Teoria da Computação	**FCT-UNL, 2º semestre 2023-2024**

Notas 9: Gramáticas livres de contexto

Autor: João Ribeiro

Introdução

Nestas notas vamos explorar Gramáticas Livres de Contexto (GLCs), que conseguem gerar mais linguagens do que as expressões regulares (ou seja, existem linguagens não regulares geradas por GLCs). As linguagens geradas por GLCs chamam-se Linguagens Livres de Contexto (LLCs).

Tal como expressões regulares são equivalentes a AFDs, GLCs são equivalentes a uma noção mais poderosa de *autómato de pilha*, que corresponde, intuitivamente, a um AFD com acesso extra a uma *stack* (no entanto, não vamos explorar esta equivalência). GLCs têm aplicações práticas no contexto de parsing em compiladores.

9.1 Gramáticas e linguagens livres de contexto

Antes de definirmos rigorosamente o que são GLCs e LLCs, discutimos um pouco a motivação que antecede estes conceitos. Anteriormente, vimos como expressões regulares nos permitem gerar todas as linguagens regulares. No entanto, também vimos que expressões regulares sofrem de algumas limitações. Por exemplo, vimos que a linguagem

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

não é regular, e portanto não é gerada por nenhuma expressão regular. Se substituirmos "0" por "(" e "1" por ")" na linguagem L acima, concluímos que AFDs e expressões regulares não são poderosos o suficiente para verificarem se uma da expressão, ou linha de código de um programa, tem parênteses bem formados, pois, nesse caso, tem de existir o mesmo número de "(" e ")".

As GLCs são mecanismos de geração de linguagens mais poderosos do que expressões regulares, que tiveram origem na linguística, no estudo da língua natural (para os interessados em história e linguística, recomendo o trabalho de Chomsky e Schützenberger, e.g., [Cho56, Cho59, CS63] e esta página da Wikipedia). Estas gramáticas transcenderam a sua aplicação original, e mostraram também ser ferramentas de grande utilidade na especificação e compilação de linguagens de programação, como veremos mais à frente nestas notas.

Uma GLC G consiste em:

- Uma variável inicial S;
- Um conjunto V de variáveis, com $S \in V$;

- Um conjunto Σ de símbolos terminais;
- Um conjunto R de regras de substituição (ou de produção) da forma $A \to \beta$, onde A é uma variável e β é uma string de variáveis e símbolos terminais. Se α é uma string de variáveis e símbolos terminais, podemos aplicar a regra $A \to \beta$ a α substituindo uma ocorrência da variável A em α da nossa escolha pela string β.

Por exemplo, consideremos a GLC $G = (V, \Sigma, R, S)$ com conjunto de variáveis $V = \{S\}$, conjunto de símbolos terminais $\Sigma = \{0, 1\}$, e regras de substituição

$$(R_1) \quad S \to 0S1\,,$$
$$(R_2) \quad S \to \varepsilon\,.$$

Se temos várias regras $A \to \beta_1, A \to \beta_2, \ldots, A \to \beta_k$ para a mesma variável A, pode ser conveniente abreviá-las como

$$A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_k\,.$$

Por exemplo, para G acima poderíamos apenas escrever $S \to 0S1 \| \varepsilon$. No entanto, aconselhamos sempre darem nomes às regras que definem.

Uma string $w \in \Sigma^*$ diz-se gerada por G, o que denotamos por $S \overset{*}{\underset{G}{\Longrightarrow}} w$, se existir uma sequência finita de aplicações de regras de substituição à variável inicial S que resulta em w. Por exemplo, a string 0011 é gerada por G através da derivação

$$S \underset{R_1}{\Longrightarrow} 0S1 \underset{R_1}{\Longrightarrow} 00S11 \underset{R_2}{\Longrightarrow} 00\varepsilon 11 = 0011,$$

e podemos então escrever $S \overset{*}{\underset{G}{\Longrightarrow}} 0011$. Mais geralmente, se α e β são sequências de variáveis e símbolos terminais (ou seja, α e β são sequências de símbolos de $V \cup \Sigma \cup \{\varepsilon\}$), então escrevemos $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ quando existe uma sequência finita de regras de substituição que transformam α em β.

A linguagem associada a uma GLC G, denotada por $L(G)$, corresponde exactamente a todas as strings geradas por G, ou seja,

$$L(G) = \left\{ w \in \Sigma^* \,\middle|\, S \overset{*}{\underset{G}{\Longrightarrow}} w \right\}.$$

Neste caso, dizemos que $L(G)$ é uma *Linguagem Livre de Contexto* (LLC). Curiosamente, para a GLC G definida acima temos

$$L(G) = \{0^n 1^n \mid n \in \mathbb{N}\},$$

o que mostra que algumas GLCs geram linguagens não regulares! Veremos adiante que LLCs são uma classe mais geral do que as linguagens regulares.

9.2 Exemplos

Consideramos mais alguns exemplos de GLCs.

Gramáticas Livres de Contexto

$G = (V, \Sigma, R, S)$

$V \to$ Conj variáveis (símbolos não terminais)

$\Sigma \to$ Alfabeto (símbolos terminais)

$S \to$ variável inicial

$R \to$ conj de regras de substituição

$X \to \alpha$

$X \in V$ e α é sequência de símbolos $V \cup \Sigma$

exemplo

$L = \{0^n \mid n$ é par$\}$

$\Sigma = \{0\}$ $\quad V = \{S\}$ $\quad S = S \leftarrow$ início

(R1) $S \to S00$ ← concatenação de $S \in G$ com 00

(R2) $S \to \varepsilon$

Será que $w = 000000 \in L$

$S \xrightarrow[R1]{} S00 \xrightarrow[R1]{} S0000 \xrightarrow[R1]{} S000000 \xrightarrow[R2]{} \varepsilon 000000 = 000000 = w$

— // —

$S \overset{*}{\underset{G}{\Rightarrow}} w \to w$ é derivável a partir de S sobre G ← gramática x

$\alpha_1, \alpha_2, \ldots, \alpha_k \in (V \cup \Sigma)^*$ e Regras $R_1, R_2, \ldots, R_{k+1} \in R$ tal que

$S \underset{R_1}{\Rightarrow} \alpha_1 \underset{R_2}{\Rightarrow} \alpha_2 \underset{R_3}{\Rightarrow} \alpha_3 \Rightarrow \ldots \underset{R_k}{\Rightarrow} \alpha_k \underset{R_{k+1}}{\Rightarrow} \varepsilon w = w$

$\underbrace{L(G)}_{\text{Linguagem gerada por } G} = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$

— // —

$L = \{0^n 1^n \mid n \in \mathbb{N}\}$

$G = (V, \Sigma, R, S)$

$V = \{S\}$ $\quad S = S \leftarrow$ início

$\Sigma = \{0, 1\}$

(R1) $S \to 0S1$

(R2) $S \to \varepsilon$ } Se a linguagem fosse $L' = \{0^n 1^n \mid n \in \mathbb{N}^+\}$ seria tudo igual exceto R2 que ficaria (R2) $S \to 01$

$L = L(G)$

— // —

Def L é linguagem livre de contexto se existe GLC (gramática livre de contexto) G tal que $L = L(G)$

— // —

$L = \{a^n b^k c^k d^n \mid k, n \in \mathbb{N}\}$

$G = (V, \Sigma, R, S)$ $\quad S = S \leftarrow$ início

$V = \{V, T\}$

$\Sigma = \{a, b, c, d\}$

(R1) $S \to aSd$

(R2) $T \to bTc$

(R3) $S \to T$

(R4) $T \to \varepsilon$

$w = abbccd$

$S \xrightarrow[R1]{} aSd \xrightarrow[R3]{} aTd \xrightarrow[R2]{}$
$\to abTcd \xrightarrow[R2]{} abbTccd \xrightarrow[R4]{}$
$\to abb\varepsilon ccd = abbccd = w$

— // —

$L = \{a^n b^n c^k d^k \mid k, n \in \mathbb{N}\}$

$G = (V, \Sigma, R, S)$ $\quad S = S$ ↑ início

$V = \{S, T, U\}$

$\Sigma = \{a, b, c, d\}$

(R1) $S \to TU$

Exemplo 9.1 Consideremos a GLC $G = (V, \Sigma, R, S)$ com $V = \{S\}$, $\Sigma = \{(,)\}$, e regras

$$(R_1) \quad S \to (S)$$
$$(R_2) \quad S \to SS$$
$$(R_3) \quad S \to \varepsilon\,.$$

Para ganharmos intuição, experimentamos gerar algumas strings:

$$S \underset{R_1}{\Longrightarrow} (S) \underset{R_1}{\Longrightarrow} ((S)) \underset{R_1}{\Longrightarrow} (((S))) \underset{R_3}{\Longrightarrow} ((()))\,.$$
$$S \underset{R_2}{\Longrightarrow} SS \underset{R_1}{\Longrightarrow} (S)S \underset{R_1}{\Longrightarrow} (S)(S) \underset{R_3}{\Longrightarrow} ()(S) \underset{R_1}{\Longrightarrow} ()((S)) \underset{R_3}{\Longrightarrow} ()(())\,.$$

Devem convencer-se de que esta GLC gera exactamente as sequências de parênteses bem formados (i.e., *nested*).

Exemplo 9.2 Vemos agora um exemplo com mais do que uma variável. Vamos mostrar que a linguagem

$$L = \{a^n b^m c^m d^n \mid m, n \in \mathbb{N}\}$$

é livre de contexto. Temos, então, de descrever uma GLC G tal que $L = L(G)$. Consideramos $G = (V, \Sigma, R, S)$ com $V = \{S, T\}$, $\Sigma = \{a, b, c, d\}$, e regras

$$(R_1) \quad S \to aSd$$
$$(R_2) \quad S \to T$$
$$(R_3) \quad T \to bTc$$
$$(R_4) \quad T \to \varepsilon\,.$$

Por exemplo, temos $abbccd \in L(G)$, pois

$$S \underset{R_1}{\Longrightarrow} aSd \underset{R_2}{\Longrightarrow} aTd \underset{R_3}{\Longrightarrow} abTcd \underset{R_3}{\Longrightarrow} abbTccd \underset{R_4}{\Longrightarrow} abbccd\,.$$

9.3 Todas as linguagens regulares são livres de contexto

Vimos na Secção 9.1 que certas linguagens não regulares são geradas por GLCs. Isto mostra que LLCs são uma classe diferente das linguagens regulares. Mais ainda, intuitivamente, parece que GLCs são mais poderosas que AFDs e expressões regulares. Demonstramos isto de forma rigorosa.

Teorema 9.1 *Se $L \subseteq \Sigma^*$ é regular, então L também é LLC.*

Demonstração: Se L é uma linguagem regular, então existe um AFD completo $M = (S, \Sigma, \delta, s, F)$ tal que $L = L(M)$. Vamos criar uma GLC G a partir de M tal que $L(G) = L(M)$. Isto mostra que $L = L(M)$ é LLC.

Consideremos a GLC $G = (V, \Sigma, R, s)$ cujas variáveis são estados de M (ou seja, $V = S$), com variável inicial s (o estado inicial de M), e com conjunto de símbolos terminais Σ (o alfabeto de M). As regras de substituição de G são obtidas da seguinte forma: Para quaisquer estados $q, q' \in S$ e símbolo $x \in \Sigma$ tal que $\delta(q, x) = q'$, adicionamos a regra

$$q \to xq'$$

ao conjunto de regras R. Se $q \in F$, adicionamos também a regra

$$q \to \varepsilon.$$

Deixamos como exercício para o leitor mostrar que $L(G) = L(M)$. ■

A Figura 9.1 resume a relação entre a classe das linguagens regulares, a classe das LLCs, e a classe de todas as linguagens Σ^*.

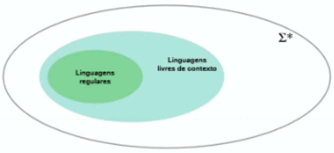

Figure 9.1: Comparação entre linguagens regulares e linguagens livres de contexto.

9.4 Parsing

Parsing, ou análise sintática, consiste na análisa da "estrutura/interpretação" de uma dada palavra w. Mais precisamente, dada uma gramática G e uma palavra w, um *parser* deve (1) verificar se $w \in L(G)$ (ou seja, se w é uma palavra válida), e, em caso afirmativo, (2) "encontrar uma derivação" de w através da gramática G (seremos mais precisos sobre isto adiante). Parsers são componentes importantes no desenho de compiladores, e a sua "eficiência" (a rapidez com que geram a derivação de w) é fundamental. Fragmentos de certas linguagens de programação foram desenhados de forma a serem capturados por GLCs com propriedades especiais que permitem a geração muito eficiente (em tempo linear no tamanho de w) e sistemática de derivações. Uma excelente fonte sobre a aplicação de autómatos e gramáticas no desenho de compiladores é o ***Dragon Book*** de Aho, Lam, Sethi, e Ullman [ALSU06].

(R2) $T \to aTb$

(R3) $U \to cUd$

(R4) $T \to \varepsilon$

(R5) $U \to \varepsilon$

$L \subseteq \Sigma^*$ linguagem regular

Será que L é livre de contexto? Sim!

Dem

seja $M = (S, \Sigma, \delta, s, F)$ AFD que reconhece L
Consideremos a GLC $G = (V, \Sigma, R, Init)$ onde $V = S$,
$Init = s$ e Regras

(R1) $q \to a\,\delta(q, a)$, onde $a \in \Sigma$ e $q \in S$

(R2) $q \to \varepsilon$, onde $q \in F$

* se existe um autómato que reconhece a linguagem então existe uma gramática que reconhece essa linguagem *

9.4.1 Árvores de parsing e derivações leftmost

Na realidade, o objectivo principal de um parser é gerar o que chamamos de ***árvore de parsing*** de uma palavra w. Uma árvore de parsing corresponde a uma classe de equivalência de possíveis derivações, existindo uma correspondência única com um tipo especial de derivações que apelidamos de ***derivações leftmost***.

Começamos com um exemplo. Consideremos a GLC descrita no Exemplo 9.1. A Figura 9.2 ilustra a árvore de parsing segundo G para a palavra $w = ()() \in L(G)$ associada à derivação

$$S \implies SS \implies (S)S \implies ()S \implies ()(S) \implies ()().$$

Na raíz da árvore colocamos a variável inicial, S. Começamos por aplicar a regra $S \to SS$. Isto gera dois filhos (um por símbolo), ambos etiquetados por S. Por sua vez, aplicamos a regra $S \to (S)$ ao S à esquerda, e depois ao S à direita. Cada uma destas aplicações gera três filhos (um por cada símbolo), etiquetados por "(", S, e ")", respectivamente. Finalmente, os nós etiquetados por S têm filhos etiquetados por ε, resultantes de aplicações da regra $S \to \varepsilon$. O *resultado* da árvore corresponde à concatenação das etiquetas das suas folhas da esquerda para direita: $(\varepsilon)(\varepsilon) = ()()$.

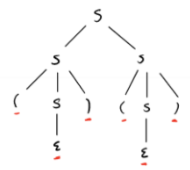

Figure 9.2: Árvore de parsing de $w = ()()$.

Como mencionámos acima, uma árvore de parsing captura várias possíveis derivações de w. Por exemplo, outra derivação possível para $w = ()()$ é

$$S \implies SS \implies (S)S \implies (S)(S) \implies ()(S) \implies ()(), \tag{9.1}$$

que também gera a árvore de parsing da Figura 9.2. Chamamos derivações que levam à mesma árvore de parsing ***equivalentes***. Intuitivamente, derivações equivalentes diferem apenas em aspectos que não afectam o "significado" de w.

Formalmente, uma árvore de parsing é definida da seguinte maneira.

Definição 9.1 (Árvore de parsing) *Uma árvore de parsing para uma GLC $G = (V, \Sigma, R, S)$ tem o seguinte formato:*

- *A raíz da árvore é etiquetada por S, a variável inicial de G.*
- *As folhas da árvore são etiquetados por símbolos terminais de G (i.e., um símbolo de $\Sigma \cup \{\varepsilon\}$).*
- *Os vértices que não são folhas são etiquetados por símbolos não terminais de G (i.e., uma variável de V).*
- *Se um vértice é etiquetado por $X \in V$ e tem filhos etiquetados por $X_1, X_2, \ldots, X_n$, com $X_i \in V \cup \Sigma \cup \{\varepsilon\}$, então a regra*
$$X \to X_1 X_2 \ldots X_n$$
tem de pertencer a R.

A concatenação das etiquetas das folhas da árvore de parsing, da esquerda para a direita, representam o resultado *w da árvore, que é a palavra de $L(G)$ gerada (ou derivada) através dessa árvore.*

Outro conceito importante no contexto do desenho de parsers é o de derivação *leftmost*. Informalmente, numa derivação leftmost aplicamos sempre, por defeito, regras de substituição à variável mais à esquerda da nossa palavra.

Definição 9.2 (Derivação leftmost) *Uma derivação leftmost de w segundo uma GLC $G = (V, \Sigma, R, S)$ é qualquer derivação*

$$S = \alpha_0 \implies \alpha_1 \implies \alpha_2 \implies \cdots \implies \alpha_k = w$$

em que para cada $i \in \mathbb{N}^+$ temos que α_i é obtido a partir de α_{i-1} através da aplicação de uma regra de substituição à variável mais à esquerda de α_{i-1}.

Por exemplo, a derivação descrita na Equação (9.1) é uma derivação leftmost de $()()$. Também há casos em que existem várias derivações leftmost da mesma palavra. Por exemplo, para $w = ()()()$ (segundo a mesma GLC) temos as derivações leftmost

$$S \implies SS \implies SSS \implies (S)SS \implies ()SS \implies ()(S)S \implies ()()S \implies ()()(S) \implies ()()()$$

e

$$S \implies SS \implies (S)S \implies ()S \implies ()SS \implies ()(S)S \implies ()()S \implies ()()(S) \implies ()()().$$

As árvores de parsing associadas a estas duas derivações encontram-se nas Figuras 9.3 and 9.4, respectivamente. Observamos que as duas derivações leftmost distintas originam árvores de parsing diferentes. Na realidade, isto é um fenómeno mais geral, e existe uma correspondência unívoca entre árvores de parsing e derivações leftmost. Podemos interpretar as derivações leftmost como representantes "canónicos" de uma árvore de parsing (que, como já vimos, captura um conjunto de derivações equivalentes). Mais formalmente, temos o seguinte para qualquer GLC G e $w \in L(G)$:

- w tem uma derivação leftmost;
- Derivações leftmost distintas de w geram árvores de parsing diferentes;
- Cada árvore de parsing de w é gerada por uma derivação leftmost de w.

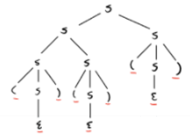

Figure 9.3: Árvore de parsing de $w = ()()()$.

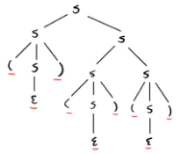

Figure 9.4: Outra árvore de parsing de $w = ()()()$.

9.4.2 Ambiguidade de GLCs

Como mencionado acima, o objectivo principal de um parser consiste em gerar (por vezes implicitamente) uma árvore de parsing de uma dada palavra $w \in L(G)$, que representa a estrutura de w segundo a GLC G. E, como também vimos acima nas Figuras 9.3 and 9.4, existem casos onde w pode ter mais do que uma árvore de parsing. Isto acontece exactamente quando w tem mais do que uma derivação leftmost. No contexto da interpretação e tradução mecânica de linhas de código durante a compilação, esta ambiguidade pode ser problemática. Se existem duas interpretações possíveis de uma linha de código, qual devemos usar?

Um exemplo concreto onde a ambiguidade é problemática é o seguinte: Consideramos a GLC de expressões aritméticas simples $G = (V, \Sigma, R, S)$ com $V = \{\mathsf{expr}\}$, $\Sigma = \{+, -, 0, 1, \ldots, 9\}$, $S = \mathsf{expr}$, e regras

$$
\begin{array}{ll}
(R_1) & \mathsf{expr} \to \mathsf{expr} + \mathsf{expr} \\
(R_2) & \mathsf{expr} \to \mathsf{expr} - \mathsf{expr} \\
(R_3) & \mathsf{expr} \to 0 \mid 1 \mid \cdots \mid 9 \,.
\end{array}
$$

Consideremos a expressão aritmética $w = 9 - 5 + 2 \in L(G)$. Em que ordem realizamos estas

operações? Se primeiro calcularmos a soma e depois a diferença, obtemos 2. Se primeiro calcularmos a diferença e depois a soma, obtemos 6. Uma maneira de resolver esta ambiguidade consiste em, por exemplo, seguir a convenção de que aplicamos sempre primeiro a operação mais à esquerda, caso em que $w = 6$ é a interpretação correcta. Gostaríamos que a nossa GLC evitasse estas ambiguidades, estando estas convenções codificadas no próprio desenho da GLC. No entanto, a GLC acima definida é ambígua relativamente à ordem de aplicação das operações, pois $w = 9 - 5 + 2$ tem as derivações leftmost

$$\mathsf{expr} \implies \mathsf{expr} + \mathsf{expr} \implies \mathsf{expr} - \mathsf{expr} + \mathsf{expr} \implies 9 - \mathsf{expr} + \mathsf{expr} \implies 9 - 5 + \mathsf{expr} \implies 9 - 5 + 2$$

e

$$\mathsf{expr} \implies \mathsf{expr} - \mathsf{expr} \implies 9 - \mathsf{expr} \implies 9 - \mathsf{expr} + \mathsf{expr} \implies 9 - 5 + \mathsf{expr} \implies 9 - 5 + 2$$

com árvores de parsing associadas ilustradas nas Figuras 9.5 and 9.6.

Figure 9.5: Árvore de parsing de $w = 9 + 5 - 2$.

Figure 9.6: Outra árvore de parsing de $w = 9 + 5 - 2$.

Definição 9.3 (GLC ambígua) *Uma GLC G é* ambígua *se existir $w \in L(G)$ com mais do que uma derivação leftmost a partir de G. Equivalentemente, G é ambígua se existir $w \in L(G)$ com mais do que uma árvore de parsing segundo G.*

Uma GLC não ambígua pode ter várias derivações possíveis para a mesma palavra $w \in L(G)$. No entanto, é garantido que existe uma única derivação *leftmost* de w em G, e w tem associada apenas uma única árvore de parsing segundo G.

Segundo a definição acima, a GLC de expressões aritméticas já discutida é ambígua. No entanto, existe uma GLC ***não ambígua*** $G' = (V, \Sigma, S, R')$ que gera a mesma linguagem. Basta alterarmos o conjunto de regras R de G para R' que consiste nas regras

$$\begin{array}{ll} (R_1) & \mathsf{expr} \to \mathsf{expr} + \mathsf{digit} \\ (R_2) & \mathsf{expr} \to \mathsf{expr} - \mathsf{digit} \\ (R_3) & \mathsf{expr} \to \mathsf{digit} \\ (R_4) & \mathsf{digit} \to 0 \mid 1 \mid \cdots \mid 9\,. \end{array}$$

Esta GLC G' captura a regra mencionada acima em que aplicamos sempre primeiro a operação mais à esquerda. Neste caso, $w = 9 - 5 + 2 \in L(G') = L(G)$, e tem uma única árvore de parsing, ilustrada na Figura 9.7.

Figure 9.7: Única árvore de parsing de $w = 9 + 5 - 2$ segundo G'.

A título de curiosidade, mencionamos que existem LLCs L que são *inerentemente ambíguas*, no sentido em que qualquer GLC G que gera L é ambígua. A existência de tais linguagens foi estabelecida por Parikh [Par66].

9.4.3 Recursividade à esquerda

Outra propriedade importante de GLCs é *recursividade à esquerda*. Intuitivamente, uma GLC tem recursividade à esquerda se é possível entrar num loop infinito através de aplicações leftmost de regras de substituição a alguma variável. No contexto de parsers "top-down", que começam na variável inicial da GLC e tentam produzir uma derivação leftmost (ou, equivalentemente, uma árvore de parsing) da palavra em causa, é importante evitar recursividade à esquerda. Mais formalmente, temos a seguinte definição.

Definição 9.4 (Recursividade à esquerda) *Uma GLC G tem* recursividade à esquerda *se existe*

uma variável $X \in V$ e uma sequência de variáveis e símbolos terminais $\beta \in V \cup \Sigma \cup \{\varepsilon\}$ tais que é possível derivar $X\beta$ a partir de X, i.e., $X \overset{*}{\underset{G}{\Longrightarrow}} X\beta$.

Por exemplo, a GLC G' da Secção 9.4.2 tem recursividade à esquerda, pois podemos derivar

$$\mathsf{expr} \to \mathsf{expr} + \mathsf{digit}.$$

9.5 Para explorar

Aconselhamos a leitura de [Sip13, Chapter 2] e [LP97, Chapter 3]. Se tiverem curiosidade sobre parsing e aplicações de autómatos e gramáticas na informática, aconselhamos a exploração do *Dragon Book* [ALSU06].

Deixamos aqui também outro problema interessante sobre GLCs com ligações a compressão de dados e complexidade [CLL+05].

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson, 2006. Ver também `https://suif.stanford.edu/dragonbook/`.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[CLL+05] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CS63] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. Elsevier, 1963.

[LP97] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall PTR, USA, 2nd edition, 1997.

[Par66] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, oct 1966.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation.* CEngage Learning, 3rd edition, 2013.