Capítulo 11: Linguagens e Gramáticas

Máquinas de Turing trabalham com cadeias de caracteres como entradas e saídas, através de sequências de símbolos na fita. Assim, uma forma natural de trabalhar é representar um problema como uma cadeia, colocá-lo na fita, executar uma computação, terminando com a solução como uma cadeia.

Os computadores atuais funcionam da mesma maneira. Considere um programa que encontre a distância de viagem mais curta entre cidades. Provavelmente trabalhamos introduzindo as distâncias do mapa como cadeias de símbolos e introduzindo as duas cidades desejadas como duas cadeias, e depois de executar o programa, temos as direções de saída como uma cadeia de caracteres. Assim, as cadeias de caracteres, e coleções de cadeias, são essenciais.

11.1. Linguagens

As nossas máquinas recebem como entrada e fornecem como saídas cadeias de símbolos. Consideramos um símbolo (por vezes chamado de **token**) como sendo uma unidade atômica que uma máquina pode ler e escrever.[1] Nos computadores binários quotidianos os símbolos são os bits, 0 e 1. Um alfabeto é um conjunto não vazio e finito de símbolos. Denota-se normalmente um alfabeto pela letra grega maiúscula Σ, embora uma exceção seja o alfabeto dos bits, $\mathbb{B} = \{0, 1\}$. Uma **cadeia** (ou **string**) sobre um alfabeto é uma sequência de símbolos desse alfabeto. Utilizamos letras gregas minúsculas como σ e τ para denotar cadeias. Utilizamos ε para denotar a cadeia vazia, a sequência de símbolos de comprimento zero. O conjunto de todas as cadeias sobre Σ é Σ^*.

Definição: Linguagem

Uma linguagem L sobre um alfabeto Σ é um conjunto de cadeias obtidas a partir desse alfabeto. Ou seja, $L \subseteq \Sigma^*$.

Exemplo 11.1.1

O conjunto de cadeias de bits que começam com 1 é $L = \{1, 10, 11, 100, \ldots\}$.

Exemplo 11.1.2

Outra linguagem sobre $\mathbb{B}$ é o conjunto finito $\{1000001, 1100001\}$.

1 Podemos imaginar o funcionário de Turing calculando sem ler e escrever símbolos, por exemplo, mantendo um registro da informação através do deslocamento de elefantes para o lado esquerdo de uma estrada ou para o lado direito. Mas podemos traduzir qualquer procedimento deste tipo num só, utilizando marcas que a cabeça de leitura/escrita do nosso mecanismo pode tratar. Assim, a leitura e a escrita não são essenciais, mas exigimo-las na definição de símbolos como uma conveniência; afinal de contas, elefantes são inconvenientes.

Exemplo 11.1.3

Seja $\Sigma = \{a, b\}$. A linguagem que consiste em cadeias onde o número de a's é o dobro do número de b's é $L = \{\varepsilon, aab, aba, baa, aaaabb, \ldots\}$.

Exemplo 11.1.4

Seja $\Sigma = \{a, b, c\}$. A linguagem das cadeias de comprimento dois sobre esse alfabeto é $L = \Sigma^2 = \{aa, ab, ba \ldots, cc\}$. Sobre o mesmo alfabeto, esta é a linguagem das cadeias de comprimento três com caracteres em ordem não descendente:

$$\{aaa, bbb, ccc, aab, aac, abb, abc, acc, bbc, bcc\}$$

(Não é que o conjunto seja ordenado, uma vez que os conjuntos não têm uma ordem. Em vez disso, cada cadeia tem os seus caracteres em ordem não descendente).

Definição: Palíndromo

Um palíndromo é uma cadeia que se lê da mesma forma para a frente e para trás. Algumas palavras da língua portuguesa que são palíndromos são 'arara', 'anilina', e 'osso'.

Exemplo 11.1.5

A linguagem dos palíndromos sobre $\Sigma = \{a, b\}$ é $L = \{\sigma \in \Sigma^* \mid \sigma = \sigma^R\}$, onde σ^R é o reverso de σ. Alguns membros são abba, aaabaaa, e a.

Exemplo 11.1.6

Seja $\Sigma = \{a, b, c\}$. As triplas pitagóricas $\langle i, j, k\rangle \in \mathbb{N}^3$ são aquelas em que $i^2 + j^2 = k^2$. Algumas dessas triplas são $\langle 3, 4, 5\rangle$, $\langle 5, 12, 13\rangle$, e $\langle 8, 15, 17\rangle$. Uma forma de descrever as triplas pitagóricas é com esta linguagem.

$$\begin{aligned} L &= \{a^i b^j c^k \in \Sigma* \mid i, j, k \in \mathbb{N} \text{ e } i^2 + j^2 = k^2\} \\ &= \{aaabbbbccccc = a^3b^4c^5, a^5b^{12}c^{13}, a^8b^{15}c^{17}, \ldots\} \end{aligned}$$

Exemplo 11.1.7

O conjunto vazio é uma linguagem $L = \{\ \}$ sobre qualquer alfabeto. Assim é também o conjunto cujo único elemento é a cadeia vazia $\hat{\mathcal{L}} = \{\varepsilon\}$. Estas duas linguagens são diferentes, porque a primeira não tem membros.

Podemos pensar que uma linguagem natural como o português consiste em sentenças, que são cadeias de palavras de um dicionário. Aqui Σ é o conjunto de palavras de um dicionário e σ é uma sentença. Isto explica a definição de "língua" como um conjunto de cadeias de palavras. Claro que a nossa definição permite que uma língua

seja qualquer conjunto de cadeias de palavras, enquanto que em português não se pode formar uma sentença apenas tomando qualquer sequência louca de palavras; uma sentença deve ser construída de acordo com regras. Estudaremos conjuntos de regras, as gramáticas, mais à frente, neste capítulo.

Definição

Uma coleção de linguagens é uma **classe**.

Exemplo 11.1.8

Fixe um alfabeto Σ. A coleção de todas as linguagens finitas sobre esse alfabeto é uma classe.

Exemplo 11.1.9

Seja P_e uma máquina de Turing, usando o alfabeto de entrada $\Sigma = \{B, 1\}$. O conjunto de cadeias $L_e = \{\sigma \in \Sigma* \mid P_e \text{ pára na entrada } \sigma \}$ é uma linguagem. A coleção de todas estas linguagens, das L_e para todo $e \in \mathbb{N}$, é a classe de linguagens computacionalmente enumeráveis sobre Σ.

Em seguida, consideraremos as operações sobre linguagens. Linguagens são conjuntos de modo que se aplicam as operações de união, interseção, etc. No entanto, por exemplo, a união de uma linguagem sobre $\{a\}^*$ com uma linguagem sobre $\{b\}^*$ é um casamento estranho, uma combinação de cadeias de a's com cadeias de b's. Ou seja, a união de uma linguagem sobre Σ_0 com uma linguagem sobre Σ_1 é uma linguagem sobre $\Sigma_0 \cup \Sigma_1$. O mesmo acontece com a intersecção.

Outras operações sobre linguagens são extensões de operações sobre cadeias de caracteres. Definimos a concatenação de linguagens como sendo a linguagem das concatenações, etc.

Definição (Operações sobre linguagens)

A **concatenação de linguagens, $L_0 \hat{} L_1$ ou $L_0 L_1$**, é o conjunto de concatenações, $\{\sigma_0 \hat{} \sigma_1 \mid \sigma_0 \in L_0 \text{ e } \sigma_1 \in L_1\}$.

Para qualquer linguagem, **a potenciação, L^k**, é a linguagem que consiste na concatenação de k cadeias, $L^k = \{\sigma_0 \hat{} \ldots \hat{} \sigma_{k-1} \mid \sigma_i \in L\}$ com $k > 0$. Em particular, $L^1 = L$. Para $k = 0$ tomamos $L^0 = \{\varepsilon\}$.[2] **O fecho de Kleene de uma linguagem, L^***, é a linguagem que consiste na concatenação de qualquer número de cadeias.

$$L^* = \{ \sigma_0 \hat{} \ldots \hat{} \sigma_{k-1} \mid k \in \mathbb{N} \text{ e } \sigma_0, \ldots, \sigma_{k-1} \in L \}$$

Isto inclui a concatenação de 0 cadeias, portanto $\varepsilon \in L^*$ se $L \neq \emptyset$.[3]

O reverso, L^R, de uma linguagem L é a linguagem das reversões, $L^R = \{\sigma^R \mid \sigma \in L\}$.

2 Por conveniência, consideramos $L^0 = \{\varepsilon\}$ até mesmo quando $L = \emptyset$.
3 Analogamente ao que fizemos para a potenciação, considera-se $\emptyset* = \{\varepsilon\}$.

Exemplo 11.1.10

Seja a linguagem o conjunto de strings de bits $L = \{1000001, 1100001\}$, então o reverso é $L^R = \{1000001, 10000011\}$.

Exemplo

Se a linguagem L consiste de duas cadeias $\{a, bc\}$ então a segunda potência dessa linguagem é $L^2 = \{aa, abc, bca, bcbc\}$. O seu fecho de Kleene é este:

$$L^* = \{\varepsilon, a, bc, aa, abc, bca, bcbc, aaa, \ldots\}$$

Observação. Aqui estão dois pontos sobre o fecho de Kleene. Anteriormente, para um alfabeto Σ definimos Σ^* como sendo o conjunto de strings sobre esse alfabeto, de qualquer comprimento. As duas definições concordam se tomarmos cada caractere do alfabeto como uma string de comprimento um. Além disso, poderíamos definir a operação de escolher repetidamente strings a partir da linguagem de duas maneiras. Poderíamos escolher uma cadeia σ a partir da linguagem e depois replicá-la, obtendo o conjunto dos σ^k's, Ou, poderíamos repetidamente escolher cadeias a partir da linguagem, obtendo $\sigma_0\hat{}\sigma_1\hat{}\ldots\hat{}\sigma_{k-1}$'s. O segundo caso é mais útil e essa é de fato a definição de L^*, como vimos anteriormente.

Encerramos essa seção com um comentário que se refere à forma como utilizaremos as linguagens posteriormente. Diremos que uma máquina **decide uma linguagem** se essa máquina computa se a sua entrada é ou não um membro da linguagem. Contudo, para alguns conjuntos há uma máquina que determina num tempo finito, para todas as entradas, se a entrada é um membro desse conjunto, mas nenhuma máquina pode determinar num tempo finito para todas as entradas que a entrada não está no conjunto. Diremos que **uma máquina reconhece** (ou aceita, ou semidecide) uma linguagem se, dado uma entrada, se ela está na linguagem, então essa máquina é capaz de computar esse fato, e se a entrada não está na linguagem, então a máquina nunca computará incorretamente que está. (A máquina pode determinar explicitamente que não está, ou simplesmente não comunicar uma conclusão, talvez por não ter parado). Em suma, decidir uma linguagem significa que para qualquer entrada a máquina computa corretamente todas as respostas “sim” e “não”, enquanto que reconhecer uma linguagem requer apenas que todas as respostas “sim” sejam obtidas e estejam corretas.

11.2. Gramáticas

Definimos que uma linguagem é um conjunto de cadeias. Mas isto permite qualquer conjunto sem pé nem cabeça. Na prática, uma linguagem é normalmente dada por regras.

Eis aqui um exemplo. Os falantes nativos de inglês dirão que o substantivo “the big red barn” (o grande celeiro vermelho) soa bem, mas que “the red big barn” soa mal. Ou seja, as sentenças em linguagem natural são construídas segundo padrões e a segunda dessas sentenças não segue o padrão para o inglês. As linguagens artificiais

como as linguagens de programação também têm regras de sintaxe, geralmente regras muito estritas.

Uma gramática é um conjunto de regras para a formação de cadeias numa linguagem, ou seja, é uma análise da estrutura de uma linguagem. Num aforismo, as gramáticas são a linguagem das linguagens. Antes da definição formal, vamos ver primeiro um exemplo.

Exemplo 11.2.1

Este é um subconjunto das regras para o inglês: (1) uma sentença (*sentence*) pode ser feita a partir de uma frase substantiva (*noun phrase*) seguida de uma frase verbal (*verb phrase*), (2) uma frase substantiva pode ser feita a partir de um artigo (*article*) seguido por um substantivo (*noun*), (3) uma frase substantiva também pode ser feita a partir de um artigo seguido por um adjetivo (*adjective*) e depois por um substantivo, (4) uma frase verbal pode ser feita com um verbo (*verb*) seguido por uma frase substantiva, (5) um artigo é '*the*', (6) um adjetivo é '*young*', (7) um verbo é '*caught*', (8) dois substantivos são '*man*' e '*ball*'.

Esta é uma notação conveniente para as regras que acabam de ser listadas:

⟨*sentence*⟩ → ⟨*noun phrase*⟩ ⟨*verb phrase*⟩
⟨*noun phrase*⟩ → ⟨*article*⟩ ⟨*noun*⟩
⟨*noun phrase*⟩ → ⟨*article*⟩ ⟨*adjective*⟩ ⟨*noun*⟩
⟨*verb phrase*⟩ → ⟨*verb*⟩ ⟨*noun phrase*⟩
⟨*article*⟩ → `the`
⟨*adjective*⟩ → `young`
⟨*verb*⟩ → `caught`
⟨*noun*⟩ → `man` | `ball`

Cada linha é uma **regra de produção ou de reescrita**. Cada uma tem uma seta, →.[4] À esquerda de cada seta está uma cabeça (da regra) e à direita um corpo ou expansão. Por vezes, duas regras têm a mesma cabeça, como com ⟨*noun phrase*⟩. Há também duas regras para ⟨*noun*⟩ mas abreviamos combinando os corpos usando o símbolo da barra vertical ' | ' (pipe).[5]

Fazemos regras utilizando dois tipos diferentes de componentes. Aqueles escritos em fonte monoespaçada, como `young`, são do alfabeto Σ da linguagem. Estes são **terminais**. Os componentes escritos com parênteses angulares e em itálico, como ⟨*article*⟩, são **não terminais**. Estes são como variáveis, e são utilizados para passos intermediários.

Os dois símbolos "→" e "|" não são nem terminais nem não terminais. São **metacaracteres**, fazem parte da sintaxe das próprias regras.

Estas regras de reescrita regem a **derivação** de cadeias na linguagem. Sob a gramática do inglês, toda a derivação começa com ⟨*sentence*⟩. Ao longo do caminho, as cadeias intermediárias contêm uma mistura de não terminais e terminais. Todas as regras têm uma cabeça com um único não terminal. Assim, para derivar a sequência

4 Leia a seta em voz alta como "pode produzir", ou "pode expandir para", ou "pode ser construído como".
5 Leia em voz alta como "ou".

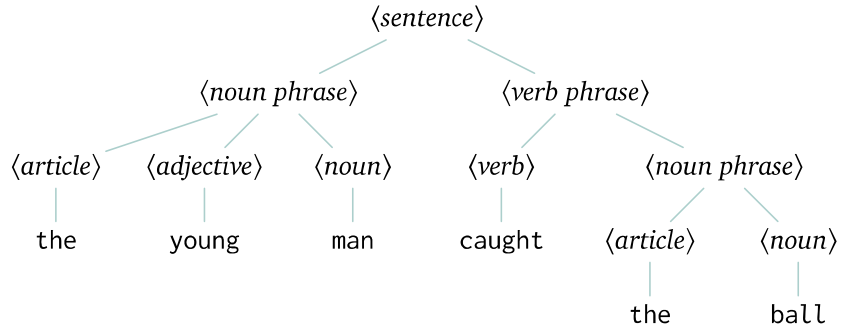
seguinte, escolha um não terminal na sequência atual e substitua um corpo de regras associado.

$$
\begin{aligned}
\langle \textit{sentence} \rangle &\Rightarrow \langle \textit{noun phrase} \rangle \, \langle \textit{verb phrase} \rangle \\
&\Rightarrow \langle \textit{article} \rangle \, \langle \textit{adjective} \rangle \, \langle \textit{noun} \rangle \, \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the} \, \langle \textit{adjective} \rangle \, \langle \textit{noun} \rangle \, \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young} \, \langle \textit{noun} \rangle \, \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young man} \, \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young man} \, \langle \textit{verb} \rangle \, \langle \textit{noun phrase} \rangle \\
&\Rightarrow \texttt{the young man caught} \, \langle \textit{noun phrase} \rangle \\
&\Rightarrow \texttt{the young man caught} \, \langle \textit{article} \rangle \, \langle \textit{noun} \rangle \\
&\Rightarrow \texttt{the young man caught the} \, \langle \textit{noun} \rangle \\
&\Rightarrow \texttt{the young man caught the ball}
\end{aligned}
$$

Note que a seta de linha única $\rightarrow$ é para regras enquanto a seta de linha dupla $\Rightarrow$ é para derivações.[6]

Essa derivação sempre substitui o não terminal mais à esquerda, por isso é uma **derivação mais à esquerda**. Contudo, em geral, poderíamos substituir qualquer não terminal.

A **árvore de derivação** ou **árvore de análise sintática** é uma representação alternativa.[7]



Definição.

Uma **gramática livre de contexto**, é uma 4-upla $G = \langle \Sigma, N, S, P \rangle$. Em primeiro lugar, Σ é um alfabeto, cujos elementos são os símbolos terminais. Em segundo lugar, N é um conjunto de *não terminais* ou *categorias sintáticas*. (Partimos do princípio que Σ e N são disjuntos e que nenhum deles contém metacaracteres). Terceiro, $S \in N$ é o *símbolo de início*. Quarto, P é um conjunto de *regras de produção* ou *regras de reescrita*.

Consideraremos o símbolo de início como sendo a cabeça da primeira regra.

Exemplo 11.2.2

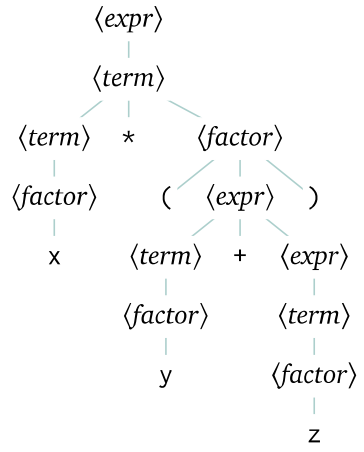
Esta gramática livre de contexto descreve expressões algébricas que envolvem apenas adição, multiplicação, e parênteses.

6 Leia '⇒' em voz alta como "deriva" ou "expande para".
7 Os termos "terminal" e "não terminal" vêm de onde os componentes se encontram nesta árvore.

⟨*expr*⟩ → ⟨*term*⟩ + ⟨*expr*⟩ | ⟨*term*⟩
⟨*term*⟩ → ⟨*term*⟩ * ⟨*factor*⟩ | ⟨*factor*⟩
⟨*factor*⟩ → (⟨*expr*⟩) | a | b | ... | z

Aqui está uma derivação da cadeia x*(y+z).

⟨*expr*⟩ ⇒ ⟨*term*⟩
⇒ ⟨*term*⟩ * ⟨*factor*⟩
⇒ ⟨*factor*⟩ * ⟨*factor*⟩
⇒ x * ⟨*factor*⟩
⇒ x * (⟨*expr*⟩)
⇒ x * (⟨*term*⟩ + ⟨*expr*⟩)
⇒ x * (⟨*term*⟩ + ⟨*term*⟩)
⇒ x * (⟨*factor*⟩ + ⟨*term*⟩)
⇒ x * (⟨*factor*⟩ + ⟨*factor*⟩)
⇒ x * (y + ⟨*factor*⟩)
⇒ x * (y + z)



Nesse exemplo, as regras para ⟨*expr*⟩ e ⟨*term*⟩ são recursivas. Mas não ficamos presos a uma regressão infinita porque a questão não é se poderíamos continuar perversamente a expandir ⟨expr⟩ para sempre; a questão é se, dada uma sequência como x*(y+z), se pode encontrar uma derivação que termine.

No exemplo anterior, os não terminais como ⟨*expr*⟩ ou ⟨*term*⟩ descrevem o papel desses componentes na linguagem, tal como os fragmentos de gramática inglesa ⟨noun phrase⟩ e ⟨article⟩. Mas nos exemplos a seguir utilizamos frequentemente pequenas gramáticas cujos terminais e não terminais não têm nenhum significado particular. Para estes casos, passamos frequentemente da notação verbosa como '⟨sentence⟩ → ⟨noun phrase⟩ ⟨verb phrase⟩' para a escrita de letras isoladas, com não terminais em maiúsculas e terminais em minúsculas.

Exemplo 11.2.3

Esta gramática de duas regras tem um não terminal, S.

S → aSb | ε

Aqui está uma derivação da cadeia a^2b^2.

S ⇒ aSb ⇒ aaSbb ⇒ aaεbb ⇒ aabb

Do mesmo modo, S ⇒ aSb ⇒ aaSbb ⇒ aaaSbbb ⇒ aaaεbbb ⇒ aaabbb é uma derivação de a^3b^3. Para esta gramática, as cadeias deriváveis têm a forma a^nb^n para $n \in \mathbb{N}$.

A seguir damos uma descrição completa de como as regras de produção regem as derivações. Cada regra numa gramática livre de contexto tem a forma 'cabeça → corpo' onde a cabeça é constituída por um único não terminal. O corpo é uma sequência de terminais e não-terminais. Cada etapa de uma derivação tem a forma abaixo, onde τ_0 e τ_1 são sequências de terminais e não-terminais.

$$\tau_0 \hat{} \text{cabeça} \hat{} \tau_1 \Rightarrow \tau_0 \hat{} \text{corpo} \hat{} \tau_1$$

Ou seja, se houver uma correspondência para a cabeça da regra, então podemos substituí-la pelo corpo.

Sempre que σ_0, σ_1 são sequências de terminais e não terminais, se estiverem relacionados por uma sequência de etapas de derivação, então podemos escrever $\sigma_0 \Rightarrow^* \sigma_1$. Onde $\sigma_0 = S$ é o símbolo de início, se houver uma derivação $\sigma_0 \Rightarrow^* \sigma_1$ que termina com uma sequência de terminais $\sigma_1 \in \Sigma^*$ então dizemos que σ_1 tem uma derivação a partir da gramática.[8]

Esta descrição é similar aquela em que detalhamos, no capítulo anterior, como as instruções de uma máquina Turing determinam a evolução da sequência de configurações, o que corresponde a uma computação. Ou seja, as regras de produção são como um programa, dirigindo uma derivação. No entanto, uma diferença daquela descrição é que ali as máquinas Turing são determinísticas, de modo que a partir de uma determinada cadeia de entrada há uma determinada sequência de configurações. Aqui, a partir de um dado símbolo de início, uma derivação pode ramificar-se para ir para muitas cadeias finais diferentes.

Definição.

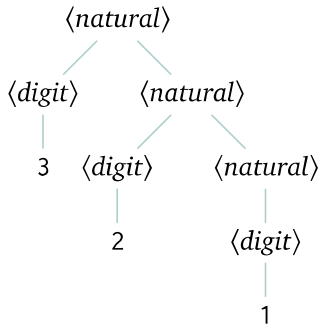
A **linguagem derivada de uma gramática** é o conjunto de cadeias de terminais com derivações que começam com o símbolo de início.

Exemplo 11.2.4

Esta linguagem gramatical é o conjunto de representações de números naturais.

⟨*natural*⟩ → ⟨*digit*⟩ | ⟨*digit*⟩⟨*natural*⟩
⟨*digit*⟩ → `0` | ... | `9`

Esta é uma derivação para a cadeia `321`, junto a sua árvore de análise sintática.



⟨natural⟩ ⇒ ⟨*digit*⟩⟨*natural*⟩
⇒ 3⟨*natural*⟩
⇒ 3⟨*digit*⟩⟨natural⟩
⇒ 32⟨natural⟩
⇒ 32⟨*digit*⟩
⇒ 321

Exemplo 11.2.5

A linguagem desta gramática é o conjunto de cadeias que representam números naturais em unário.

8 Esta definição de regras, gramáticas, e derivações é suficiente para nós, mas não é a mais geral. Uma definição mais geral permite cabeças no formato $\sigma_0 X \sigma_1$, onde σ_0 e σ_1 são cadeias de terminais. (Os σ_i's pode ser vazios.) Por exemplo, considere esta gramática: (i) S → `a`BS`c` | `abc`, (ii) B`a` → `a`B, (iii) B`b` → `bb`. A regra (ii) diz que se você vir uma cadeia com algo seguido por `a`, então pode substituir essa cadeia por `a` seguido por essa coisa. As gramáticas com cabeças na forma $\sigma_0 X \sigma_1$ são sensíveis ao contexto porque só podemos substituir o X no contexto de σ_0 e σ_1. Estas gramáticas descrevem mais linguagens do que as que estamos utilizando livres de contexto. Mas a nossa definição satisfaz as nossas necessidades atuais e corresponde à classe de gramáticas que você mais verá na prática.

⟨natural⟩ → ε | 1⟨natural⟩

Exemplo 11.2.6

Qualquer linguagem finita é derivada de uma gramática. Esta fornece a linguagem de todas as cadeias de bits de comprimento 2, utilizando a abordagem de força bruta de simplesmente listar todas as cadeias que são membros.

S → 00 | 01 | 10 | 11

Já a gramática abaixo fornece as cadeias de bits de comprimento 3, utilizando os não terminais para manter a contagem.

A → 0B | 1B
B → 0C | 1C
C → 0 | 1

Exemplo 11.2.7

Para esta gramática

S → aSb | T | U
T → aS | a
U → Sb | b

uma alternativa é substituir T e U pelas suas expansões para obter isto.

S → aSb | aS | a | Sb | b

Ela gera a linguagem L = $\{a^i b^j \in \{a, b\}^* \mid i \neq 0 \text{ ou } j \neq 0\}$.

O exemplo anterior é o primeiro em que a linguagem gerada não é clara, por isso faremos uma verificação formal. Mostraremos contenção mútua, que a linguagem gerada é um subconjunto de L e que é também um superconjunto. A regra que elimina T e U mostra que qualquer etapa de derivação τ_0^cabeça^$\tau_1 \Rightarrow \tau_0$^corpo^$\tau_1$ apenas adiciona a's à esquerda e b's à direita, portanto toda cadeia na linguagem tem a forma $a^i b^j$. Essa mesma regra mostra que em qualquer derivação terminada S deve eventualmente ser substituída ou por a ou b. Juntos, estes dois dão que a linguagem gerada é um subconjunto de L.

Para contenção no sentido contrário, provaremos que cada σ ∈ L tem uma derivação. Utilizaremos indução no comprimento |σ|. Pela definição de L, o caso base é |σ| = 1. Neste caso, ou σ = a ou σ = b, cada um dos quais tem obviamente uma derivação.

Para o passo indutivo, suponha que cada cadeia de L de comprimento $k = 1, \ldots, k = n$ tem uma derivação para $n \geq 1$ e seja $n + 1$ o comprimento de σ. Escreva $\sigma = a^i b^j$. Há três casos: ou $i > 1$, ou $j > 1$, ou $i = j = 1$. Se $i > 1$ então $\hat{\sigma} = a^{i-1} b^j$ é uma cadeia de comprimento n, então pela hipótese indutiva tem uma derivação $S \Rightarrow \ldots \Rightarrow \hat{\sigma}$. Prefixando essa derivação com um passo S ⇒ aS colocará um a adicional à esquerda. O caso $j > 1$ funciona da mesma maneira, e $\sigma = a^1 b^1$ é fácil.

Exemplo 11.2.8

O fato das derivações poderem ir por mais do que um caminho leva a uma questão importante com as gramáticas, que elas podem ser ambíguas. Considere este fragmento de uma gramática para declarações `if` numa linguagem tipo C

⟨*stmt*⟩ → `if` ⟨*bool*⟩ ⟨*stmt*⟩
⟨*stmt*⟩ → `if` ⟨*bool*⟩ ⟨*stmt*⟩ `else` ⟨*stmt*⟩

e esta cadeia de código.

```
if enrolled(s) if studied(s) grade='P' else grade='F'
```

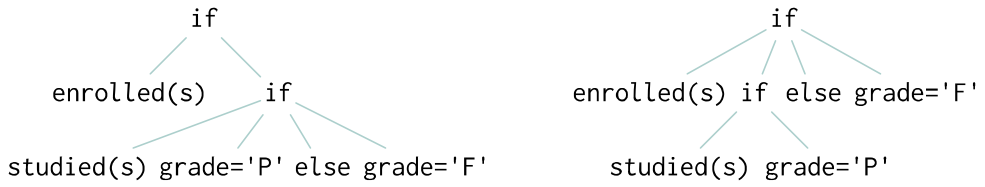
Aqui estão as duas primeiras linhas de uma derivação:

⟨*stmt*⟩ ⇒ `if` ⟨*bool*⟩ ⟨*stmt*⟩
⇒ `if` ⟨*bool*⟩ `if` ⟨*bool*⟩ ⟨*stmt*⟩ `else` ⟨*stmt*⟩

e aqui estão as duas primeiras de uma outra:

⟨*stmt*⟩ ⇒ `if` ⟨*bool*⟩ ⟨*stmt*⟩ `else` ⟨*stmt*⟩
⇒ `if` ⟨*bool*⟩ `if` ⟨*bool*⟩ ⟨*stmt*⟩ `else` ⟨*stmt*⟩

Ou seja, não podemos dizer se o `else` na linha de código está associado ao primeiro `if` ou ao segundo. As árvores de análise sintática resultantes para a linha de código completa dramatizam a diferença



assim como estas cópias da cadeia de código identada para mostrar a associação.

```
if enrolled(s)
    if studied(s)
        grade='P'
    else
        grade='F'
```

```
if enrolled(s)
    if studied(s)
        grade='P'
else
    grade='F'
```

Obviamente, esses programas se comportam de forma diferente. Isto é conhecido como um **else pendente**.

Exemplo 11.2.9

Esta gramática para expressões elementares de álgebra

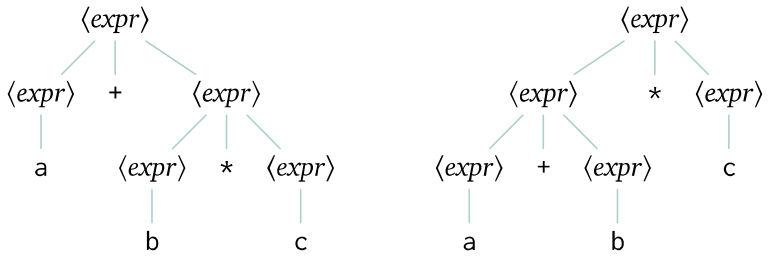
⟨*expr*⟩ → ⟨*expr*⟩ + ⟨*expr*⟩
| ⟨*expr*⟩ * ⟨*expr*⟩
| (⟨*expr*⟩) | a | b | ... z

é ambígua porque a+b*c tem duas derivações mais à esquerda.

⟨*expr*⟩ ⇒ ⟨*expr*⟩+⟨*expr*⟩ ⇒ a + ⟨*expr*⟩
⇒ a + ⟨*expr*⟩*⟨*expr*⟩ ⇒ a + b * ⟨*expr*⟩ ⇒ a + b * c

⟨*expr*⟩ ⇒ ⟨*expr*⟩*⟨*expr*⟩ ⇒ ⟨*expr*⟩ + ⟨*expr*⟩ * ⟨*expr*⟩
⇒ a + ⟨*expr*⟩*⟨*expr*⟩ ⇒ a + b * ⟨*expr*⟩ ⇒ a + b * c

As duas correspondem a árvores de análise sintática diferentes:



Mais uma vez, a questão é que temos dois comportamentos diferentes. Por exemplo, substitua a por 1, b por 2, e c por 3. A árvore esquerda fornece 1 + (2 · 3) = 7 enquanto que a árvore da direita fornece (1 + 2) · 3 = 9.

Em contraste, esta gramática para expressões elementares de álgebra é isenta de qualquer ambiguidade.

⟨*expr*⟩ → ⟨*expr*⟩ + ⟨*term*⟩
| ⟨*term*⟩
⟨*term*⟩ → ⟨*term*⟩ * ⟨*factor*⟩
| ⟨*factor*⟩
⟨*factor*⟩ → (⟨*expr*⟩)
| a | b | ... | z

A escolha de gramáticas que não sejam ambíguas é importante na prática.

11.3. BNF

Introduziremos algumas conveniências de notação gramatical que são amplamente utilizadas. Juntas são chamadas Backus-Naur form, BNF.

O estudo da gramática, as regras para a estrutura das frases e a formação de sentenças, tem uma longa história, que remonta já ao século V a.C. Os matemáticos, incluindo A Thue e E Post, começaram a sistematizá-lo como regras de reescrita no início dos anos 1900. A variante BNF foi produzida por J Backus nos finais da década de 1950 como parte da concepção de uma das primeiras linguagens de programação, o ALGOL60. Desde então, estas regras tornaram-se uma forma padrão de expressar gramáticas.

Uma diferença em relação à subsecção anterior é uma pequena alteração tipográfica. Originalmente os metacaracteres não eram datilografáveis com um teclado padrão. A vantagem de ter metacaracteres que não se encontram num teclado é que muito provavelmente todos os caracteres da língua são datilografáveis. Portanto, não há necessidade de distinguir, digamos, entre o carácter de barra vertical | quando usado como parte de uma linguagem e quando usado como um metacaractere. Mas a desvanta-

gem está em ter de digitar o não digitável. No final, a conveniência de ter caracteres dactilografáveis prevaleceu sobre o ganho técnico de poder distinguir tipograficamente os metacaracteres. Por exemplo, durante muito tempo não havia setas num teclado padrão, por isso em vez do símbolo de seta '→', a BNF usa '::='. (Estes ajustes foram feitos por P Naur, como editor do relatório ALGOL60.)[9]

A BNF é ao mesmo tempo clara e concisa, ela pode expressar a gama de linguagens que normalmente queremos expressar, e traduz-se facilmente em um analisador sintático (parser).[10] Isto é, a BNF é uma correspondência de impedância — ela se encaixa com o que normalmente queremos fazer. Aqui incorporaremos algumas extensões para agrupamento e replicação que são semelhantes ao que você verá na prática.

Exemplo 11.3.1

Esta é uma gramática BNF para números reais com uma parte decimal finita. Veja as regras para ⟨natural⟩ no Exemplo 11.2.4.

⟨*start*⟩ ::= −⟨*fraction*⟩ | +⟨*fraction*⟩ | ⟨*fraction*⟩
⟨*fraction*⟩ ::= ⟨*natural*⟩ | ⟨*natural*⟩`.`⟨*natural*⟩

Esta uma derivação mais à direita para 2.718:

⟨*start*⟩ ⇒ ⟨*fraction*⟩ ⇒ ⟨*natural*⟩`.`⟨*natural*⟩
⇒ ⟨*natural*⟩`.`⟨*digit*⟩⟨*natural*⟩ ⇒ ⟨*natural*⟩`.`⟨*digit*⟩⟨*digit*⟩⟨*natural*⟩
⇒ ⟨*natural*⟩`.`⟨*digit*⟩⟨*digit*⟩⟨*digit*⟩ ⇒ ⟨*natural*⟩`.`⟨*digit*⟩⟨*digit*⟩8
⇒ ⟨*natural*⟩`.`⟨*digit*⟩18 ⇒ ⟨*natural*⟩`.718` ⇒ `2.718`

Aqui está uma derivação para 0.577 que não é nem a mais à esquerda nem a mais à direita.

⟨*start*⟩ ⇒ ⟨*fraction*⟩ ⇒ ⟨*natural*⟩`.`⟨*natural*⟩
⇒ ⟨*natural*⟩`.`⟨*digit*⟩⟨*natural*⟩ ⇒ ⟨*natural*⟩`.5`⟨*natural*⟩
⇒ ⟨*natural*⟩`.5`⟨*digit*⟩⟨*natural*⟩ ⇒ ⟨*digit*⟩`.5`⟨*digit*⟩⟨*natural*⟩
⇒ ⟨*digit*⟩`.5`⟨*digit*⟩⟨*digit*⟩ ⇒ ⟨*digit*⟩`.5`⟨*digit*⟩7 ⇒ `0.5`⟨*digit*⟩7
⇒ `0.577`

Exemplo 11.3.2

O tempo é um problema de engenharia difícil. Uma questão é a representação do tempo e uma solução nessa área é a RFC 3339, *Date and Time on the Internet: Timestamps*. Ela utiliza strings tais como `1958-10-12T23:20:50.52Z`. Aqui está uma gramática na BNF. Esta gramática inclui algumas extensões de metacaracteres discutidas a seguir.

⟨*date-fullyear*⟩ ::= ⟨*4-digits*⟩
⟨*date-month*⟩ ::= ⟨*2-digits*⟩
⟨*date-mday*⟩ ::= ⟨*2-digits*⟩

9 Há outras questões tipográficas que surgem com as gramáticas. Enquanto que muitos autores escrevem não terminais com parênteses angulares, como nós, outros usam outras convenções, tais como um estilo ou cor de letra diferentes.

10 A BNF é apenas vagamente definida. Várias variantes têm padrões, mas o que se vê muitas vezes não está em conformidade com nenhum padrão publicado.

⟨*time-hour*⟩ ::= ⟨*2-digits*⟩
⟨*time-minute*⟩ ::= ⟨*2-digits*⟩
⟨*time-second*⟩ ::= ⟨*2-digits*⟩
⟨*time-secfrac*⟩ ::= `.` ⟨*1-or-more-digits*⟩
⟨*time-numoffset*⟩ ::= (`+` | `-`) ⟨*time-hour*⟩ : ⟨*time-minute*⟩
⟨*time-offset*⟩ ::= `Z` | ⟨*time-numoffset*⟩
⟨*partial-time*⟩ ::= ⟨*time-hour*⟩ `:` ⟨*time-minute*⟩ `:` ⟨*time-second*⟩[⟨*time-secfrac*⟩]
⟨*full-date*⟩ ::= ⟨*date-fullyear*⟩ `-` ⟨*date-month*⟩ `-` ⟨*date-mday*⟩
⟨*full-time*⟩ ::= ⟨*partial-time*⟩ ⟨*time-offset*⟩
⟨*date-time*⟩ ::= ⟨*full-date*⟩ `T` ⟨*full-time*⟩

Este exemplo mostra uma notação BNF na regra ⟨*time-numoffset*⟩, onde os parênteses são utilizados como metacaracteres para agrupar uma escolha entre os terminais `+` e `-`. Ele também mostra uma outra extensão na regra ⟨*partial-time*⟩, que inclui colchetes como metacaracteres. Estes denotam que o ⟨*time-secfrac*⟩ é opcional.

Os colchetes são uma construção muito comum: outro exemplo é esta sintaxe para `if … then …` com um `else …` opcional:

⟨*if-stmt*⟩ ::= `if` ⟨*boolean-expr*⟩ `then` ⟨*stmt-sequence*⟩
[`else` ⟨*stmt-sequence*⟩] ⟨*end if*⟩`;`

Para mostrar repetição, a BNF pode usar um fecho de Kleene sobrescrito * para significar ‘zero ou mais’ ou um $^+$ para significar ‘um ou mais’. Isto mostra parênteses e repetição.

⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)*

Cada uma destas construções de extensão não é necessária uma vez que podemos expressá-las em BNF simples, sem as extensões. Por exemplo, poderíamos substituir a regra anterior por esta.

⟨*identifier*⟩ ::= ⟨*letter*⟩ | ⟨*letter*⟩ ⟨*atoms*⟩
⟨*atoms*⟩ ::= ⟨*letter*⟩ ⟨*atoms*⟩ | ⟨*digit*⟩ ⟨*atoms*⟩ | ε

Mas estas construções surgem com frequência suficiente para que a adoção de uma abreviatura seja conveniente.

Exemplo 11.3.3

Esta gramática para números de ponto flutuante em Python mostra as três abreviaturas.

⟨*floatnumber*⟩ ::= ⟨*pointfloat*⟩ | ⟨*exponentfloat*⟩
⟨*pointfloat*⟩ ::= [⟨*intpart*⟩] ⟨*fraction*⟩ | ⟨*intpart*⟩ `.`
⟨exponentfloat⟩ ::= (⟨intpart⟩ | ⟨pointfloat⟩) ⟨exponent⟩
⟨intpart⟩ ::= ⟨digit⟩$^+$
⟨fraction⟩ ::= `.` ⟨digit⟩$^+$
⟨exponent⟩ ::= (`e`| `E`) [`+` | `-`] ⟨digit⟩$^+$

Como parte da regra ⟨*pointfloat*⟩, a primeira ⟨*intpart*⟩ é opcional. Uma ⟨*intpart*⟩ consiste em um ou mais dígitos. E uma expansão de ⟨*exponent*⟩ deve começar com uma escolha entre `e` ou `E`.

Observação. A passagem da gramática para um parser para essa gramática é mecânica. Escrevemos um programa que toma como entrada uma gramática (por exemplo em BNF) e fornece como saída o código fonte de um programa que analisará arquivos seguindo o formato dessa gramática. Este é um **gerador de parser**, por vezes chamado de **compilador-de-compilador** (embora esse termo seja sexy, é também impreciso porque um parser é apenas parte de um compilador)[11].

11.4. Créditos

Todas as seções, foram adaptadas (traduzidas e modificadas) de [1], que está disponível sob a licença Creative Commons Attribution-ShareAlike 4.0 (CC BY-SA 4.0).

11.5. Referências

1. Hefferon, Jim. *Theory of Computation: Making Connections*. Disponível em http://hefferon.net/computation/

11.6. Licença

11 Um exemplo de tal programa é o GNU Bison, que acompanha praticamente todas distribuições do GNU/Linux.