Pi(e_0,\, G_k) \overset{\prec x}{\Leftarrow} e_0\, e_k$.*

Demonstração. A prova é por indução na complexidade do par $(e_0,\, x)$.

Quando $e_0 = \varepsilon$, temos que $e_0\, e_k = \varepsilon\, e_k \equiv e_k$. Dado que $\Pi(\varepsilon,\, G_k) = G_k$, concluímos que $\Pi(\varepsilon,\, G_k) \overset{\prec x}{\Leftarrow} \varepsilon\, e_k$.

Quando $e_0 = a$, então $e_0\, e_k = a\, e_k$. Dado que $G_k[a] \overset{\prec x}{\Leftarrow} a$ e que $G_k[p_k] \overset{\prec x}{\Leftarrow} e_k$, pela regra *con*.1 temos que $G_k[a\, p_k] \overset{\prec x}{\Leftarrow} a\, e_k$. Como $\Pi(a,\, G_k) = G_k[a\, p_k]$, podemos concluir que $\Pi(a,\, G_k) \overset{\prec x}{\Leftarrow} a\, e_k$.

Quando $e_0 = e_1\, e_2$, temos que $e_0\, e_k = (e_1\, e_2)\, e_k \equiv e_1\, (e_2\, e_k)$. Pela hipótese de indução temos que $\Pi(e_2,\, G_k) \overset{\prec x}{\Leftarrow} e_2\, e_k$, e novamente pela hipótese de indução temos que $\Pi(e_1,\, \Pi(e_2,\, G_k)) \overset{\prec x}{\Leftarrow} e_1\, (e_2\, e_k)$. Dado que $\Pi(e_1\, e_2,\, G_k) = \Pi(e_1,\, \Pi(e_2,\, G_k))$, podemos concluir que $\Pi(e_1\, e_2,\, G_k) \overset{\prec x}{\Leftarrow} (e_1\, e_2)\, e_k$.

Quando $e_0 = e_1 \mid e_2$, temos que $e_0\, e_k = (e_1 \mid e_2)\, e_k \equiv e_1\, e_k \mid e_2\, e_k$. Há duas regras associadas ao casamento da escolha $e_1 \mid e_2$: *choice*.1 e *choice*.2.

Se a regra *choice*.1 foi usada, então o casamento de e_1 foi bem sucedido. Dado que $(V_1,\, T,\, P_1,\, p_1) = \Pi(e_1,\, G_k)$, pela hipótese de indução temos que

$\Pi(e_1, G_k)[p_1] \overset{\prec x}{\Leftarrow} e_1\, e_k$. Como o casamento de p_1 também é bem sucedido na gramática $(V_2, T, P_2, p_1 \,/\, p_2) = \Pi(e_1 \,|\, e_2, G_k)$, pela regra *ord*.1 concluímos que $\Pi(e_1 \,|\, e_2, G_k) \overset{\prec x}{\Leftarrow} (e_1 \,|\, e_2)\, e_k$.

Se a regra *choice*.2 foi usada, então o casamento de e_2 foi bem sucedido. Dado que o casamento de e_2 foi bem sucedido, vamos analisar dois casos: quando o casamento de e_1 é bem sucedido, e quando o casamento de e_1 não é bem sucedido.

Se o casamento de e_1 é bem sucedido, a prova é similar a quando a regra *choice*.1 foi usada.

Se o casamento de e_1 não é bem sucedido, como a gramática $(V_1, T, P_1, p_1) = \Pi(e_1, G_k)$ é completa, pela contrapositiva do lema 3.6.2 temos que o casamento de $\Pi(e_1, G_k)[p_1]$ não é bem sucedido. Dado que $(V_2, T, P_2, p_2) = \Pi(e_2, \Pi(e_1, G_k)[p_k])$, sabemos que o casamento de p_1 também não é bem sucedido nessa gramática, e como o casamento de e_2 é bem sucedido, pela hipótese de indução temos que $\Pi(e_2, \Pi(e_1, G_k)[p_k]) \overset{\prec x}{\Leftarrow} e_2\, e_k$. Assim, pela regra *ord*.2, concluímos que $\Pi(e_1 \,|\, e_2, G_k) \overset{\prec x}{\Leftarrow} (e_1 \,|\, e_2)\, e_k$.

Finalmente, no caso de uma repetição, temos a expressão regular $e_1^*\, e_k \equiv e_1\, e_1^*\, e_k \,|\, e_k$. Há duas regras que podem ter sido usadas no casamento dessa escolha: *choice*.1 e *choice*.2.

Se a regra *choice*.1 foi usada, pela regra *con*.1 sabemos que o casamento de e_1 é bem sucedido. Dado que e_1 casa uma cadeia não vazia, temos que o casamento de e_1^* é bem sucedido para um sufixo $x' \prec x$. Assim, pela hipótese de indução, temos que $\Pi(e_1^*, G_k) \overset{\prec x}{\Leftarrow} e_1^*\, e_k$. Novamente pela hipótese de indução, temos que $\Pi(e_1, \Pi(e_1^*, G_k)) \overset{\prec x}{\Leftarrow} e_1\, e_1^*\, e_k$. Seja p_1 a expressão de parsing inicial de $\Pi(e_1, \Pi(e_1^*, G_k))$, pela regra *ord*.1 temos que $\Pi(e_1^*, G_k)[p_1 \,/\, p_k] \overset{\prec x}{\Leftarrow} e_1\, e_1^*\, e_k \,|\, e_k$, e pela regra *var*.1 concluímos que $\Pi(e_1^*, G_k) \overset{\prec x}{\Leftarrow} e_1^*\, e_k$.

Se a regra *choice*.2 foi usada, precisamos analisar se o casamento da primeira alternativa da escolha é bem sucedido ou não.

Quando o casamento da primeira alternativa da escolha é bem sucedido, então a prova é análoga a quando a regra *choice*.1 foi usada. Vamos provar então o caso em que o casamento da primeira alternativa da escolha não é bem sucedido.

Seja p_1 a expressão de parsing inicial de $\Pi(e_1, \Pi(e_1^*, G_k))$, como a gramática $\Pi(e_1^*, G_k)$ é completa, pela contrapositiva do lema 3.6.2 temos que o casamento de $\Pi(e_1^*, G_k)[p_1]$ não é bem sucedido. Dado que $G_k[p_k] \overset{\prec x}{\Leftarrow} e_k$, então temos que $\Pi(e_1^*, G_k)[p_k] \overset{\prec x}{\Leftarrow} e_k$. Assim, pela regra *ord*.2, temos que $\Pi(e_1^*, G_k)[p_1 \,/ p_k] \overset{\prec x}{\Leftarrow} e_1\, e_1^*\, e_k \,| e_k$, e pela regra *var*.1 concluímos que $\Pi(e_1^*, G_k) \overset{\prec x}{\Leftarrow} e_1^*\, e_k$. □

Agora, vamos usar os lemas 3.6.2 e 3.6.3 para provar a seguinte proposição a respeito da equivalência entre uma dada expressão regular e_0 e a PEG obtida a partir de e_0 usando a transformação Π.

Proposição 3.6.4. *Dada uma expressão regular e_0 bem formada, temos que a expressão regular $e_0\,\varepsilon \equiv e_0$ é equivalente à gramática $\Pi(e_0, \varepsilon)$.*

Demonstração. Como a expressão regular ε é equivalente à expressão de parsing ε, pelo lema 3.6.2 para qualquer cadeia x temos que $\Pi(e_0, \varepsilon) \overset{\preceq x}{\rightarrow} e_0\,\varepsilon$, e pelo lema 3.6.3 para qualquer cadeia x temos que $\Pi(e_0, \varepsilon) \overset{\preceq x}{\leftarrow} e_0\,\varepsilon$. □

Como um corolário da proposição acima, dado que $a\,\varepsilon \equiv a$ e que $\Pi(a, \varepsilon) = a$, seja e_0 uma expressão regular bem formada então $e_0\,a$ é equivalente à gramática $\Pi(e_0, a)$.

Apesar de termos estabelecido a equivalência entre expressões regulares e PEGs, ainda não discutimos quando uma expressão regular e uma PEG definem a mesma linguagem.

Anteriormente, na proposição 3.2.2, mostramos que $x \in L(e_0)$ se e somente se $e_0\ xy \overset{\text{RE}}{\rightsquigarrow} y$. Nessa definição, o casamento da expressão e_0 usando $\overset{\text{RE}}{\rightsquigarrow}$ não precisa casar toda a entrada. Contudo, tal definição não nos permite mostrar que uma expressão regular e uma PEG equivalente definem a mesma linguagem, pois uma expressão regular pode se relacionar com diferentes sufixos da entrada através da relação $\overset{\text{RE}}{\rightsquigarrow}$, enquanto que uma PEG equivalente irá se relacionar com apenas um sufixo da entrada.

Podemos ver isso no exemplo a seguir. Dada a expressão regular $a \mid a\,b$ e a PEG equivalente $a \,/\, a\,b$, para qualquer cadeia da forma `ax` temos a seguinte árvore de prova associada ao casamento da PEG:

$$\dfrac{\dfrac{}{G[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x}(\textbf{char.1})}{G[a \,/\, b]\ ax \overset{\text{PEG}}{\rightsquigarrow} x}(\textbf{ord.1})$$

Desse modo, a cadeia `ab` $\notin L(a \,/\, a\,b)$, porém `ab` $\in L(a \mid a\,b)$. Assim, precisamos mudar as definições de linguagens de expressões regulares e de PEGs para poder estabelecer a correspondência entre a linguagem de uma expressão regular e de uma PEG equivalente.

Uma forma de fazer isso é usar o símbolo $\$ \notin T$ como um marcador de final da entrada, de modo que uma expressão regular e_0 se relacione com apenas um sufixo de uma determinada entrada. Dessa forma, dada uma expressão regular e_0 e uma entrada x, se $e_0\,\$\ x\$ \overset{\text{RE}}{\rightsquigarrow} \varepsilon$ então $x \in L(e_0)$.

No caso de PEGs, dada uma entrada da forma $x\$$ só há um casamento bem sucedido de uma PEG $\Pi(e_0, \$)$ quando toda a entrada é consumida.

Assim, o casamento de uma alternativa de uma escolha da PEG $\Pi(e_0, \$)$ é bem sucedido somente se toda a entrada é consumida, como a árvore de prova a seguir mostra (omitimos as regras usadas por motivo de espaço):

$$\dfrac{\dfrac{\dfrac{\overline{G[a]\ ab\$ \overset{\text{PEG}}{\rightsquigarrow} b\$} \quad \overline{G[\$]\ b\$ \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}}{G[a\,\$]\ ab\$ \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}} \quad \dfrac{\overline{G[a]\ ab\$ \overset{\text{PEG}}{\rightsquigarrow} b\$} \quad \dfrac{\overline{G[b\$]\ b\$ \overset{\text{PEG}}{\rightsquigarrow} \$} \quad \overline{G[\$]\ \$ \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}{G[b\,\$]\ b\$ \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}{G[a\,b\$]\ ab\$ \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}{G[a\,\$ \,/\, a\,b\,\$]\ ab\$ \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}$$

Com base nisso, vamos definir que se $\Pi(e_0, \$)\ \ x\$ \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ então $x \in L(\Pi(e_0, \varepsilon))$.

Portanto, seja e_0 uma expressão regular bem formada, temos que $e_0\,\$\ \ x\$ \overset{\text{RE}}{\rightsquigarrow} \varepsilon$ se e somente se $G[\Pi(e_0, \$)]\ \ x\$ \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$. Logo, a expressão regular e_0 e a PEG $\Pi(e_0, \varepsilon)$ definem a mesma linguagem.

Existe uma outra forma de estabelecer a correspondência entre a linguagem de uma expressão regular e de uma PEG. Nessa forma, dada uma expressão regular e_0 e uma entrada x, definimos que se $e_0\ \ x \overset{\text{RE}}{\rightsquigarrow} \varepsilon$ então $x \in L(e_0)$. Para definir a linguagem de uma PEG, vamos usar a expressão de parsing **.** que discutimos na seção 2.1 e que é um açúcar sintático para casar qualquer terminal da entrada. Como o casamento da expressão de parsing **!.** é bem sucedido somente quando a entrada é vazia, podemos usar essa expressão para testar se a PEG casou toda a entrada. Assim, vamos definir que se $\Pi(e_0, \texttt{!.})\ \ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ então $x \in \Pi(e_0, \varepsilon)$.

Portanto, dada uma expressão regular e_0 bem formada e uma cadeia x, temos que $e_0\ \ x \overset{\text{RE}}{\rightsquigarrow} \varepsilon$ se e somente se $\Pi(e_0, \texttt{!.})\ \ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$.

4
Gramáticas Livres de Contexto e PEGs

Este capítulo apresenta uma nova formalização de Gramáticas Livres de Contexto (*Context-Free Grammars* —CFGs) e discute a correspondência de CFGs lineares à direita e $LL(k)$-forte com PEGs equivalentes.

Na próxima seção mostramos a definição usual de CFGs e na seção 4.2 apresentamos a nossa formalização de CFGs baseada em semântica natural, que é próxima da formalização de PEGs apresentada no capítulo 2. Em seguida, na seção 4.3, discutimos a interpretação de uma mesma gramática como uma CFG e como uma PEG. A seção 4.4 discute a correspondência entre CFGs lineares à direita e PEGs equivalentes. Na seção 4.5 abordamos a correspondência entre CFGs $LL(1)$ e PEGs, e mostramos que uma gramática $LL(1)$, com uma pequena restrição na ordem das alternativas de uma escolha, define a mesma linguagem quando interpretada como uma CFG e quando interpretada como uma PEG. A seção 4.6 estende essa discussão para gramáticas $LL(k)$-forte, e mostra que a partir de uma CFG $LL(k)$-forte podemos gerar uma PEG equivalente que possui a mesma estrutura da CFG.

4.1
Definição Usual de CFGs

Uma CFG é geralmente definida como uma tupla (V, T, P, S), onde V é um conjunto finito de variáveis ou não terminais, T é um conjunto finito de terminais, P é um conjunto finito de produções, onde uma produção é um par (A, α) que relaciona um não terminal $A \in V$ com uma cadeia α de símbolos da gramática (um símbolo da gramática pode ser um terminal ou um não terminal) e é representada como $A \rightarrow \alpha$, e S representa o não terminal inicial da gramática [Hopcroft e Ullman, 1979].

Dada uma CFG $G = (V, T, P, S)$, usamos a relação $\Rightarrow_G$ para substituir um não terminal A por uma cadeia β de símbolos da gramática, onde $A \rightarrow \beta \in P$. Dessa forma, se α e γ são cadeias de símbolos da gramática e $A \rightarrow \beta \in P$, então $\alpha A \gamma \Rightarrow_G \alpha\beta\gamma$.

A relação $\overset{*}{\Rightarrow}_G$ é o fecho reflexivo transitivo de $\Rightarrow_G$. Dada uma cadeia α de símbolos da gramática, dizemos que α gera (ou deriva) β se temos que

$\alpha \overset{*}{\Rightarrow}_G \beta$.

Dizemos que uma cadeia w pertence à linguagem de uma gramática G quando $S \overset{*}{\Rightarrow}_G w$.

4.2 Uma Nova Formalização de Linguagens Livres de Contexto usando Semântica Natural

Apresentaremos nesta seção uma nova formalização de Linguagens Livres de Contexto (*Context-Free Languages* — CFLs) que é baseada em semântica natural. Essa nova formalização de CFLs usa expressões de parsing e é próxima da formalização de PEGs apresentada no capítulo 2. O uso de uma formalização de CFLs que é próxima da formalização de PEGs nos ajudará a estabelecer a correspondência entre classes de linguagens definidas através de PEGs e através de CFGs.

Assim como em CFGs, na nova formalização de CFLs a descrição de uma linguagem é dada por uma tupla (V, T, P, p_S), onde V é um conjunto finito de não terminais, T é um conjunto finito de terminais, P é uma função de não terminais em expressões de parsing, e p_S é a expressão de parsing inicial.

Diferentemente de CFGs, onde uma produção associa um não terminal com uma cadeia de símbolos da gramática, nessa nova formalização de CFLs temos uma função P que relaciona não terminais com expressões de parsing. Apesar dessa diferença, a descrição visual de algumas linguagens é bastante similar nos dois formalismos. Como exemplo, vamos analisar a descrição a seguir, que está na Forma de Backus-Naur (*Backus-Naur Form* — BNF) [Knuth, 1964],

$$A \rightarrow B\,C \qquad B \rightarrow a \,|\, b \qquad C \rightarrow c \,|\, d$$

Se a descrição acima é de uma CFG, temos o seguinte conjunto de produções:

$$P \;=\; \{\, A \rightarrow B\,C,\;\; B \rightarrow a,\;\; B \rightarrow b,\;\; C \rightarrow c,\;\; C \rightarrow d \,\}$$

Nesse caso, o símbolo | é apenas uma convenção sintática usada para separar cadeias de símbolos da gramática associadas a um mesmo não terminal.

No caso em que a descrição anterior é baseada em expressões de parsing, temos a seguinte definição da função P:

$$P(A) \;=\; B\,C \qquad P(B) \;=\; a \,|\, b \qquad P(C) \;=\; c \,|\, d$$

Nesse caso, o símbolo | é um operador binário.

Para simplificar a descrição de CFLs através de expressões de parsing, vamos assumir que a escolha e a concatenação são associativas à direita, de modo que representaremos como $p_1 \mid p_2 \mid \cdots \mid p_m$ a expressão de parsing $p_1 \mid (p_2 \mid (\cdots \mid p_m))$, e como $p_1\, p_2 \cdots p_n$ a expressão de parsing $p_1\, (p_2\, (\cdots\, p_n))$.

Apesar de usarmos expressões de parsing na nova formalização de CFLs, não daremos significado a expressões de parsing da forma $!p$, pois com predicados poderíamos definir linguagens que não são CFLs, e da forma p^*, pois ela não é necessária para definir CFLs.

Dada a descrição de uma CFL através de expressões de parsing é possível obter uma CFG, assim como a partir de uma CFG podemos obter a descrição de uma CFL através de expressões de parsing. A primeira dessas transformações é mais complicada, de modo que iremos discuti-la a seguir.

Uma vez que na nova formalização de CFLs temos uma expressão de parsing inicial p_S ao invés de um não terminal, no caso em que $p_S \notin V$ devemos modificar a descrição da CFL. Primeiro, adicionamos um novo não terminal A ao conjunto V. Segundo, adicionamos a produção $A \rightarrow p_S$. Finalmente, tornamos A a expressão de parsing inicial.

Feito isso, o próximo passo é eliminar as escolhas que são subexpressões de concatenações. Dado que é possível agrupar expressões de parsing com o uso de parênteses, podemos ter uma expressão de parsing em que uma concatenação possui uma escolha como subexpressão, como mostrado a seguir:

$$A \rightarrow (a \mid b)\, (c \mid d)$$

Acima, o não terminal A está associado a uma conjunção de disjunções, isto é, uma concatenação de escolhas. Para obtermos uma descrição baseada em expressões de parsing que não possui uma escolha como subexpressão de uma concatenação, devemos realizar uma transformação similar a que fazemos para converter uma descrição na Forma de Backus-Naur Estendida (*Extended Backus-Naur Form* — EBNF) [Wirth, 1977, ISO] em uma CFG. Durante essa transformação, devemos criar um novo não terminal para cada escolha que é subexpressão de uma concatenação. A seguir, reescrevemos o exemplo anterior após essa transformação:

$$A \rightarrow B\, C \qquad B \rightarrow a \mid b \qquad C \rightarrow c \mid d$$

Notem que usamos novos não terminais B e C para eliminar a conjunção de disjunções.

Embora intuitivamente as transformações anteriores sejam simples, a

formalização delas é trabalhosa. Como essa formalização não possui aspectos interessantes, preferimos discutir essas transformações somente de maneira informal.

Dado que visualmente a descrição de CFLs através de CFGs é praticamente idêntica à descrição de CFLs através de expressões de parsing, e que, como discutimos acima, é possível transformar uma descrição na outra, vamos cometer um abuso de linguagem e usar daqui em diante o termo CFG (ou gramática) ao nos referirmos à descrição de uma CFL através de expressões de parsing. Quando for necessário fazer uma distinção entre essas duas formalizações de CFLs, iremos usar os termos CFG usual e CFG descrita através de expressões de parsing.

A interpretação de uma CFG $G = (V, T, P, S)$ descrita através de expressões de parsing é feita pela relação $\overset{\text{CFG}}{\rightsquigarrow}$, cuja definição usa semântica natural e é apresentada na figura 4.1. De modo análogo a PEGs, dada uma CFG $G = (V, T, P, p_S)$, usamos a notação $G[p'_S]$ para representar uma nova gramática $G' = (V, T, P, p'_S)$.

Na figura 4.1, podemos ver que dada uma gramática e uma cadeia de entrada, a relação $\overset{\text{CFG}}{\rightsquigarrow}$ casa um prefixo da entrada de acordo com as produções da gramática. De maneira mais precisa, definimos $\overset{\text{CFG}}{\rightsquigarrow}$ como uma relação $(G \times T^*) \times T^*$, onde $\overset{\text{CFG}}{\rightsquigarrow}$ relaciona uma gramática G e uma entrada xy com um sufixo y da entrada. Usamos a notação $G\ xy \overset{\text{CFG}}{\rightsquigarrow} y$ para representar que $((G, xy), y) \in \overset{\text{CFG}}{\rightsquigarrow}$.

A relação $\overset{\text{CFG}}{\rightsquigarrow}$ não é uma função, pois ela não relaciona cada par (G, w) com um sufixo de w, e ela também pode relacionar um par (G, w) com diferentes sufixos de w.

A seguir, definimos a linguagem de uma CFG descrita através de expressões de parsing:

Definição 4.2.1. *Dada uma CFG G descrita através de expressões de parsing, a linguagem de G consiste das cadeias x tais que $G\ xy \overset{\text{CFG}}{\rightsquigarrow} y$, onde y é uma cadeia qualquer.*

Ao longo do texto, usaremos $L(G)$ para representar a linguagem de uma CFG G.

Note na definição anterior que o sufixo y pode ser qualquer cadeia, pois não temos predicados em CFGs e portanto y não é usado para determinar o resultado de um casamento.

A seguir, vamos discutir as linguagens de uma CFG usual e da sua formalização correspondente descrita através de expressões de parsing.

Como vimos anteriormente, a partir de uma CFG $G = (V, T, P, S)$ descrita de forma usual é possível obter uma gramática $G' = (V, T, P', S)$

Cadeia Vazia $\dfrac{}{G[\varepsilon]\ x \overset{\text{CFG}}{\rightsquigarrow} x}\,(\textbf{empty.1})$ **Terminal** $\dfrac{}{G[a]\ ax \overset{\text{CFG}}{\rightsquigarrow} x}\,(\textbf{char.1})$

Variável $\dfrac{G[P(A)]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}{G[A]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}\,(\textbf{var.1})$ **Concatenação** $\dfrac{G[p_1]\ xyz \overset{\text{CFG}}{\rightsquigarrow} yz \quad G[p_2]\ yz \overset{\text{CFG}}{\rightsquigarrow} z}{G[p_1\,p_2]\ xyz \overset{\text{CFG}}{\rightsquigarrow} z}\,(\textbf{con.1})$

Escolha $\dfrac{G[p_1]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}\,(\textbf{choice.1})$ $\dfrac{G[p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}\,(\textbf{choice.2})$

Figura 4.1: Definição da Relação $\overset{\text{CFG}}{\rightsquigarrow}$ Usando Semântica Natural

descrita através de expressões de parsing. Além disso, dada a definição de $\overset{\text{CFG}}{\rightsquigarrow}$ podemos afirmar que $S \Rightarrow_G x$ se e somente se $G'\ xy \overset{\text{CFG}}{\rightsquigarrow} y$.

Para provar formalmente essa correspondência precisamos formalizar a transformação de G em G', o que não iremos fazer. Contudo, dadas as regras da relação $\overset{\text{CFG}}{\rightsquigarrow}$, acreditamos que não é difícil ver que essa correspondência é verdadeira.

Uma consequência da correspondência entre $\Rightarrow_G$ e $\overset{\text{CFG}}{\rightsquigarrow}$ é que w pertence à linguagem de G se e somente se $w \in L(G')$.

Após mostrar a correspondência entre $\Rightarrow_G$ e $\overset{\text{CFG}}{\rightsquigarrow}$, vamos discutir quando uma gramática descrita através de expressões de parsing possui estrutura BNF e enunciar algumas propriedades de CFGs descritas através de expressões de parsing. Mais adiante, na seção 4.3, vamos estudar a correspondência entre as relações $\overset{\text{CFG}}{\rightsquigarrow}$ e $\overset{\text{PEG}}{\rightsquigarrow}$.

4.2.1 Estrutura BNF

A seguir, definimos quando uma CFG descrita através de expressões de parsing possui estrutura BNF:

Definição 4.2.2. *Uma CFG $G = (V, T, P, p_S)$ descrita através de expressões de parsing possui estrutura BNF se ela está de acordo com as seguintes restrições:*

1. *Nenhuma escolha de G é uma subexpressão de uma concatenação.*
2. *$p_S \in V$*
3. *Para toda escolha $p_1 \mid p_2$ de G onde uma alternativa casa a cadeia vazia e a outra não, temos que a alternativa p_2 é a que casa a cadeia vazia.*

A restrição 1 já foi discutida anteriormente, quando mostramos como transformar uma CFG descrita através de expressões em uma CFG usual. Se uma gramática G respeita essa restrição, ela possui uma representação direta na forma BNF.

A segunda restrição diz respeito à expressão de parsing inicial, que deve ser um não terminal. Essa restrição será útil ao provarmos a correspondência entre CFGs $LL(1)$ e $LL(k)$-forte e PEGs.

Por fim, a restrição 3 exige que quando somente uma alternativa de uma escolha casa a cadeia vazia, essa alternativa seja a última. Essa restrição não afeta a linguagem da CFG, pois a linguagem de uma CFG não muda quando na sua descrição BNF alteramos a ordem das alternativas de uma escolha. Apesar dessa restrição parecer um pouco artificial, veremos mais adiante que ela simplifica a discussão da correspondência entre CFGs $LL(1)$ com expressões ε e PEGs.

4.2.2 Propriedades de CFGs

Iremos definir dois lemas a respeito do casamento em uma gramática usando a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$. O primeiro desses lemas possui lemas correspondentes em PEGs, mas o segundo não.

Lema 4.2.1. *Dada uma CFG G, se existe um casamento $G\ x \overset{\text{CFG}}{\rightsquigarrow} x'$ então para toda subárvore $G\ y \overset{\text{CFG}}{\rightsquigarrow} y'$ desse casamento temos que y é um sufixo de x e que x' é um sufixo de y'.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{CFG}}{\rightsquigarrow}$. Vamos provar que o lema é verdadeiro nos antecedentes (subárvores próprias) de todas as regras de $\overset{\text{CFG}}{\rightsquigarrow}$.

Quando a expressão de parsing é da forma ε ou da forma a, como as regras *empty*.1 e *char*.1 não possuem antecedentes o lema é trivialmente satisfeito.

Quando a expressão de parsing é da forma A, pela regra *var*.1 temos que o antecedente é $G[P(A)]\ y \overset{\text{CFG}}{\rightsquigarrow} y'$, onde $y = x$ e $y' = x'$, e pela hipótese de indução concluímos que o lema é verdadeiro nessa subárvore.

Quando a expressão de parsing é da forma $p_1\, p_2$, seja $x = tuv$, pela regra *con*.1 temos as subárvores $G[p_1]\ tuv \overset{\text{CFG}}{\rightsquigarrow} uv$ e $G[p_2]\ uv \overset{\text{CFG}}{\rightsquigarrow} v$. No caso da primeira subárvore, temos que tuv é um sufixo de tuv, e que v é um sufixo de uv, e pela hipótese de indução o lema é verdadeiro nas suas subárvores. Na caso da segunda subárvore, temos que uv é um sufixo de tuv, e que v é um sufixo de v, e pela hipótese de indução o lema é verdadeiro nas suas subárvores.

Quando a expressão de parsing é da forma $p_1 \mid p_2$, as regras associadas são *choice*.1 e *choice*.2. Se a regra *choice*.1 foi usada, temos uma subárvore

$G[p_1]\ \ y \overset{\text{CFG}}{\rightsquigarrow} y'$, onde $y = x$ e $y' = x'$, e pela hipótese de indução o lema é verdadeiro nas suas subárvores. O caso em que a regra *choice*.2 foi usada é análogo. □

O lema anterior é o correspondente para CFGs dos lemas 2.3.2 e 2.3.3, que são a respeito de PEGs. Por outro lado, o próximo lema não possui um lema correspondente em PEGs, por razões que discutiremos logo em seguida:

Lema 4.2.2. *Dada uma CFG G, temos que se $G\ \ xy \overset{CFG}{\rightsquigarrow} y$ então $\forall y' \cdot G\ \ xy' \overset{CFG}{\rightsquigarrow} y'$.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{CFG}}{\rightsquigarrow}$.

No caso de uma concatenação $p_1\, p_2$, somente a regra *con*.1 pode ter sido usada. Por essa regra sabemos que $G[p_1]\ \ x_1x_2y \overset{\text{CFG}}{\rightsquigarrow} x_2y$ e que $G[p_2]\ \ x_2y \overset{\text{CFG}}{\rightsquigarrow} y$, onde $x = x_1x_2$. Pela hipótese de indução temos que $G[p_1]\ \ x_1x_2y' \overset{\text{CFG}}{\rightsquigarrow} x_2y'$ e que $G[p_2]\ \ x_2y' \overset{\text{CFG}}{\rightsquigarrow} y'$. Assim, pela regra *con*.1 concluímos que $G[p_1\, p_2]\ \ x_1x_2y' \overset{\text{CFG}}{\rightsquigarrow} y'$.

A prova dos outros casos é similar. □

Dada uma PEG G, onde G é livre de predicado, em um primeiro momento poderíamos pensar em adaptar o lema 4.2.2 para PEGs. Contudo, essa adaptação do lema 4.2.2 não é possível. Para ver a razão disso, vamos considerar a PEG a seguir, onde S é a expressão de parsing inicial da gramática:

$$S \rightarrow A\,B \qquad A \rightarrow a\,b\,a \,/\, a \qquad B \rightarrow b$$

Dada a entrada `abc`, temos o casamento $G\ \ abc \overset{\text{PEG}}{\rightsquigarrow} c$, com subárvores $G[A]\ \ abc \overset{\text{PEG}}{\rightsquigarrow} bc$ e $G[B]\ \ bc \overset{\text{PEG}}{\rightsquigarrow} c$. Dada uma nova entrada `abac`, temos o casamento $G\ \ abac \overset{\text{PEG}}{\rightsquigarrow}$ `fail`, com subárvores $G[A]\ \ abac \overset{\text{PEG}}{\rightsquigarrow} c$ e $G[B]\ \ c \overset{\text{PEG}}{\rightsquigarrow}$ `fail`. Como podemos ver, ao mudarmos um sufixo da entrada, o não terminal A passou a casar um prefixo diferente, o que fez com que o casamento de B falhasse e com que os casamentos de $A\,B$ e de S também falhassem.

Em resumo, no caso de expressões regulares e CFGs os lemas 3.2.3 e 4.2.2, nos dizem que quando uma expressão regular ou CFG casa um prefixo x da entrada e deixa um sufixo y, podemos mudar esse sufixo y da entrada e ainda casar o mesmo prefixo x. Já no caso de PEGs, com base na discussão anterior podemos concluir que quando uma PEG casa um prefixo x da entrada e deixa um sufixo y, isso não implica que podemos mudar esse sufixo y para y' e obter um casamento bem sucedido para a entrada xy'.

4.3 Correspondência entre $\overset{\text{CFG}}{\rightsquigarrow}$ e $\overset{\text{PEG}}{\rightsquigarrow}$

Com base na definição de $\overset{\text{CFG}}{\rightsquigarrow}$ apresentada na seção anterior e na definição de $\overset{\text{PEG}}{\rightsquigarrow}$ apresentada na seção 2.3, vamos comparar as regras de $\overset{\text{CFG}}{\rightsquigarrow}$ com as regras de $\overset{\text{PEG}}{\rightsquigarrow}$ para ver as semelhanças e as diferenças entre CFGs e PEGs.

Podemos notar que somente $\overset{\text{PEG}}{\rightsquigarrow}$ possui regras onde o resultado de um casamento é `fail`. As regras de $\overset{\text{PEG}}{\rightsquigarrow}$ usam `fail` porque, como discutimos no capítulo 2, a noção de falha é importante em PEGs para definir a escolha ordenada e o predicado de negação. Sem a noção de falha, seria mais difícil expressar essas definições. Uma outra maneira de expressá-las seria através do uso de um quantificador existencial. Nesse caso, dada uma gramática G e uma cadeia x, expressaríamos a noção de falha como $\nexists x' \preceq x \cdot G\ x \overset{\text{CFG}}{\rightsquigarrow} x'$.

Embora o uso de `fail` seja essencial em $\overset{\text{PEG}}{\rightsquigarrow}$, ele não é apropriado nas regras de $\overset{\text{CFG}}{\rightsquigarrow}$, uma vez que em CFGs a escolha não é ordenada e não há predicados. Se tentássemos usar a noção de falha na definição das regras de $\overset{\text{CFG}}{\rightsquigarrow}$, o fato da escolha não ser ordenada resultaria em uma semântica onde um casamento poderia ser bem sucedido e falhar para uma mesma entrada. Nessa semântica, a noção de falha seria de pouca utilidade, pois quando o resultado do casamento de uma dada entrada é `fail`, isso não significa que não há um casamento bem sucedido para essa entrada.

Se analisarmos as regras de $\overset{\text{CFG}}{\rightsquigarrow}$ e de $\overset{\text{PEG}}{\rightsquigarrow}$ podemos ver que as regras *empty*.1, *char*.1 e *choice*.1 de $\overset{\text{CFG}}{\rightsquigarrow}$ são, respectivamente, idênticas às regras *empty*.1, *char*.1 e *ord*.1 de $\overset{\text{PEG}}{\rightsquigarrow}$, e que as regras *var*.1 e *con*.1 de $\overset{\text{CFG}}{\rightsquigarrow}$ diferem das regras *var*.1 e *con*.1 de $\overset{\text{PEG}}{\rightsquigarrow}$ somente pelo fato de que o resultado destas duas últimas regras também pode ser `fail`. Assim, podemos concluir que a principal diferença entre $\overset{\text{CFG}}{\rightsquigarrow}$ e $\overset{\text{PEG}}{\rightsquigarrow}$ está nas regras *choice*.2 e *ord*.2.

Como é possível usar a regra *choice*.2 em um casamento mesmo quando há um casamento bem sucedido através da regra *choice*.1, temos que o casamento usando a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$ é não determinístico. Assim, dada uma gramática G e uma entrada x, o resultado do casamento de x em G usando $\overset{\text{CFG}}{\rightsquigarrow}$ pode não ser único.

Por outro lado, como a regra *ord*.2 só pode ser usada em um casamento quando não há um casamento bem sucedido através da regra *ord*.1, temos que o casamento usando a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$ é determinístico. Assim, dada uma gramática G e uma entrada x, o resultado do casamento de x em G usando $\overset{\text{PEG}}{\rightsquigarrow}$ é sempre o mesmo.

4.3.1 Interpretação de uma Gramática como uma CFG e como uma PEG

Uma vez que a descrição de uma CFG é muito próxima da descrição de uma PEG, dada uma gramática G, podemos interpretá-la usando a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$ ou a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$.

A única diferença entre a descrição de uma CFG e de uma PEG está na notação usada para representar uma escolha. Como norma geral, daqui em diante usaremos a notação $p_1 \mid p_2$ para representar uma escolha nos dois formalismos. Contudo, quando desejarmos enfatizar que uma gramática está sendo interpretada como uma PEG, usaremos a notação $p_1 \,/ p_2$ para representar uma escolha.

Dada uma gramática G, vamos usar a notação $L^{CFG}(G)$ para representar a linguagem que G define quando interpretada como uma CFG. De modo análogo, usaremos a notação $L^{PEG}(G)$ para representar a linguagem que G define quando interpretada como uma PEG.

Dado que a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$ é diferente da semântica de $\overset{\text{PEG}}{\rightsquigarrow}$, geralmente uma gramática G interpretada como uma CFG não define a mesma linguagem que ela define quando interpretada como uma PEG. Contudo, dada uma gramática G que é livre de repetição e livre de predicado, se G casa uma cadeia x quando interpretada como uma PEG, ela também casa x quando interpretada como uma CFG, conforme o lema a seguir:

Lema 4.3.1. *Dada uma gramática G, onde G é livre de repetição e livre de predicado, temos que se $G\ xy \overset{PEG}{\rightsquigarrow} y$ então $G\ xy \overset{CFG}{\rightsquigarrow} y$.*

Demonstração. A prova é por indução na altura da árvore de prova dada pela relação $\overset{\text{PEG}}{\rightsquigarrow}$. O caso interessante da prova é o da expressão de parsing $p_1 \,/\, p_2$. Há duas regras em $\overset{\text{PEG}}{\rightsquigarrow}$ relacionadas a esse caso: *ord*.1 e *ord*.2.

Se a regra *ord*.1 terminou a prova, então $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Assim, pela hipótese de indução, $G[p_1]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$, e pela regra *choice*.1 concluímos que $G[p_1 \mid p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$.

Se a regra *ord*.2 terminou a prova, então $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ e $G[p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Pela hipótese de indução temos que $G[p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$, e pela regra *choice*.2 concluímos que $G[p_1 \mid p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$. □

Dada uma gramática G livre de repetição e livre de predicado, um corolário do lema 4.3.1 é que $L^{PEG}(G) \subseteq L^{CFG}(G)$.

Mais adiante, na seção 4.5, veremos que quando a gramática G é $LL(1)$, então temos que $L^{PEG}(G) = L^{CFG}(G)$.

4.4 Correspondência entre CFGs Lineares à Direita e PEGs

Na formalização usual de CFGs, gramáticas lineares à direita são uma classe de CFGs onde o lado direito de uma produção possui uma das seguintes formas [Hopcroft e Ullman, 1979]:

$$A \rightarrow w\,B \qquad\qquad A \rightarrow w$$

Como podemos ver, em gramáticas lineares à direita um não terminal sempre aparece na extremidade direita de uma concatenação, de modo que nunca temos uma concatenação onde um não terminal precede um símbolo da gramática.

A classe de linguagens descritas por CFGs lineares à direita é igual à classe de linguagens regulares, que são as linguagens descritas por expressões regulares e autômatos finitos.

Dado que todo autômato finito pode ser transformado em uma gramática linear à direita, ao discutirmos a correspondência entre CFGs lineares à direita e PEGs, estamos indiretamente discutindo como transformar autômatos finitos em PEGs, como sugerido em Ierusalimschy [2009].

Na nossa formalização de CFGs, uma gramática G é linear à direita se todas as concatenações da gramática são da forma $(w)\,B$[1] ou w, onde a expressão de parsing w representa a concatenação de vários terminais.

A partir de uma gramática linear à direita $G = (V, T, P, S)$ completa, podemos gerar uma PEG $G' = (V, T, P', S)$ equivalente da seguinte maneira, onde usamos a expressão `.` que é um açúcar sintático que casa qualquer terminal:

para toda produção $A \rightarrow p_1 \mid p_2 \mid \cdots \mid p_n \in P$:

$$P'(A) = matchEnd(p_1) \;/\; matchEnd(p_2) \;/\; \cdots \;/\; matchEnd(p_n)$$

A definição de *matchEnd* é dada a seguir:

$$matchEnd(p) = \begin{cases} (w)\,!. & \text{se } p = w \\ (w)\,B & \text{se } p = (w)\,B \end{cases}$$

Iremos nos referir ao processo anterior de obter a gramática G' a partir da gramática linear à direita G de *transformação ε-End*.

[1]Caso não usássemos parênteses para agrupar a expressão $w = a_1 a_2 \cdots a_n$, teríamos a concatenação $a_1\,(a_2 \cdots a_n B)$.

A gramática G' é completa, pois G não possui produções recursivas à esquerda e a transformação ε-End não introduz produções recursivas à esquerda.

Dado que o casamento da expressão de parsing $!\mathbf{.}$ é bem sucedido apenas quando a entrada é vazia, uma propriedade interessante da PEG G' obtida através da transformação ε-End é que um casamento em G' só é bem sucedido quando toda a entrada é consumida, como nos diz o lema a seguir:

Lema 4.4.1. *Dada uma gramática linear à direita G completa, e uma gramática G' obtida a partir de G usando a transformação ε-End, temos que se $G'[A]\ x \overset{PEG}{\rightsquigarrow} y$ então $y = \varepsilon$.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$.

Dado que G' foi obtida a partir da transformação ε-End, então a forma geral de uma concatenação é $(w)!\mathbf{.}$ ou $(w)\,B$.

No caso de uma concatenação da forma $(w)\,!\mathbf{.}$, seja $x = wy$, temos que $G'[(w)\,!\mathbf{.}]\ wy \overset{\text{PEG}}{\rightsquigarrow} y$, e pela regra *con*.1 temos que $G'[!\mathbf{.}]\ y \overset{\text{PEG}}{\rightsquigarrow} y$. Dado que o casamento de $!\mathbf{.}$ só é bem sucedido para a entrada vazia, então concluímos que $y = \varepsilon$.

No caso de uma concatenação da forma $(w)\,B$, seja $x = wx'$, temos que $G'[(w)\,B]\ wx' \overset{\text{PEG}}{\rightsquigarrow} y$. Pela regra *con*.1 sabemos que $G'[B]\ x' \overset{\text{PEG}}{\rightsquigarrow} y$, e pela hipótese de indução concluímos que $y = \varepsilon$. □

Um corolário que segue do lema anterior é que em uma PEG G', obtida a partir de uma CFG linear à direita, o casamento de uma alternativa de uma escolha só é bem sucedido se toda a cadeia de entrada é consumida.

No caso de PEGs, onde temos uma escolha ordenada, devemos notar que quando as duas alternativas de uma escolha casam toda a entrada, o casamento dessa escolha ocorrerá somente através da primeira alternativa.

Agora, vamos usar o lema anterior para provar a correspondência entre CFGs lineares à direita e PEGs obtidas através da transformação ε-End:

Proposição 4.4.2. *Dada uma gramática linear à direita G completa, e uma gramática G' obtida a partir de G usando a transformação ε-End, temos que $G\ x \overset{CFG}{\rightsquigarrow} \varepsilon$ se e somente se $G'\ x \overset{PEG}{\rightsquigarrow} \varepsilon$.*

Demonstração. ($\Leftarrow$): Vamos provar esta parte por indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$. O caso interessante é o da expressão de parsing da forma $p'_1 \,/\, p'_2$, cujas regras associadas são *ord*.1 e *ord*.2.

Se a regra *ord*.1 foi usada, então pela hipótese de indução temos que $G[p_1]\ x \overset{\text{CFG}}{\rightsquigarrow} \varepsilon$, e pela regra *choice*.1 concluímos que $G[p_1 \mid p_2]\ x \overset{\text{CFG}}{\rightsquigarrow} \varepsilon$.

Se a regra *ord*.2 foi usada, então pela hipótese de indução temos que $G[p_2]\ x \overset{\text{CFG}}{\rightsquigarrow} \varepsilon$, e pela regra *choice*.2 concluímos que $G[p_1 \mid p_2]\ x \overset{\text{CFG}}{\rightsquigarrow} \varepsilon$.

($\Rightarrow$): A prova desta parte é por indução na altura da árvore de prova dada por $\overset{\text{CFG}}{\rightsquigarrow}$. O caso interessante é o da expressão de parsing $p_1 \mid p_2$, cujas regras associadas são *choice*.1 e *choice*.2.

Se a regra *choice*.1 foi usada, pela hipótese de indução temos que $G'[p'_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$, e pela regra *ord*.1 concluímos que $G'[p'_1 \,/\, p'_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$.

Se a regra *choice*.2 foi usada, pela hipótese de indução temos que $G'[p'_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$. Vamos analisar dois casos: quando o casamento de p_1 em $\overset{\text{CFG}}{\rightsquigarrow}$ se relaciona com a cadeia vazia, e quando o casamento de p_1 em $\overset{\text{CFG}}{\rightsquigarrow}$ não se relaciona com a cadeia vazia.

Se p_1 se relaciona com a cadeia vazia em $\overset{\text{CFG}}{\rightsquigarrow}$, então pela hipótese de indução temos que $G'[p'_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$, e pela regra *ord*.1 concluímos que $G'[p'_1 \,/\, p'_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$.

Se p_1 não se relaciona com a cadeia vazia em $\overset{\text{CFG}}{\rightsquigarrow}$, pela contrapositiva temos que p'_1 não se relaciona com a cadeia vazia em $\overset{\text{PEG}}{\rightsquigarrow}$. Como G' é completa, pela contrapositiva do lema 4.4.1 temos que $G'[p'_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Como sabemos que $G'[p'_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$, pela regra *ord*.2 concluímos que $G'[p'_1 \,/\, p'_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$. □

Se assumirmos que uma cadeia $w \in L(G)$ somente no caso em que $G\ w \overset{\text{CFG}}{\rightsquigarrow} \varepsilon$, então temos que $L^{CFG}(G) = L^{PEG}(G')$.

Como mostramos uma correspondência entre gramáticas lineares à direita e PEGs equivalentes, então podemos transformar um autômato finito em uma PEG equivalente.

Dado que podemos transformar expressões regulares em autômatos finitos, também temos uma outra correspondência entre expressões regulares e PEGs.

4.5 Correspondência entre CFGs $LL(1)$ e PEGs

As gramáticas $LL(1)$ são uma classe de CFGs em que um parser *top-down* correspondente pode decidir que produção da gramática deve ser usada olhando apenas o próximo terminal da entrada.

Como discutimos no capítulo 1, acredita-se que quando uma gramática é $LL(1)$, ela define a mesma linguagem quando interpretada como CFG e quando interpretada como PEG. Assim, para tentar comprovar essa conjectura, estudaremos nesta seção a correspondência entre CFGs $LL(1)$ e PEGs.

Na seção 4.5.1, mostramos que quando uma gramática G é $LL(1)$ e não possui expressões ε temos que $L^{CFG}(G) = L^{PEG}(G)$. Em seguida, na seção 4.5.2, provamos que, com uma pequena restrição na ordem das alternativas de

uma escolha, gramáticas $LL(1)$ com expressões ε também definem a mesma linguagem quando interpretadas como CFGs e quando interpretadas como PEGs.

Na discussão a seguir, usaremos o fato de que gramáticas $LL(1)$ não possuem regras recursivas à esquerda [Fischer e LeBlanc, 1991], e portanto são completas.

4.5.1 Gramáticas LL(1) sem Expressões ε

Vamos começar o nosso estudo da correspondência entre CFGs $LL(1)$ sem expressões ε e PEGs revendo a definição da função $FIRST^G$.

Dada uma gramática G, onde G não possui expressões ε, e uma cadeia α de símbolos da gramática, $FIRST^G(\alpha)$ é o conjunto de terminais que podem ser o primeiro terminal de uma cadeia gerada a partir de α na gramática G, como definido a seguir:

$$FIRST^G(\alpha) = \{\, a \in T \mid \alpha \Rightarrow_G^* a\beta \,\}$$

A definição correspondente de $FIRST^G$ na nossa formalização de CFGs, que usa expressões de parsing e a relação $\overset{\text{CFG}}{\rightsquigarrow}$, é dada a seguir:

$$FIRST^G(p) = \{\, a \in T \mid G[p]\ axy \overset{\text{CFG}}{\rightsquigarrow} y \,\}$$

Como a gramática G não possui nenhuma expressão de parsing da forma ε, então nenhuma expressão da gramática casa a cadeia vazia. Assim, podemos definir que uma gramática G é $LL(1)$ se todas as escolhas $p_1 | p_2$ de G satisfazem a seguinte condição:

$$FIRST^G(p_1) \cap FIRST^G(p_2) = \emptyset$$

De acordo com a definição acima, dada uma gramática $LL(1)$ G sem expressões ε, temos que para toda escolha $p_1 \,|\, p_2$ de G o casamento de uma alternativa nunca é bem sucedido para uma entrada quando o casamento da outra alternativa é bem sucedido para essa mesma entrada. Assim, o casamento de p_2 é bem sucedido somente quando o casamento de p_1 não é bem sucedido. Dessa forma, quando o casamento da escolha é bem sucedido, o resultado da interpretação da gramática G como uma CFG ou como uma PEG é o mesmo, como dito pela proposição a seguir:

Proposição 4.5.1. *Dada uma gramática $LL(1)$ G, onde G não possui expressões ε, temos que $G\ xy \overset{\text{CFG}}{\rightsquigarrow} y$ se e somente se $G\ xy \overset{\text{PEG}}{\rightsquigarrow} y$.*

Demonstração. Como o lema 4.3.1 prova a parte $\Leftarrow$ desta proposição, precisamos provar apenas a parte $\Rightarrow$. Vamos provar esta parte por indução na altura da árvore de prova dada por $\overset{\text{CFG}}{\rightsquigarrow}$. O caso interessante da prova é o da expressão de parsing $p_1 \mid p_2$. Há duas regras em $\overset{\text{CFG}}{\rightsquigarrow}$ associadas a esse caso: *choice*.1 e *choice*.2.

Se a regra *choice*.1 terminou a prova, então $G[p_1]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$. Assim, pela hipótese de indução, $G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e pela regra *ord*.1 concluímos que $G[p_1 \,/\, p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$.

Se a regra *choice*.2 terminou a prova, sabemos que $G[p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$. Como G não possui expressões ε, uma expressão de parsing sempre casa uma cadeia não vazia, de modo que a cadeia x é da forma aw.

Dado que p_2 casa aw sabemos que $a \in FIRST^G(p_2)$, e como G é $LL(1)$ então $a \notin FIRST^G(p_1)$. Portanto, seja y' um sufixo de awy, temos que $\nexists y' \cdot G[p_1]\ awy \overset{\text{CFG}}{\rightsquigarrow} y'$. Como gramáticas $LL(1)$ são completas, sabemos que G é completa. Assim, pela contrapositiva do lema 4.3.1, temos que $G[p_1]\ awy \overset{\text{PEG}}{\rightsquigarrow}$ `fail`. Uma vez que $G[p_2]\ awy \overset{\text{CFG}}{\rightsquigarrow} y$, pela hipótese de indução temos que $G[p_2]\ awy \overset{\text{PEG}}{\rightsquigarrow} y$, e pela regra *ord*.2 concluímos que $G[p_1 \,/\, p_2]\ awy \overset{\text{PEG}}{\rightsquigarrow} y$. □

Um corolário da proposição 4.5.1 é que dada uma gramática $LL(1)$ G, onde G não possui expressões ε, temos que $G\ xy \overset{\text{CFG}}{\rightsquigarrow} y$ se e somente se $G\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, ou seja, G define a mesma linguagem quando a interpretamos como uma CFG e quando a interpretamos como uma PEG.

4.5.2 Gramáticas LL(1) com Expressões ε

Se usarmos expressões ε para definir uma gramática, então alguma alternativa de uma escolha pode casar a cadeia vazia, o que torna a proposição 4.5.1 inválida.

Para ver a razão disso, vamos interpretar a expressão de parsing $p_1 \mid p_2$, onde p_1 casa a cadeia vazia. Se usarmos a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$, então o casamento de p_1 sempre é bem sucedido e o casamento de $p_1 \,/\, p_2$ nunca acontece através da regra *ord*.2. Por outro lado, se interpretarmos $p_1 \mid p_2$ usando a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$, continua sendo verdade que o casamento de p_1 sempre é bem sucedido, mas também pode acontecer um casamento bem sucedido da escolha através da regra *choice*.2. Em virtude disso, uma gramática $LL(1)$ com expressões ε pode definir linguagens diferentes quando interpretada como uma CFG e quando interpretada como uma PEG. A gramática $LL(1)$ G descrita a seguir ilustra esse fato:

$$S \to A \mid B \qquad A \to a\,A \mid \varepsilon \qquad B \to b \mid c$$

Se interpretarmos G como uma CFG, então $L^{CFG}(G) = \{a^n, b, c\}$. Por outro lado, se interpretarmos G como uma PEG, temos que $L^{PEG}(G) = \{a^n\}$, pois o casamento do não terminal A sempre é bem sucedido, e portanto o não terminal B nunca casa.

Como veremos mais adiante, se colocarmos uma restrição na ordem das alternativas de uma escolha podemos obter uma gramática que define a mesma linguagem quando interpretada como uma PEG e quando interpretada como uma CFG. Porém, antes de discutir essa restrição precisamos definir alguns conceitos relacionados a uma gramática $LL(1)$ com expressões ε.

Na discussão a seguir, assumiremos que a gramática G possui uma estrutura BNF. Como discutido na seção 4.2, se G é uma gramática com estrutura BNF, então uma escolha $p_1 \mid p_2$ nunca é uma subexpressão de uma concatenação, de modo que ela ou está associada a um não terminal A, ou ela é uma subexpressão de uma escolha que está associada a um não terminal A. Além disso, quando G possui estrutura BNF temos que $p_S = S$, ou seja, a expressão de parsing inicial de G é sempre um não terminal S.

Vamos começar atualizando a definição de $FIRST^G$:

$$FIRST^G(p) = \{\, a \in T \mid G[p]\ axy \overset{\text{CFG}}{\rightsquigarrow} y \,\} \cup matchEmpty(p)$$

$$matchEmpty(p) = \begin{cases} \{\varepsilon\} & \text{se } G[p]\ x \overset{\text{CFG}}{\rightsquigarrow} x \\ \emptyset & \text{caso contrário} \end{cases}$$

Também precisamos definir a função $FOLLOW^G$, que nos fornece o conjunto de terminais que podem seguir imediatamente um não terminal. Para definir $FOLLOW^G$ usaremos o símbolo \$ como um marcador de fim da entrada, onde $\$ \notin T$. A seguir, definimos quando um terminal a pertence ao conjunto $FOLLOW^G$ de um não terminal A:

> Dada uma gramática $G = (V, T, P, S)$ temos que $a \in FOLLOW^G(A)$ se há uma árvore de prova $G\ w\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ com uma subárvore $G[A]\ xay \overset{\text{CFG}}{\rightsquigarrow} ay$.

Para computar $FOLLOW^G$, vamos analisar a forma geral da árvore de prova dada por $\overset{\text{CFG}}{\rightsquigarrow}$, a qual é mostrada na figura 4.2, e ver como a definição anterior de $FOLLOW^G$ funciona. Na figura 4.2, cada símbolo p_i representa uma concatenação e cada símbolo p_{ij} representa a cadeia vazia, um terminal, ou um não terminal. Nessa mesma figura, o símbolo b_i representa um terminal ou a cadeia vazia, e o símbolo x_i representa uma cadeia (possivelmente vazia).

Na parte de baixo da árvore de prova está o casamento da expressão de parsing inicial S. Pela definição de $FOLLOW^G$ sabemos que $\$ \in$

$$\dfrac{\dfrac{\dfrac{\dots \qquad \dots \qquad \dots \qquad \dots}{G[p_{i1}]\ b_1x_1\cdots b_nx_n\$ \overset{\text{CFG}}{\rightsquigarrow} b_2x_2\cdots b_nx_n\$ \quad G[p_{i2}]\ b_2x_2\cdots b_nx_n\$ \overset{\text{CFG}}{\rightsquigarrow} b_3x_3\cdots b_nx_n\$ \quad \dots \quad G[p_{in}]\ b_nx_n\$ \overset{\text{CFG}}{\rightsquigarrow} \$}}{G[p_{i1}\, p_{i2}\, \cdots\, p_{in}]\ b_1x_1\cdots b_nx_n\$ \overset{\text{CFG}}{\rightsquigarrow} \$}}{G[p_1 \mid p_2 \mid \cdots \mid p_m\ ]\ b_1x_1\cdots b_nx_n\$ \overset{\text{CFG}}{\rightsquigarrow} \$}}{G[S]\ b_1x_1\cdots b_nx_n\$ \overset{\text{CFG}}{\rightsquigarrow} \$}$$

Figura 4.2: Forma Geral da Árvore de Prova Quando a Gramática G Possui Estrutura BNF

$FOLLOW^G(S)$. No antecedente do casamento de S temos o casamento da expressão de parsing relacionada a S. Geralmente essa expressão de parsing é uma escolha, mas ela também pode ser uma concatenação, um símbolo da gramática, ou ε. Nestes três últimos casos, dizemos que temos uma *escolha unitária*.

No antecedente do casamento da escolha temos o casamento de uma das alternativas da escolha. Geralmente esse é o casamento de uma concatenação, mas também pode ser o casamento de um símbolo da gramática ou de ε.

Finalmente, temos as árvores de prova associadas com cada expressão de parsing da concatenação. A expressão de parsing p_{i1} casa a cadeia b_1x_1 (que pode ser vazia), a expressão de parsing p_{i2} casa a cadeia b_2x_2 (que pode ser vazia), e assim por diante.

Se considerarmos que p_{i1} é um não terminal A e que b_2 é um terminal, então pela definição de $FOLLOW^G$ sabemos que $b_2 \in FOLLOW^G(A)$. Como p_{i2} casa b_2x_2, também sabemos que $b_2 \in FIRST^G(p_{i2})$.

Se b_2x_2 é uma cadeia vazia e b_3 é um terminal, então $b_3 \in FOLLOW^G(A)$. Caso todas as cadeias b_jx_j, onde $1 < j \leq n$, sejam vazias, então $\$ \in FOLLOW^G(A)$.

Com base nessa descrição, podemos computar $FOLLOW^G$ para todos os não terminais $A \in V$ seguindo o algoritmo descrito na figura 4.3, onde X_{ij} pode ser ε, um terminal ou um não terminal, e a repetição converge quando nenhum terminal é adicionado a nenhum conjunto $FOLLOW^G(A)$.

No algoritmo para computar $FOLLOW^G$ usamos o operador $\otimes$, cuja definição é dada a seguir:

$$X \otimes Y = \begin{cases} X & \text{se } \varepsilon \notin X \\ (X - \{\varepsilon\}) \cup Y & \text{se } \varepsilon \in X \end{cases}$$

Vamos agora voltar a nossa atenção para a discussão sobre a correspondência entre CFGs $LL(1)$ e PEGs, onde usaremos $FIRST^G$ e $FOLLOW^G$ para definir o que é uma gramática $LL(1)$ com expressões ε.

para todo $A \in V$:
  $FOLLOW^G(A) \;=\; \emptyset$
$FOLLOW^G(S) = \{\,\$\,\}$
repita
  para toda produção $A \to X_{11} \cdots X_{1\,n_1} \mid \cdots \mid X_{m1} \cdots X_{m\,n_m}$:
    para toda concatenação $X_{i1} \cdots X_{i\,n_i}$:
      para $j = 1$ até n_i :
        se $X_{i\,j} \in V$ então :
          $FOLLOW^G(X_{i\,j}) \;=\; FOLLOW^G(X_{i\,j}) \;\cup$
                $(FIRST^G(X_{i\,j+1} \cdots X_{i\,n_i}) \;\otimes\; FOLLOW^G(A))$
até convergir

Figura 4.3: Algoritmo para Computar $FOLLOW^G$ dada uma Gramática $LL(1)$ G com Estrutura BNF

Em gramáticas $LL(1)$ as alternativas de uma escolha casam cadeias que começam com terminais diferentes, e no máximo uma das alternativas pode casar a cadeia vazia. Como vimos antes, dada uma escolha $p_1 \,|\, p_2$ onde p_1 casa a cadeia vazia, se interpretarmos essa escolha usando a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$ o casamento de p_1 sempre é bem sucedido e portanto p_2 nunca casa. Para resolver esse problema, definimos na seção 4.2 que quando uma gramática possui estrutura BNF a alternativa de uma escolha que não casa a cadeia vazia sempre precede a alternativa que casa a cadeia vazia. Assim, em gramáticas $LL(1)$ que possuem estrutura BNF, dada uma escolha $p_1 \,|\, p_2$, sabemos que somente a alternativa p_2 pode casar a cadeia vazia. Em virtude disso, a alternativa p_1 deve casar pelo menos um terminal para que o seu casamento seja bem sucedido.

Essa restrição afeta o resultado do casamento de uma escolha somente quando a interpretamos como uma PEG, pois em CFGs a ordem das alternativas de uma escolha não é relevante.

Dada então uma gramática G, onde G possui estrutura BNF, dizemos que G é $LL(1)$ se para toda produção $A \to p$ de G, seja $p_1 \,|\, p_2$ uma subexpressão de p, temos que as seguintes condições são satisfeitas:

- $FIRST^G(p_1) \;\cap\; FIRST^G(p_2) \;=\; \emptyset$
- se $\varepsilon \in FIRST^G(p_2)$ então $FIRST^G(p_1) \;\cap\; FOLLOW^G(A) \;=\; \emptyset$

Quando G é $LL(1)$, queremos mostrar que $L^{CFG}(G) = L^{PEG}(G)$. Para alcançar esse objetivo, vamos definir a relação $\overset{\text{LL(1)}}{\rightsquigarrow}$ que, assim como $\overset{\text{CFG}}{\rightsquigarrow}$,

Cadeia Vazia $\dfrac{}{G[\varepsilon]\ x \overset{\text{LL(1)}}{\rightsquigarrow} x}\,(\textbf{empty.1})$ **Terminal** $\dfrac{}{G[a]\ ax \overset{\text{LL(1)}}{\rightsquigarrow} x}\,(\textbf{char.1})$

Variável $\dfrac{G[P(A)]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y}{G[A]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y}\,(\textbf{var.1})$ **Concatenação** $\dfrac{G[p_1]\ xyz \overset{\text{LL(1)}}{\rightsquigarrow} yz \quad G[p_2]\ yz \overset{\text{LL(1)}}{\rightsquigarrow} z}{G[p_1\,p_2]\ xyz \overset{\text{LL(1)}}{\rightsquigarrow} z}\,(\textbf{con.1})$

Escolha $\dfrac{G[p_1]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y}\,(\textbf{choice.1})$

$$\frac{G[p_2]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y},\ x = \varepsilon \Rightarrow \nexists\, y' \cdot G[p_1]\ xy \overset{\text{LL(1)}}{\rightsquigarrow} y'\,(\textbf{choice}_{\textbf{LL(1)}}\textbf{.2})$$

Figura 4.4: Definição da Relação $\overset{\text{LL(1)}}{\rightsquigarrow}$ Usando Semântica Natural

relaciona uma gramática G e uma entrada xy com um sufixo y da entrada. Na figura 4.4, apresentamos a definição de $\overset{\text{LL(1)}}{\rightsquigarrow}$ usando semântica natural.

Como podemos ver, as regras de $\overset{\text{LL(1)}}{\rightsquigarrow}$ são iguais às regras de $\overset{\text{CFG}}{\rightsquigarrow}$, com exceção da regra $choice_{LL(1)}.2$, que corresponde à regra $choice.2$ de $\overset{\text{CFG}}{\rightsquigarrow}$. Na semântica de $\overset{\text{LL(1)}}{\rightsquigarrow}$, a segunda alternativa de uma escolha só casa um prefixo vazio da entrada se o casamento da primeira alternativa da escolha não é bem sucedido para essa entrada.

Quando temos uma gramática $LL(1)$, podemos estabelecer a seguinte correspondência entre as relações $\overset{\text{CFG}}{\rightsquigarrow}$ e $\overset{\text{LL(1)}}{\rightsquigarrow}$:

Lema 4.5.2. *Dada uma gramática $LL(1)$ G, temos que se $G[A]\ xay \overset{CFG}{\rightsquigarrow} ay$, onde $a \in FOLLOW^G(A)$, então $G[A]\ xay \overset{LL(1)}{\rightsquigarrow} ay$.*

Demonstração. A prova é por indução na altura da árvore de prova dada por $\overset{\text{CFG}}{\rightsquigarrow}$. O caso interessante é quando a expressão de parsing associada ao não terminal A é uma escolha $p_1 \mid p_2$ e a regra $choice.2$ é usada. Nesse caso, temos a seguinte árvore de prova:

$$\frac{\dfrac{G[p_2]\ xay \overset{\text{CFG}}{\rightsquigarrow} ay}{G[p_1 \mid p_2]\ xay \overset{\text{CFG}}{\rightsquigarrow} ay}\,(\textbf{choice.2})}{G[A]\ xay \overset{\text{CFG}}{\rightsquigarrow} ay}\,(\textbf{var.1})$$

Dado o casamento $G[p_2]\ xay \overset{\text{CFG}}{\rightsquigarrow} ay$, temos que p_2 pode ter casado a cadeia vazia ou uma cadeia não vazia.

Se p_2 casou a cadeia vazia, então $x = \varepsilon$ e temos que $G[p_2]\ ay \overset{\text{CFG}}{\rightsquigarrow} ay$, onde $\varepsilon \in FIRST^G(p_2)$. Dado que $a \in FOLLOW^G(A)$ e que G é $LL(1)$, temos

que $FIRST^G(p_1) \cap FOLLOW^G(A) = \emptyset$, de modo que $a \notin FIRST^G(p_1)$ e o casamento de p_1 não é bem sucedido usando $\overset{\text{CFG}}{\rightsquigarrow}$. Logo, temos que o casamento de p_1 não é bem sucedido usando $\overset{\text{LL(1)}}{\rightsquigarrow}$. Pela hipótese de indução temos que $G[p_2]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, e pela regra $choice_{LL(1)}.2$ temos que $G[p_1 \mid p_2]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$. Portanto, pela regra $var.1$, concluímos que $G[A]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$.

Se p_2 casou uma cadeia não vazia, então pela hipótese de indução $G[p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$. Pela regra $choice_{LL(1)}.2$ temos que $G[p_1 \mid p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$ e pela regra $var.1$ concluímos que $G[A]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$. □

Poderíamos provar também a outra direção do lema anterior. Contudo, embora seja fácil provar que todo casamento bem sucedido através da relação $\overset{\text{LL(1)}}{\rightsquigarrow}$ também é bem sucedido através da relação $\overset{\text{CFG}}{\rightsquigarrow}$, não seria tão fácil assim mostrar que a condição $a \in FOLLOW^G(A)$ do lema anterior é satisfeita no casamento através de $\overset{\text{CFG}}{\rightsquigarrow}$ dado o casamento através de $\overset{\text{LL(1)}}{\rightsquigarrow}$. Como não precisamos da outra direção do lema 4.5.2 para provar a correspondência entre CFGs $LL(1)$ e PEGs, não iremos mostrá-la.

Após estabelecer a correspondência entre $\overset{\text{CFG}}{\rightsquigarrow}$ e $\overset{\text{LL(1)}}{\rightsquigarrow}$, vamos definir o seguinte lema a respeito do casamento em uma gramática $LL(1)$ usando a semântica de $\overset{\text{LL(1)}}{\rightsquigarrow}$ e a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$:

Lema 4.5.3. *Dada uma gramática $LL(1)$ G, então $G[A]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, onde $a \in FOLLOW^G(A)$, se e somente se $G[A]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$.*

Demonstração. ($\Rightarrow$): Vamos provar esta parte por indução na altura da árvore de prova dada por $\overset{\text{LL(1)}}{\rightsquigarrow}$. O caso interessante é quando a expressão de parsing associada ao não terminal A é uma escolha $p_1 \mid p_2$. Existem duas regras em $\overset{\text{LL(1)}}{\rightsquigarrow}$ relacionadas a esse caso: $choice.1$ e $choice_{LL(1)}.2$.

Se a regra $choice.1$ foi usada, então $G[p_1]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, e pela hipótese de indução $G[p_1]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$. Assim, pela regra $ord.1$, temos que $G[p_1 \,/ p_2]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$ e pela regra $var.1$ concluímos que $G[A]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$.

Se a regra $choice_{LL(1)}.2$ foi usada, então $G[p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, onde p_2 pode ter casado a cadeia vazia ou uma cadeia não vazia.

Se p_2 casou a cadeia vazia, então $x = \varepsilon$. Dado que a gramática G é $LL(1)$, temos que $FOLLOW^G(A) \cap FIRST^G(p_1) = \emptyset$, e como $a \in FOLLOW^G(A)$ então $a \notin FIRST^G(p_1)$. Portanto, o casamento de p_1 não é bem sucedido em $\overset{\text{CFG}}{\rightsquigarrow}$. Como G é completa, pela contrapositiva do lema 4.3.1 temos que $G[p_1]\ ay \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Dado que $G[p_2]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, pela hipótese de indução temos que $G[p_2]\ ay \overset{\text{PEG}}{\rightsquigarrow} ay$, e pela regra $ord.2$ temos que $G[p_1 \,/\, p_2]\ ay \overset{\text{PEG}}{\rightsquigarrow} ay$. Assim, pela regra $var.1$, concluímos que $G[A]\ ay \overset{\text{PEG}}{\rightsquigarrow} ay$.

Se p_2 casou uma cadeia não vazia bw, então $b \in FIRST^G(p_2)$. Pela definição de gramática $LL(1)$ sabemos que $FIRST^G(p_1) \cap FIRST^G(p_2) = \emptyset$,

de modo que $b \notin FIRST^G(p_1)$. Como p_1 não casa a cadeia vazia, temos que o casamento de p_1 não é bem sucedido em $\overset{\text{CFG}}{\rightsquigarrow}$. Uma vez que gramáticas $LL(1)$ são completas, pela contrapositiva do lema 4.3.1 temos que $G[p_1]\ xay \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Dado que $G[p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, pela hipótese de indução temos que $G[p_2]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$, e pela regra $ord.2$ temos que $G[p_1 \,/\, p_2]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$. Assim, pela regra $var.1$ concluímos que $G[A]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$.

($\Leftarrow$): Vamos provar esta parte por indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$. O caso interessante é quando a expressão de parsing associada ao não terminal A é da forma $p_1 \,/\, p_2$. As duas regras associadas a esse caso são $ord.1$ e $ord.2$.

Se a regra $ord.1$ foi usada, então $G[p_1]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$. Pela hipótese de indução $G[p_1]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, e pela regra $choice.1$ temos que $G[p_1 \,|\, p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$. Portanto, pela regra $var.1$ concluímos que $G[A]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$, onde $a \in FOLLOW^G(A)$.

Se a regra $ord.2$ foi usada, então $G[p_1]\ xay \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ e $G[p_2]\ xay \overset{\text{PEG}}{\rightsquigarrow} ay$, onde p_2 pode ter casado a cadeia vazia ou uma cadeia não vazia.

Se p_2 casou a cadeia vazia, pela hipótese de indução temos que $G[p_2]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$. Pela regra $ord.2$ sabemos que $G[p_1]\ ay \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, e pela contrapositiva temos que o casamento de p_1 usando $\overset{\text{LL(1)}}{\rightsquigarrow}$ não é bem sucedido. Assim, pela regra $choice_{LL(1)}.2$, temos que $G[p_1 \,|\, p_2]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, e pela regra $var.1$ concluímos que $G[A]\ ay \overset{\text{LL(1)}}{\rightsquigarrow} ay$ onde $a \in FOLLOW^G(A)$.

Se p_2 casou uma cadeia não vazia, pela hipótese de indução temos que $G[p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$, e pela regra $choice_{LL(1)}.2$ temos que $G[p_1 \,|\, p_2]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$. Assim, pela regra $var.1$, concluímos que $G[A]\ xay \overset{\text{LL(1)}}{\rightsquigarrow} ay$ onde $a \in FOLLOW^G(A)$. □

Dada uma gramática $LL(1)$ G, a seguinte proposição afirma que $L^{CFG}(G) = L^{PEG}(G)$:

Proposição 4.5.4. *Dada uma gramática $LL(1)$ $G = (V, T, P, S)$, temos que $G\ x\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ se e somente se $G\ x\$ \overset{\text{PEG}}{\rightsquigarrow} \$$.*

Demonstração. ($\Rightarrow$): Dado que $G\ x\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ e que $\$ \in FOLLOW^G(S)$, pelo lema 4.5.2 temos que $G\ x\$ \overset{\text{LL(1)}}{\rightsquigarrow} \$$, e pelo lema 4.5.3 concluímos que $G\ x\$ \overset{\text{PEG}}{\rightsquigarrow} \$$.

($\Leftarrow$): Dado que $G\ x\$ \overset{\text{PEG}}{\rightsquigarrow} \$$, pelo lema 4.3.1 concluímos que $G\ x\$ \overset{\text{CFG}}{\rightsquigarrow} \$$. □

4.6 Correspondência entre CFGs $LL(k)$-Forte e PEGs

As gramáticas $LL(k)$-forte são uma classe de CFGs onde um parser top-down correspondente pode decidir que produção da gramática deve ser usada olhando somente os próximos k terminais da entrada.

A classe de linguagens descrita pelas gramáticas $LL(k)$-forte é uma subclasse da classe de linguagens descrita pelas gramáticas $LL(k)$. Uma gramática G é $LL(k)$ se um parser top-down correspondente pode decidir que produção da gramática usar com base nos próximos k terminais da entrada e também em quais produções da gramáticas foram usadas anteriormente, ou seja, um parser $LL(k)$ precisa de um contexto para decidir corretamente que produção da gramática deve ser usada [Fischer e LeBlanc, 1991]. No caso específico em que $k = 1$ temos que a classe de linguagens descrita por gramáticas $LL(1)$-forte é igual à classe de linguagens descrita por gramáticas $LL(1)$ [Fischer e LeBlanc, 1991, Grune e Jacobs, 2006].

Nesta seção iremos mostrar que a partir de uma CFG $LL(k)$-forte podemos gerar uma PEG equivalente que possui a mesma estrutura da CFG.

De modo similar ao que fizemos na seção 4.5.2, iremos assumir que a gramática G possui uma estrutura BNF, pois assim cada escolha é uma subexpressão de uma escolha que está associada a algum não terminal A, e a expressão de parsing inicial da gramática é um não terminal S.

Antes de discutir a correspondência entre CFGs $LL(k)$-forte e PEGs, precisamos definir as funções $FIRST_k^G$ e $FOLLOW_k^G$, que estendem, respectivamente, as funções $FIRST^G$ e $FOLLOW^G$.

Vamos começar definindo a função $FIRST_k^G$ como a seguir:

$$FIRST_k^G(p) \;=\; \{\, take_k(x) \mid G[p] \;\; xy \overset{\text{CFG}}{\rightsquigarrow} y \,\}$$

Na definição anterior usamos a função $take_k$, que recebe uma cadeia e retorna um prefixo dessa cadeia com comprimento menor ou igual a k, como descrito a seguir:

$$take_k(xy) = \begin{cases} xy & \text{se } |xy| \leq k \\ x & \text{se } |xy| > k, \text{ onde } |x| = k \end{cases}$$

Agora, iremos definir a função $FOLLOW_k^G$, que fornece o conjunto de cadeias de comprimento k que podem seguir um não terminal.

Vamos usar novamente o símbolo \$ para indicar o fim da entrada, onde $\$ \notin T$. Quando discutimos gramáticas $LL(1)$, colocamos um único símbolo \$ no final da entrada. Porém, no caso de gramáticas $LL(k)$-forte colocaremos k

para todo $A \in V$:
  $FOLLOW_k^G(A) \;=\; \emptyset$
$FOLLOW_k^G(S) = \{\$^k\}$
repita
  para toda produção $A \to X_{11} \cdots X_{1\,n_1} \mid \cdots \mid X_{m1} \cdots X_{m\,n_m}$:
    para toda concatenação $X_{i1} \cdots X_{i\,n_i}$:
      para $j = 1$ até n_i :
        se $X_{i\,j} \in V$ então :
          $FOLLOW_k^G(X_{i\,j}) \;=\; FOLLOW_k^G(X_{i\,j}) \;\cup$
                $(FIRST_k^G(X_{i\,j+1} \cdots X_{i\,n_i}) \;\otimes_k\; FOLLOW_k^G(A))$
até convergir

Figura 4.5: Algoritmo para Computar $FOLLOW_k^G$ dada uma Gramática $LL(k)$-Forte G com Estrutura BNF

símbolos \$ no final da entrada, de modo que cada não terminal é seguido por pelo menos k terminais. Usaremos a notação $\$^k$ para indicar uma cadeia de k símbolos \$. A seguir, definimos quando uma cadeia de terminais pertence ao conjunto $FOLLOW_k^G$ de um não terminal A:

> Dada uma gramática $G = (V, T, P, S)$, temos que $take_k(y) \in FOLLOW_k^G(A)$ se há uma árvore de prova $G\ w\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ com uma subárvore $G[A]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$.

Dada a definição anterior, o lema 4.2.1 nos diz que em toda subárvore $G[A]\ xy \overset{\text{CFG}}{\rightsquigarrow} y$ temos que $|y| \geq k$, o que implica que todas as cadeias em $FOLLOW_k^G(A)$ possuem comprimento k. Assim, nenhuma cadeia de $FOLLOW_k^G(A)$ é um prefixo de outra cadeia desse conjunto. Esse fato será útil mais adiante, ao gerarmos uma expressão de parsing para casar os elementos de $FOLLOW_k^G(A)$.

Podemos computar o conjunto $FOLLOW_k^G$ para todos os não terminais $A \in V$ usando o algoritmo descrito na figura 4.5, que é similar ao algoritmo da figura 4.3, onde $X_{i\,j}$ pode ser ε, um terminal ou um não terminal, e a repetição converge quando não é possível adicionar nenhuma cadeia a nenhum conjunto $FOLLOW_k^G(A)$.

O operador $\otimes_k$ usado na figura 4.5 é definido a seguir:

$$X \otimes_k Y \;=\; \{\, take_k(w) \mid w \in XY \,\}$$

Após definirmos as funções $FIRST_k^G$ e $FOLLOW_k^G$, vamos definir o que é uma gramática $LL(k)$-forte. Dada uma gramática G, onde G possui estrutura BNF, dizemos que G é $LL(k)$-forte se todas as escolhas $p_1 \mid p_2$ associadas a um não terminal A de G satisfazem a seguinte condição:

$$(FIRST_k^G(p_1) \otimes_k FOLLOW_k^G(A)) \cap (FIRST_k^G(p_2) \otimes_k FOLLOW_k^G(A)) = \emptyset$$

A seguir, temos o exemplo de uma gramática G que é $LL(2)$-forte:

$$S \rightarrow A \mid B \qquad A \rightarrow a\,b \mid C \qquad B \rightarrow a \mid C\,d \qquad C \rightarrow c$$

Os conjuntos $FIRST_2^G$ e $FOLLOW_2^G$ associados aos não terminais da gramática G são os seguintes:

- $FIRST_2^G(S) = \{a,\, ab,\, c,\, cd\}$
- $FIRST_2^G(A) = \{ab,\, c\}$
- $FIRST_2^G(B) = \{a,\, cd\}$
- $FIRST_2^G(C) = \{c\}$
- $FOLLOW_2^G(S) = FOLLOW_2^G(A) = FOLLOW_2^G(B) = \{\$\$\}$
- $FOLLOW_2^G(C) = \{d\$,\, \$\$\}$

Se interpretarmos G como uma CFG, então $L^{CFG}(G) = \{\, a,\, ab,\, c,\, cd \,\}$. Por outro lado, se interpretarmos G como uma PEG temos que $L^{PEG}(G) = \{\, a,\, ab,\, c \,\}$. Podemos ver que $cd \notin L^{PEG}(G)$, uma vez que o casamento de A sempre é bem sucedido quando a entrada possui prefixo c. Portanto, a gramática G descreve linguagens diferentes quando interpretada como uma CFG e quando interpretada como uma PEG.

Ao contrário de gramáticas $LL(1)$ com expressões ε, uma simples restrição na ordem das alternativas de uma escolha não torna a linguagem da CFG igual à linguagem da PEG. Se mudarmos a escolha $A \mid B$ associada a S para $B \mid A$, o resultado seria que $L^{PEG}(G) = \{\, a,\, c,\, cd \,\}$. Nesse caso, temos que $ab \notin L^{PEG}(G)$, uma vez que o casamento de B sempre é bem sucedido quando a entrada possui prefixo a.

Dada uma gramática $LL(k)$-forte G, como $L^{CFG}(G) \neq L^{PEG}(G)$, para estabelecer a correspondência entre CFGs $LL(k)$-forte e PEGs iremos gerar uma nova gramática G' a partir da gramática G, onde $L^{CFG}(G) = L^{PEG}(G')$.

4.6.1 Transformação de uma CFG $LL(k)$-forte em uma PEG Equivalente

Para transformar uma CFG $LL(k)$-forte em uma PEG equivalente iremos usar, como mencionado na seção 2.3, a expressão de parsing $\&p$ como um açúcar sintático para a expressão $!!p$. Na discussão a seguir, também usaremos o fato de que gramáticas $LL(k)$-forte não possuem regras recursivas à esquerda, e portanto são completas.

Dada uma cadeia de terminais w, usaremos p_w para representar a expressão de parsing correspondente, onde p_w casa w apenas. Como p_w não possui não terminais, o casamento de p é independente da gramática G.

Para transformar uma CFG $LL(k)$-forte em uma PEG equivalente vamos precisar definir a função auxiliar *set2choice*, que recebe um conjunto Z, cujos elementos são cadeias de terminais, e nos dá uma expressão de parsing que casa os elementos de Z:

$$set2choice\,(Z) \;=\; p_{z_1} \;\mid\; p_{z_2} \;\mid\; \cdots \;\mid\; p_{z_n}\text{ , onde } z_i \in Z$$

A função *set2choice* nos dá uma expressão de parsing que não possui não terminais, de modo que o casamento dessa expressão é independente da gramática. No caso da gramática $LL(2)$-forte G que apresentamos anteriormente, a função *set2choice* dos daria as seguintes expressões de parsing:

$$set2choice(S) = set2choice(A) = set2choice(B) = \$\,\$ \qquad set2choice(C) = d\,\$\,/\$\,\$$$

Dada um não terminal A, como todos os elementos de $FOLLOW_k^G(A)$ possuem o mesmo comprimento, nenhum elemento pode ser prefixo de outro. Em virtude disso, a escolha resultante de $set2choice(FOLLOW_k^G(A))$ é uma escolha disjunta. Dado um não terminal A, usaremos $\phi(A)$ para representar $set2choice(FOLLOW_k^G(A))$.

A partir de uma gramática $LL(k)$-forte $G = (V,\, T,\, P,\, S)$, podemos gerar uma PEG $G' = (V,\, T,\, P',\, S)$ equivalente da seguinte maneira:

$$\begin{array}{c}\text{para toda produção } A \to p_1 \mid p_2 \mid \cdots \mid p_n \in P : \\ P'(A) \;=\; p_1\,\&\phi(A) \;/\; p_2\,\&\phi(A) \;/\; \cdots \;/\; p_n\,\&\phi(A)\end{array}$$

Acima, dado um não terminal A, usamos $\&\phi(A)$ para testar se é possível casar alguma cadeia $x \in FOLLOW_k^G(A)$ depois de casar uma alternativa da escolha associada a A. Iremos nos referir ao método anterior de obter a gramática G' a partir da gramática G de *transformação $LL(k)$-PEG*. A gramática G' obtida dessa forma é completa, pois G é completa a transformação $LL(k)$-PEG

Cadeia Vazia $\dfrac{}{G[\varepsilon]\ x \overset{\text{LL(k)}}{\rightsquigarrow} x}\ (\mathbf{empty.1})$ **Terminal** $\dfrac{}{G[a]\ ax \overset{\text{LL(k)}}{\rightsquigarrow} x}\ (\mathbf{char.1})$

Variável $\dfrac{G[P(A)]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y}{G[A]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y}\ ,\ take_k(y) \in FOLLOW_k^G(A)\ (\mathbf{var_{LL(k)}.1})$

Concatenação $\dfrac{G[p_1]\ xyz \overset{\text{LL(k)}}{\rightsquigarrow} yz \quad G[p_2]\ yz \overset{\text{LL(k)}}{\rightsquigarrow} z}{G[p_1\, p_2]\ xyz \overset{\text{LL(k)}}{\rightsquigarrow} z}\ (\mathbf{con.1})$

Escolha $\dfrac{G[p_1]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y}\ (\mathbf{choice.1})$ $\dfrac{G[p_2]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y}\ (\mathbf{choice.2})$

Figura 4.6: Definição da Relação $\overset{\text{LL(k)}}{\rightsquigarrow}$ Usando Semântica Natural

não introduz produções recursivas à esquerda, nem expressões de parsing da forma p^* onde p cada a cadeia vazia.

No caso da gramática $LL(2)$-forte G que apresentamos anteriormente, a transformação $LL(k)$-PEG nos daria a seguinte gramática G':

$$S \rightarrow A\,\&(\$\,\$) \,/\, B\,\&(\$\,\$) \qquad A \rightarrow a\,b\,\&(\$\,\$) \,/\, C\,\&(\$\,\$)$$
$$B \rightarrow a\,\&(\$\,\$) \,/\, C\,d\,\&(\$\,\$) \qquad C \rightarrow c\,\&(d\,\$ \,/\, \$\,\$)$$

Para provar a equivalência entre uma CFG $LL(k)$-forte G e a PEG G' obtida através da transformação $LL(k)$-PEG, vamos definir uma nova relação $\overset{\text{LL(k)}}{\rightsquigarrow}$ que, assim como $\overset{\text{CFG}}{\rightsquigarrow}$, relaciona uma gramática G e uma entrada xy com um sufixo y da entrada. Na figura 4.6, apresentamos a definição de $\overset{\text{LL(k)}}{\rightsquigarrow}$ usando semântica natural.

Assim como as regras de $\overset{\text{LL(1)}}{\rightsquigarrow}$, quase todas as regras de $\overset{\text{LL(k)}}{\rightsquigarrow}$ são iguais às regras de $\overset{\text{CFG}}{\rightsquigarrow}$. No caso de $\overset{\text{LL(k)}}{\rightsquigarrow}$, a única regra diferente é a regra $var_{LL(k)}.1$, que trata do casamento de um não terminal. Na semântica de $\overset{\text{LL(k)}}{\rightsquigarrow}$, o casamento de um não terminal A é bem sucedido para uma entrada xy somente quando A casa um prefixo x da entrada e $take_k(y) \in FOLLOW_k^G(A)$.

No caso da relação $\overset{\text{LL(1)}}{\rightsquigarrow}$, poderíamos ter definido uma regra $var_{LL(1)}.1$, análoga à regra $var_{LL(k)}.1$, ao invés de ter definido a regra $choice_{LL(1)}.2$. Contudo, se definíssemos $var_{LL(1)}.1$, para provar a equivalência entre CFGs $LL(1)$ e PEGs teríamos que gerar uma PEG a partir de uma gramática $LL(1)$, de modo similar ao que estamos fazendo para gramáticas $LL(k)$-forte. Embora a definição da regra $choice_{LL(1)}.2$ não seja tão simples quanto o que seria a definição da regra $var_{LL(1)}.1$, o uso de $choice_{LL(1)}.2$ nos possibilitou interpretar

gramáticas $LL(1)$ G como CFGs e PEGs e mostrar que $L^{CFG}(G) = L^{PEG}(G)$.

Dadas as definições de $\overset{\text{CFG}}{\rightsquigarrow}$ e de $\overset{\text{LL(k)}}{\rightsquigarrow}$, podemos estabelecer a seguinte correspondência entre essas relações quando a gramática G é $LL(k)$-forte:

Lema 4.6.1. *Dada uma gramática $LL(k)$-forte G, temos que $G[A]\ xy \overset{CFG}{\rightsquigarrow} y$, onde $take_k(y) \in FOLLOW_k^G(A)$, se e somente se $G[A]\ xy \overset{LL(k)}{\rightsquigarrow} y$.*

Demonstração. Trivial, pois quando $G[A]\ \ xy \overset{\text{CFG}}{\rightsquigarrow} y$ com $take_k(y) \in FOLLOW_k^G(A)$, a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$ torna-se idêntica à semântica de $\overset{\text{LL(k)}}{\rightsquigarrow}$. □

Após estabelecer a correspondência entre $\overset{\text{CFG}}{\rightsquigarrow}$ e $\overset{\text{LL(k)}}{\rightsquigarrow}$, vamos definir o seguinte lema sobre o casamento de expressões de parsing que possuem a forma restrita das expressões resultantes da função *set2choice*:

Lema 4.6.2. *Dada uma expressão de parsing p, onde p só possui subexpressões da forma ε, a, $p_1\,p_2$ e $p_1 \mid p_2$, e onde todas as subexpressões da forma $p_1 \mid p_2$ possuem alternativas disjuntas, temos que $p\ xy \overset{LL(k)}{\rightsquigarrow} y$ se e somente se $p\ xy \overset{PEG}{\rightsquigarrow} y$, para quaisquer gramáticas G e G'.*

Demonstração. A parte $\Rightarrow$ usa indução na altura da árvore de prova dada por $\overset{\text{LL(k)}}{\rightsquigarrow}$, enquanto que a parte $\Leftarrow$ usa indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$.

Dado que as alternativas de uma escolha são disjuntas e não há subexpressões da forma A, é trivial mostrar que o resultado de um casamento é o mesmo nas duas semânticas. Dado que p não possui subexpressões da forma A, o seu casamento não depende da gramática. □

Agora, vamos usar o lema 4.6.2 para provar o seguinte lema a respeito do casamento em G usando $\overset{\text{LL(k)}}{\rightsquigarrow}$ e do casamento em G' usando $\overset{\text{PEG}}{\rightsquigarrow}$:

Lema 4.6.3. *Dada uma gramática $LL(k)$-forte G e uma gramática G' obtida a partir de G usando a transformação $LL(k)$-PEG, temos que $G\ xy \overset{LL(k)}{\rightsquigarrow} y$ se e somente se $G'\ xy \overset{PEG}{\rightsquigarrow} y$.*

Demonstração. ($\Leftarrow$): A prova desta parte usa indução na altura da árvore de prova dada por $\overset{\text{PEG}}{\rightsquigarrow}$. O caso interessante é quando a expressão de parsing associada a um não terminal A é uma escolha $p_1 \,/\, p_2$. Há duas regras em $\overset{\text{PEG}}{\rightsquigarrow}$ relacionadas ao casamento de uma escolha: *ord*.1 e *ord*.2.

Se a regra *ord*.1 foi usada, então $G'[p_1 \,\&\phi(A)]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Pela regra *con*.1 temos que $G'[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e por *con*.1 e *not*.1 temos que $G'[\phi(A)]\ y \overset{\text{PEG}}{\rightsquigarrow} y'$. Pela hipótese de indução temos que $G[p_1]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$, e pelo lema 4.6.2 temos que $G[\phi(A)]\ y \overset{\text{LL(k)}}{\rightsquigarrow} y'$. Assim, pela regra *choice*.1, temos que $G[p_1 \,|p_2]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$. Como o casamento de $\phi(A)$ é bem sucedido, então $take_k(y) \in FOLLOW_k^G(A)$, e pela regra $var_{LL(k)}$.1 podemos concluir que $G[A]\ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$.

Se a regra *ord*.2 foi usada, então $G'[p_2 \,\&\phi(A)]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Pela regra *con*.1 temos que $G'[p_2]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$ e por *con*.1 e *not*.1 temos que $G'[\phi(A)]\ \ y \overset{\text{PEG}}{\rightsquigarrow} y'$. Pela hipótese de indução $G[p_2]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$, e pelo lema 4.6.2 sabemos que $G[\phi(A)]\ \ y \overset{\text{LL(k)}}{\rightsquigarrow} y'$. Assim, pela regra *choice*.2, temos que $G[p_1 \mid p_2]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$. Dado que o casamento de $\phi(A)$ é bem sucedido, então $take_k(y) \in FOLLOW_k^G(A)$, e pela regra $var_{LL(k)}$.1 concluímos que $G[A]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$.

($\Rightarrow$): Esta parte da prova usa indução na altura da árvore de prova dada por $\overset{\text{LL(k)}}{\rightsquigarrow}$. O caso interessante é o da expressão de parsing $p_1 \mid p_2$, cujas regras associadas são *choice*.1 e *choice*.2. Nesta prova usaremos o fato de que G possui estrutura BNF, e portanto toda escolha $G[p_1 \mid p_2]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$ possui um consequente $G[A]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$, onde por $var_{LL(k)}$.1 sabemos que $take_k(y) \in FOLLOW_k^G(A)$.

Se a regra *choice*.1 foi usada, temos que $G[p_1]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$, e pela hipótese de indução $G'[p_1]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Dado que $take_k(y) \in FOLLOW_k^G(A)$, como $\phi(A)$ casa $take_k(y)$ então $G[\phi(A)]\ \ y \overset{\text{LL(k)}}{\rightsquigarrow} y'$. Pelo lema 4.6.2 sabemos que $G'[\phi(A)]\ \ y \overset{\text{PEG}}{\rightsquigarrow} y'$ e pela regra *not*.1 temos que $G'[\&\phi(A)]\ \ y \overset{\text{PEG}}{\rightsquigarrow} y$. A regra *con*.1 nos dá que $G'[p_1 \,\&\phi(A)]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e pela regra *ord*.1 concluímos que $G'[p_1 \,/\, p_2]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$.

Se *choice*.2 foi usada, então $G[p_2]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$. Como p_2 casa x, temos que $x \in FIRST_k^G(p_2)$, e dado que $take_k(y) \in FOLLOW_k^G(A)$ sabemos que $G[\phi(A)]\ \ y \overset{\text{LL(k)}}{\rightsquigarrow} y'$. Uma vez que G é $LL(k)$-forte temos que $take_k(xy) \notin (FIRST_k^G(p_1) \otimes_k FOLLOW_k^G(A))$, e portanto o casamento de $p_1\,\phi(A)$ não é bem sucedido quando a entrada possui prefixo $take_k(xy)$.

Como G' é completa, pela contrapositiva temos que $G'[p_1\,\phi(A)]\ \ xy \overset{\text{PEG}}{\rightsquigarrow}$ `fail`, logo também temos que $G'[p_1 \,\&\phi(A)]\ \ xy \overset{\text{PEG}}{\rightsquigarrow}$ `fail`. Dado que $G[p_2]\ \ xy \overset{\text{LL(k)}}{\rightsquigarrow} y$, pela hipótese de indução temos que $G'[p_2]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e dado que $G[\phi(A)]\ \ y \overset{\text{LL(k)}}{\rightsquigarrow} y'$, pelo lema 4.6.2 e pela regra *not*.1 temos que $G'[\&\phi(A)]\ \ y \overset{\text{PEG}}{\rightsquigarrow} y$. Finalmente, pela regra *con*.1 temos que $G'[p_2 \,\&\phi(A)]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$, e pela regra *ord*.2 podemos concluir que $G'[p_1 \,/\, p_2]\ \ xy \overset{\text{PEG}}{\rightsquigarrow} y$. □

Dada uma CFG $LL(k)$-forte G e uma PEG G' obtida a partir de G através da transformação $LL(k)$-PEG, a proposição a seguir nos diz que $L^{CFG}(G) = L^{PEG}(G')$:

Proposição 4.6.4. *Dada uma gramática $LL(k)$-forte $G = (V, T, P, S)$ e uma gramática G' obtida a partir de G usando a transformação $LL(k)$-PEG, temos que $G\ w\$^k \overset{CFG}{\rightsquigarrow} \$^k$ se e somente se $G'\ w\$^k \overset{PEG}{\rightsquigarrow} \$^k$.*

Classe da CFG G	PEG Equivalente	Equivalência
Linear à Direita	G' (transformação ε-End)	$G\ w \overset{\text{CFG}}{\rightsquigarrow} \varepsilon \Leftrightarrow G'\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$
LL(1)	G (com estrutura BNF)	$G\ w\$ \overset{\text{CFG}}{\rightsquigarrow} \$ \Leftrightarrow G\ w\$ \overset{\text{PEG}}{\rightsquigarrow} \$$
$LL(k)$-forte	G' (transformação $LL(k)$-PEG)	$G\ w\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k \Leftrightarrow G'\ w\$^k \overset{\text{PEG}}{\rightsquigarrow} \$^k$

Tabela 4.1: Equivalência Entre Classes de CFGs e PEGs

Demonstração. ($\Rightarrow$): Dado que $G\ w\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ e que $\$^k \in FOLLOW_k^G(S)$, pelo lema 4.6.1 temos que $G\ w\$^k \overset{\text{LL(k)}}{\rightsquigarrow} \$^k$, e pelo lema 4.6.3 concluímos que $G'\ w\$^k \overset{\text{PEG}}{\rightsquigarrow} \$^k$.

($\Leftarrow$): Dado que $G'\ w\$^k \overset{\text{PEG}}{\rightsquigarrow} \$^k$, pelo lema 4.6.3 temos que $G\ w\$^k \overset{\text{LL(k)}}{\rightsquigarrow} \$^k$, e pelo lema 4.6.1 concluímos que $G\ w\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$. □

Na tabela 4.1, podemos ver um resumo da correspondência que apresentamos entre algumas classes de CFGs e PEGs equivalentes.

5
Conclusão

Neste capítulo, vamos discutir alguns trabalhos relacionados e elencar as contribuições deste trabalho.

Embora o surgimento de PEGs tenha gerado um grande interesse prático e muitos geradores de parsers baseados em PEGs estejam disponíveis [Grimm, 2006, Ierusalimschy, 2009, Piumarta, 2007], o interesse acadêmico por PEGs mostrou-se reduzido, de modo que a literatura sobre PEGs é escassa.

Na próxima seção discutimos trabalhos relacionados e na seção 5.2 revemos as contribuições deste trabalho.

5.1
Trabalhos Relacionados

A formalização de PEGs apresentada por Ford foi bastante influenciada pelo trabalho anterior de Birman [Birman, 1970, Birman e Ullman, 1973], que definiu TDPL e GTDPL. Várias das propriedades de TDPL e GTDPL foram depois adaptadas por Ford para PEGs. Se analisarmos as definições desses formalismos, podemos ver que as características fundamentais de PEGs são a noção de falha e o uso de uma escolha ordenada, pois são características que também estão presentes em TDPL e GTDPL. Ao contrário de PEGs, o uso prático de TDPL e GTDPL foi bastante limitado. Segundo Ford [Ford, 2004], isso se deveu em grande parte ao fato de TDPL e GTDPL terem sido apresentados como modelos formais para parsers top-down, e não como formalismos sintáticos que poderiam ser usados para a descrição de parsers top-down.

Ford [Ford, 2002, 2004] apresenta PEGs como um formalismo para descrever linguagens que, ao contrário de CFGs, não permite expressar ambiguidade. Como o próprio Ford nota, embora o uso de PEGs para descrever algumas linguagens seja mais conveniente, nem sempre PEGs são a ferramenta mais adequada, e o uso de um operador de escolha ordenada introduz o problema de determinar corretamente a ordem das alternativas. Em seu trabalho, Ford ressalta a importância de estudar o relacionamento entre CFGs e PEGs, mas não realiza esse estudo nem sugere uma abordagem específica para estabelecer

essa correspondência.

A transformação Π apresentada aqui pode ser facilmente estendida de modo a converter extensões usadas por bibliotecas de casamento de padrões para PEGs. O trabalho de Oikawa et al. [2010] apresenta mais informalmente a transformação Π, sem mostrar uma prova de corretude da transformação, e discute como adaptá-la para converter extensões usadas por bibliotecas de casamento de padrões para PEGs.

Um outro método para converter expressões regulares para PEGs é descrito por Ierusalimschy [Ierusalimschy, 2009]. No método proposto por Ierusalimschy, uma expressão regular é primeiro convertida para um autômato, e em seguida o autômato é convertido para uma PEG. Uma desvantagem dessa abordagem é que ela não permite acomodar extensões usadas por bibliotecas de casamento de padrões, uma vez que não poderíamos representar essas extensões através de autômatos.

Outra tentativa de usar conceitos clássicos de parsers top-down preditivos para definir a linguagem de PEGs foi feita por Redziejowski [Redziejowski, 2009], que definiu as funções $FIRST^G$ e $FOLLOW^G$ para PEGs. Contudo, as funções $FIRST^G$ e $FOLLOW^G$ usadas por Redziejowski permitem estabelecer apenas de forma aproximada qual é a linguagem definida por uma PEG.

Alguns geradores de parsers baseados em PEGs, como Rats! [Grimm, 2006], implementam regras específicas quando uma das alternativas de uma escolha ordenada pode casar a cadeia vazia. No caso de Rats!, não é possível gerar um parser se a gramática possui uma escolha ordenada onde a alternativa que casa a cadeia vazia não é a última.

ANTLR [Parr e Quong, 1995, Parr, 2007] é um gerador de parsers top-down para linguagens $LL(k)$-forte e foi uma das primeiras ferramentas a permitir o uso de predicados sintáticos. Quando predicados sintáticos são usados, ANTLR gera um parser que pode fazer backtracking. Uma outra maneira em ANTLR de gerar um parser que faz backtracking é através do modo de backtracking. Quando esse modo está habilitado, caso não seja possível gerar um parser $LL(k)$-forte a partir de uma dada gramática, é gerado um parser que tenta casar as alternativas de uma escolha em ordem. Nesse caso, os parsers gerados por ANTLR implementam a mesma semântica de PEGs [Parr, 2007, Ford, 2004].

A correspondência entre CFGs lineares à direita e PEGs foi apontada anteriormente por Ierusalimschy [Ierusalimschy, 2009]. Contudo, uma prova mais detalhada dessa correspondência não foi apresentada.

5.2 Contribuições

Apresentamos um estudo sobre PEGs e estabelecemos a sua correspondência com expressões regulares e CFGs lineares à direita e $LL(k)$-forte.

Revimos a formalização original de PEGs dada por Ford, apresentamos uma nova formalização de PEGs baseada em semântica natural, e mostramos a correspondência entre a nossa formalização e a formalização usada por Ford.

Também apresentamos uma formalização de expressões regulares baseada em semântica natural, e definimos a equivalência entre expressões regulares e PEGs. Com base nessa definição de equivalência, discutimos a função Π que transforma uma expressão regular em uma PEG equivalente e provamos a sua corretude. Além disso, mostramos como podemos obter expressões regulares bem formadas.

No estudo da correspondência entre CFGs e PEGs, apresentamos uma nova formalização de CFGs baseada em semântica natural. Essa nova formalização facilitou o estudo da correspondência de CFGs lineares à direita e $LL(k)$-forte com PEGs, pois nos permitiu ver claramente quais são os pontos em comum e quais são as diferenças entre as semânticas de CFGs e de PEGs. Mostramos que a diferença principal entre CFGs e PEGs está na definição das regras que tratam de uma escolha, uma vez que a definição dada para CFGs faz com que o casamento usando a semântica de $\overset{\text{CFG}}{\rightsquigarrow}$ seja não determinístico, enquanto que a definição dada para PEGs faz com que o casamento usando a semântica de $\overset{\text{PEG}}{\rightsquigarrow}$ seja determinístico.

Mostramos como transformar uma CFG linear à direita em uma PEG quase idêntica e que descreve a mesma linguagem que a CFG quando consideramos apenas os casamentos onde toda a entrada é consumida.

Discutimos a interpretação de uma gramática como uma CFG e como uma PEG, e provamos que uma gramática $LL(1)$ sem expressões ε descreve a mesma linguagem quando interpretada como uma CFG e quando interpretada como uma PEG.

Em seguida, provamos que uma gramática $LL(1)$ com uma pequena restrição, a de que somente a última alternativa de uma escolha pode casar a cadeia vazia, descreve a mesma linguagem se a interpretarmos como uma CFG ou como uma PEG.

Mostramos também como converter uma CFG $LL(k)$-forte em uma PEG equivalente que é bastante similar, onde para cada produção $A \rightarrow p_1 \mid p_2$ na CFG, há uma produção $A \rightarrow p_1 \,\&\phi(A) \,/\, p_2 \,\phi(A)$ correspondente na PEG.

Além de provar a equivalência entre algumas classes de CFGs e PEGs, outra contribuição deste trabalho é a abordagem formal que apresentamos para

estudar o relacionamento entre CFGs e PEGs. Esperamos que essa abordagem seja usada para estabelecer a correspondência entre outras classes de CFGs e PEGs equivalentes.

Referências Bibliográficas

Alfred Aho e Jeffrey Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, EUA, 1972. ISBN 0-13-914556-7. 1

Alexander Birman. *The TMG Recognition Schema*. PhD thesis, Princeton University, 1970. 1, 2.2, 5.1

Alexander Birman e Jeffrey Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, 1973. 1, 2.2, 5.1

Charles Fischer e Richard LeBlanc. *Crafting a Compiler with C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, EUA, 1991. ISBN 0-8053-2166-7. 4.5, 4.6

Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002. 1, 1.1, 5.1

Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, EUA, 2004. ACM. ISBN 1-58113-729-X. doi: http://doi.acm.org/10.1145/964001.964011. 1, 1.1, 2.1, 2.2, 2.3, 2.3, 5.1

Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006. ISBN 0596528124. 1

Jan Goyvaerts e Steven Levithan. *Regular Expressions Cookbook*. O'Reilly Media, Inc., 2009. 1

Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, EUA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1133987. 5, 5.1

Dick Grune e Ceriel Jacobs. *Parsing Techniques — A Practical Guide*. Springer-Verlag New York, Inc., Secaucus, EUA, 2006. 4.6

John Hopcroft e Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, EUA, 1979. ISBN 0321462254. 3.1, 4.1, 4.4

Roberto Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Software - Practice & Experience*, 39(3):221–258, 2009. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.v39:3. 1, 4.4, 5, 5.1

ISO. *Syntactic metalanguage — Extended BNF*. International Standards Organization, 1996. ISO/IEC 14977. 1.1, 4.2

Gilles Kahn. Natural semantics. In *STACS '87: Symposium on Theoretical Aspects of Computer Science*, pages 22–39, Londres, Reino Unido, 1987. Springer-Verlag. ISBN 3-540-17219-X. 1

Donald Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, 1964. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355588.365140. 4.2

John Levine, Tony Mason, e Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, EUA, 1992. ISBN 1-56592-000-7. 1.1

Marcelo Oikawa, Roberto Ierusalimschy, e Ana Moura. Converting regexes to parsing expression grammars. In *Simpósio Brasileiro de Linguagens de Programação (a ser publicado)*, 2010. ISBN 978-1-60558-270-2. 1, 5.1

Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007. ISBN 0978739256. 1.1, 5.1

Terence Parr e Russell Quong. ANTLR: a predicated-LL(k) parser generator. *Software - Practice & Experience*, 25(7):789–810, 1995. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380250705. 1.1, 5.1

perldoc. perldoc.perl.org — official documentation for the Perl programming language. `http://perldoc.perl.org/perlre.html`, 2010. [Online; página visitada em julho de 2010]. 3.5

Ian Piumarta. peg/leg — recursive-descent parser generators for C. `http://piumarta.com/software/peg/`, 2007. [Online; página visitada em julho de 2010]. 5

Roman Redziejowski. Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae*, 79(3-4):513–524, 2008. ISSN 0169-2968. 1

Roman Redziejowski. Applying classical concepts to parsing expression grammar. *Fundamenta Informaticae*, 93(1-3):325–336, 2009. ISSN 0169-2968. 5.1

Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996. ISBN 053494728X. 3.1

Franklyn Turbak e David Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008. ISBN 0262201755, 9780262201759. 1

Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, EUA, 2000. ISBN 0596000278. 3.5

Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, EUA, 1993. ISBN 0-262-23169-7. 1

Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359863.359883. 1.1, 4.2