Linguagens formais e autômatos

Linguagens formais e autômatos

Alex de Vasconcellos Garcia
Edward Hermann Haeusler

Dados Internacionais de Catalogação na Publicação (CIP)

Garcia, Alex de Vasconcellos
G216l Linguagens formais e autômatos / Alex de Vasconcellos Garcia, Edward Hermann Haeusler. – Londrina : Editora e Distribuidora Educacional S.A., 2017.
208 p.

ISBN 978-85-522-0189-2

1. Linguagens formais. I. Haeusler, Edward Hermann. II. Título.

CDD 629

2017
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Palavras do autor

O ser humano parece ter uma característica que nenhuma outra espécie animal sobre a face da Terra possui: nós somos capazes de nos expressarmos através de uma linguagem, e mais do que isso, cada grupo humano teve historicamente um desenvolvimento próprio e original de uma linguagem específica do grupo. Na China desenvolveram-se o Mandarim e o Cantonês; no Oriente Médio tivemos o desenvolvimento das línguas semíticas; na Europa surgiram muitas outras línguas. Ninguém duvida hoje que todos os povos do mundo possam se comunicar oralmente entre eles de forma precisa. Sem dúvida, todas as linguagens que surgiram nos diversos grupos possuem uma estrutura comum: orações são formadas com sujeito e predicado. Um predicado tem pelo menos um verbo e o sujeito pode ser formado de outros constituintes, como um nome próprio ou um pronome. Estes conceitos estão presentes em todas as línguas naturais conhecidas. São conceitos gramaticais universais. Em 1950, Noam Chomsky utilizou-se deste conceito de gramática universal para explicar como o ser humano adquire linguagem. Ele lança a tese de que o ser humano possui a capacidade de reconhecer padrões gramaticais e linguísticos por possuir uma espécie de gramática universal em sua mente. Trata-se, portanto, de uma habilidade inata, ou seja, não é adquirida após o nascimento. O trabalho de Chomsky nos anos 50 definiu o conceito de linguagem formal, do qual as linguagens naturais e as de programação são casos particulares. Aproveitando-se de modelos computacionais já conhecidos na época, ele os associou, em níveis de complexidade, às respectivas classes de linguagens que eles podem processar. O modelo mais simples possui uma memória finita sendo capaz tão somente de ler palavras, símbolo por símbolo em uma só direção. O autômato mais sofisticado é capaz de ler, escrever e reescrever símbolos e operar nas duas direções durante seu processamento. Este último modelo também é conhecido como Máquina de Turing. Alan Turing criou este modelo computacional nos anos de 1930, demonstrando matematicamente a existência de máquinas programáveis.

O objetivo deste curso é introduzir o aluno ao conceito de linguagens formais, hierarquia de Chomsky e a tese de Turing-

Church, que diz que todo processo computável é Turing-computável. A tese de Church é um dos poucos enunciados que faz com que possamos chamar a área de atuação da computação como Ciência da Computação.

Sob um ponto de vista mais prático, na Unidade 1 estudaremos elementos de matemática discreta, conceitos básicos de linguagens (tais como: alfabetos, cadeias, linguagens formais e gramáticas) e ainda a Hierarquia de Chomsky. Na Unidade 2 veremos linguagens regulares, autômatos finitos e expressões regulares. O autômato finito é extensamente usado na análise léxica em compiladores. Na Unidade 3 veremos gramáticas e linguagens livres de contexto, bem como o autômato de pilha. Este último também é usado em compiladores, na fase de análise sintática. Entretanto um analisador sintático usa a versão determinística de algoritmos que serão estudados neste curso em sua versão não determinística. Finalmente, na Unidade 4 veremos linguagens sensíveis ao contexto, máquinas de Turing e recursividade.

Convidamos o leitor para o estudo desta fascinante área do conhecimento humano.

Introdução a linguagens formais e autômatos

Convite ao estudo

Esta primeira unidade introduz os conceitos fundamentais de linguagens formais e autômatos. Na primeira seção apresentaremos uma introdução aos fundamentos da matemática discreta, necessários para posteriormente apresentar os conceitos de linguagens formais e autômatos.

Na Seção 1.2 apresentaremos o conceito de linguagem, e, na Seção 1.3, apresentaremos os conceitos de gramáticas e linguagens geradas por uma gramática. Ainda na Seção 1.3, será apresentada a Hierarquia de Chomsky, que mostra a relação do tipo de gramática com o tipo de autômato necessário para a solução do problema da análise sintática.

Você se candidatou a um processo de seleção em uma instituição de pesquisa meteorológica, que trabalha com o processamento de grande quantidade de dados coletados em todo o mundo e os processa. Mil e duzentas pessoas ficaram interessadas pela vaga no recém-inaugurado escritório no Brasil. Após ser aprovado em um exame escrito e em uma dinâmica de grupo, você está entre os 10 finalistas para a vaga. Na etapa final da seleção, você deve escolher um tópico de Ciência da Computação e dar uma palestra de 3 horas sobre o tópico, indicando também como ele pode ser usado em seu trabalho na empresa.

Pronto para o desafio? Sigamos adiante.

Seção 1.1

Elementos de matemática discreta

Diálogo aberto

Em função das crescentes aplicações de computadores no nosso dia a dia, a organização *Association for Computing Machinery* (ACM) – passou a recomendar que todos os cursos de engenharia dos EUA incluíssem em seus currículos um curso de matemática discreta, pois se trata da matemática que fundamenta a Computação. Essa recomendação se deu no ano de 1968. Hoje em dia, após o surgimento de microcomputadores e da internet, essa recomendação é muito mais pertinente.

Lembramos que na etapa final do processo de seleção no qual você se encontra, você deve dar uma palestra de 3 horas sobre um tópico de ciência da computação à sua escolha. Buscando se diferenciar dos demais candidatos, você escolhe o tópico Linguagens Formais e Autômatos. Você possui três dias para preparar a palestra.

Uma vez que o público que irá assistir à sua palestra - e não somente os responsáveis pela sua seleção, mas também profissionais da empresa, com diferentes formações - você decide dedicar a primeira parte de sua palestra a conceitos de matemática discreta que serão úteis para o entendimento de Linguagens Formais e Autômatos. Para tanto, você deve preparar uma apresentação com 10 slides sobre o tema.

Ao preparar a sua apresentação, tenha em mente o seu público-alvo, suponha que o mesmo não tem conhecimento prévio do assunto, procure entender quais são os assuntos mais importantes para preparar uma apresentação objetiva, que não ultrapasse o tamanho previsto. Como cativar um público leigo com um assunto tão diferente? Será possível fazer uma plateia tão diversa despertar interesse pelo tema? Está colocado o desafio.

Não pode faltar

A matemática discreta é a parte da matemática que está interessada no estudo de estruturas discretas (e não contínuas). Como os

computadores não podem representar números reais, a matemática discreta é a base não só para Linguagens Formais e Autômatos, mas também, para qualquer curso de Computação com ênfase em teoria. Nesta seção será feita a introdução aos fundamentos de matemática discreta. Os assuntos que veremos incluem conjuntos, relações e funções. Falaremos também sobre conjuntos enumeráveis, que é um tópico um pouco mais avançado, mas fundamental para o conteúdo do presente curso. As referências clássicas de matemática discreta são Menezes (2013) e Preparata e Yeh (1973).

A teoria de conjuntos é uma área relativamente recente da matemática. Seu principal autor, George Cantor, publicou seus trabalhos a partir de 1874 (CANTOR, 1874) e vale dizer que não foram muito bem aceitos no início, apesar de rigorosamente corretos. Sua contribuição ao conhecimento matemático é essencial para o nosso entendimento de coleções ou conjuntos com uma infinidade de elementos. Aparentemente, somente matemáticos e teólogos se interessam pelo infinito. Não fosse, entretanto, os trabalhos de Cantor, talvez a própria Computação não teria tido um desenvolvimento tão bem-sucedido no início do século passado.

Por serem objetos matemáticos primitivos, conjuntos não possuem definições formais, apesar de contarem com uma teoria bem definida. Isto quer dizer que não há como caracterizá-los por uma única propriedade, de forma simples. Você não vai encontrar facilmente na literatura matemática algo do tipo "Um conjunto é....". Aliás, partir para este tipo de abordagem pode trazer muitos problemas que estão totalmente fora do nosso escopo. São os famosos problemas de fundamentação da teoria dos conjuntos. Por enquanto é bom saber somente que eles existem, mas não interferem no uso corriqueiro e em aplicações da teoria dos conjuntos. Ficamos então com o entendimento que um conjunto é qualquer coleção (no sentido informal) de objetos, sendo estes abstratos, concretos, em quantidade finita ou não. Para indicar que um certo objeto a pertence à coleção (conjunto) A, utiliza-se a notação "$a \in A$". Esta notação "$a \in A$", é um predicado lógico, uma proposição, que é verdadeira ou falsa, conforme o elemento pertença ou não ao conjunto.

Pode-se especificar um conjunto particular via uma propriedade, desta forma o conjunto é formado somente por aqueles objetos que satisfaçam a esta propriedade. Se falarmos do conjunto dos times de

futebol que disputam a primeira divisão do Campeonato Brasileiro de Futebol em 2017, teremos um conjunto de 20 equipes. Outro exemplo de especificação é o conjunto dos objetos que são distintos deles mesmos. Como não há objeto que possa ser distinto de si mesmo, este conjunto é na realidade o conjunto vazio.

Formalmente, propriedades são especificadas através de uma linguagem lógica que faz uso dos conectivos lógicos: conjunção ($\wedge$), disjunção ($\vee$), negação ($\neg$) e implicação ($\rightarrow$). Além destes conectivos, utilizamos os quantificadores existencial ($\exists$) e universal ($\forall$) e o predicado identidade (=) que indica quando dois objetos são o mesmo objeto. Na prática, outros predicados são usados, sempre relativos ao domínio dos conjuntos que estão sendo especificados. Por exemplo, se estamos lidando com números Naturais, é usual fazermos uso do predicado de comparação (<) e, por vezes, das operações de soma e multiplicação. A especificação de conjuntos é feita formalmente, a partir de outro conjunto: o conjunto que fornece os elementos que satisfazem (ou não) a propriedade. Em teoria dos conjuntos, este mecanismo de especificação é denominado de Axioma da especificação. É importante que você se recorde da definição de ocorrência de variável livre e ligada em uma fórmula da lógica de primeira ordem. Confira nos exemplos a seguir se ainda se lembra disso. Como o único conceito primitivo na teoria dos conjuntos (TC) é a pertinência ($\in$), as fórmulas da linguagem básica de TC só tem ocorrência de "$\in$" para formar predicados atômicos, além da igualdade =. Observe que se $\varphi(x)$ for a fórmula $\neg(x = x)$ e A for um conjunto arbitrário, então $\{x / \neg(x = x)\}$ é o subconjunto vazio de A.

Assimile

Axioma da Especificação. Seja $\varphi(x)$ uma fórmula na linguagem, onde x ocorre livre em $\varphi(x)$. Seja A um conjunto. $\{a / a \in A$ e $\varphi(a)\}$ é um subconjunto de A.

Dado um subconjunto arbitrário B de um conjunto A, a propriedade $x \in B$ especifica B. Ou seja, B e $\{x / x \in A$ e $x \in B\}$ são o mesmo conjunto. Como identificar conjuntos, ou seja, como sabemos que dois conjuntos são iguais? Ora, dois conjuntos são iguais se ambos possuem os mesmos elementos. Além disso, dizemos que

A está incluído em B se todo elemento de A também é elemento de B. Usando este conceito de inclusão entre conjuntos, podemos dizer que dois conjuntos B e C são iguais, se, e somente se, $B \subseteq C$ e $C \subseteq B$.

Assimile

Inclusão entre conjuntos. Sejam B e C conjuntos. Dizemos que B está incluído em C, ou que C contém B, se, e somente se, todo elemento de B é também um elemento de C. Em lógica, isto é descrito como $\forall x((x \in B) \rightarrow (x \in C))$.

Quando for possível, podemos apresentar um conjunto enumerando seus elementos. Por exemplo, podemos falar do conjunto dos símbolos $\triangleright$, $\Diamond$ e $^{\circ}$. Isto é, o conjunto com 3 elementos que são enumerados anteriormente. Costumamos escrevê-los entre chaves para evitar ambiguidades. Daí o conjunto em questão é descrito por $\{\triangleright, \Diamond, ^{\circ}\}$. Na definição de produto cartesiano a seguir, usamos pares ordenados (b,c) de elementos de $a \in A$ e $b \in B$. Um par ordenado (a,b) é definido formalmente como sendo o conjunto $\{\{b\},\{b,c\}\}$, observe que a relação de inclusão entre conjuntos indica uma ordem na qual b vem antes de c no par ordenado. Daí o nome "par ordenado".

Sejam B e C subconjuntos de A. Definimos:

- União: $B \cup C$ é especificado por $\{x / x \in A$ e $((x \in B) \vee (x \in C))\}$
- Interseção: $B \cap C$ é especificado por $\{x / x \in A$ e $((x \in B) \wedge (x \in C))\}$
- Complemento em A: $C_A B$ é especificado por $\{x / x \in A$ e $\neg(x \in B)\}$
- Diferença: $B - C$ é especificado por $\{x / x \in A$ e $((x \in B) \wedge \neg(x \in C))\}$
- Produto Cartesiano: $B \times C$ é especificado por $\{(x,y) / x \in A$ e $(y \in B)\}$
- Conjunto potência: $\wp(A)$ é especificado por $\{B / B \subseteq A\}$

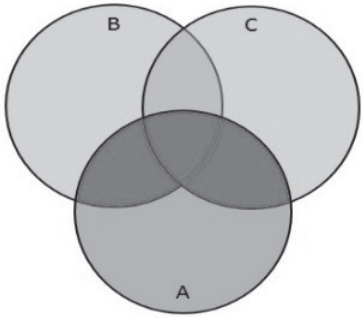
Existem algumas igualdades entre conjuntos que podem ser

verificadas para quaisquer conjuntos em geral. Sejam A, B e C conjuntos. Então as seguintes igualdades valem:

- $A \cap A = A$, $A \cap B = B \cap A$
- $A \cup B = B \cup A$
- $A \cap \varnothing = \varnothing$, $A \cup \varnothing = A$
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

A Figura 1.1 representa tais igualdades.

Figura 1.1 | Representação de conjuntos



Fonte: elaborada pelo autor (2017).

Entre os seres humanos, a relação de ser-mãe-de pode ser vista como o conjunto dos pares ordenados de seres humanos, onde o segundo é mãe do primeiro. Desta forma, podemos usar TC para falar de relações familiares, tais como a relação de "ser-mãe-de", "ser-pai-de", "ser-irmão-de", ou mesmo relações não familiares, tais como "ser-amigo-de", "ser-colega-de", "ser-conterrâneo-de".

Relações entre elementos de dois conjuntos, mesmo que estes sejam diferentes, são facilmente especificadas usando o produto cartesiano. Por exemplo, a relação de posse entre seres humanos e automóveis é um subconjunto de $Humanos \times Autos$. Alguns seres humanos não possuem carros. Por exemplo, se um certo h não possui carros, saberemos isso ao verificarmos que não existe nenhum par ordenado na forma (h, a) no conjunto $Humanos \times Autos$, onde a pode ser tomado como cada elemento em $Autos$.

Assimile

R é uma relação entre elementos de A e de B se, e somente se, $R \subseteq A \times B$.

Um tipo especial de relação é a funcional. Uma função de F de A em B é qualquer relação na qual para cada a existe um, e somente um, b, tal que $(a,b) \in F$. Isto também implica que todo elemento de a está relacionado com algum $b \in B$. Como cada a possui um, e somente um, b, podemos denominar tal b $F(a)$. Por exemplo, cada ser humano possui uma, e somente uma, mãe biológica. Quando Maria é mãe de João, podemos denotar Maria por $Mae-de(Joao)$. Neste caso, também se chama b de imagem de a via F.

Assimile

F é uma função de A em B se, e somente se, $F \subseteq A \times B$ e para todo $a \in A$ existe um, e somente um, $b \in B$, tal que $(a,b) \in F$. Obs.: Quando $(a,b) \in F$ e F é função, pode-se denotar b por $F(a)$.

Estamos interessados em dois tipos especiais de funções: as injetivas e as sobrejetivas.

Seja F uma função de A em B.

- F é injetiva se, e somente se, para todo a_1 e a_2, se $F(a_1) = F(a_2)$ então $a_1 = a_2$.

- F é sobrejetiva se, e somente se, para todo $b \in B$ existe $a \in A$ tal que $F(a) = b$.

As funções injetivas de A em B não podem ter nenhum elemento do contra-domínio (B) que não seja imagem de alguém do domínio (A). Funções injetivas têm imagens diferentes a partir de elementos de A que sejam diferentes. Visto como função a relação de "ser-mãe-de" não é injetiva. Pois caso Maria tenha João e Amália como filhos, então $Mae(Joao) = Mae(Amalia)$, mas João e Amália não são iguais.

Reflita

Analise o axioma da especificação e imagine porque ele pressupõe o conjunto A. Sua aplicação é imediata para a definição/especificação de subconjuntos de A. Seria possível prescindir de A? Ou seja, o que acontece se usarmos somente a propriedade representada por $\varphi(x)$, sem menção à A? Reflita sobre a seguinte tentativa de especificarmos um conjunto: $\{x / \neg(x \in x)\}$.

Observe a especificação do conjunto potência. Ela é um pouco diferente das outras especificações. Por que isto acontece?

Exemplificando

Na fórmula $\exists y(\neg(y = 0) \wedge y + x = z)$ a ocorrência de y em $y = 0$ é ligada ao quantificador existencial e as ocorrências de z e x são livres, pois não estão no escopo de nenhum quantificador. Na fórmula $\forall x((x \in z) \rightarrow (x \in y))$ as ocorrências de z e y são livres e as duas ocorrências de z são ligadas.

Quando queremos introduzir um objeto matemático, ou até mesmo um conceito matemático, podemos fazer isso essencialmente de duas formas distintas.

A primeira alternativa é definirmos este objeto usando objetos que já estão definidos. A segunda alternativa é fazemos um elenco de todas as propriedades relevantes que nosso objeto ou conceito satisfaz.

Por exemplo, sendo suc a função sucessor no conjunto dos números naturais, que a cada n associa $n+1$, a propriedade $\neg\exists y(suc(y) = x)$ só é verdadeira quando x é o número 0 (zero). Dizemos que $\neg\exists y(suc(y) = x)$ é uma definição do número zero, quando estamos no domínio dos números naturais, ou seja, a fórmula $\neg\exists y(suc(y) = x)$ define 0, pois é verdade se, e somente se, $x = 0$. Mostramos ser possível definir logicamente o número zero somente com o auxílio do conceito de "sucessor". Outro exemplo é, a partir da soma de números naturais e do número 0, definirmos a relação de ordem "menor que". Isto é feito com a fórmula $\exists y(\neg(y = 0) \wedge x + y = z)$, que indica que $x < z$. Esta forma de introdução de um objeto/conceito é exatamente o que se faz em

algumas linguagens de programação quando as usamos para definir um tipo de dado que não é explicitamente dado pelas mesmas. Por exemplo, quando fazemos um programa definindo números complexos como pares de números reais e as operações de soma, produto etc. através de funções.

Introduzir novos conceitos e objetos via definição é a forma mais usada por programadores e por matemáticos. Entretanto, nem sempre é possível sua aplicação. Quando não é possível aplicar definições? Quando o conceito ou objeto que você quer introduzir não pode ser definido a partir de outros conceitos mais básicos. Por exemplo, a relação "maior que" não pode ser definida se não tivermos a adição à nossa disposição. Neste caso, o melhor que podemos fazer é especificar as propriedades que são essenciais. Isto se faz teorizando o "maior que", indicando tratar-se de uma relação com certas propriedades. O resultado é uma Teoria que especifica as relações logicamente semelhantes à relação de "maior que". A lista de fórmulas/propriedades que especifica o conceito é de fato uma axiomatização da Teoria. A Teoria propriamente dita é formada por todas as propriedades que podem ser demonstradas a partir desta axiomatização. Estas propriedades são chamadas de teoremas.

Um teorema é uma proposição que possui uma demonstração. O exemplo mais conhecido e popular de teorema talvez seja o de Pitágoras: "Em um triângulo retângulo, o quadrado da hipotenusa é igual à soma dos quadrados dos catetos". Existem diversas provas deste teorema, algumas são totalmente visuais enquanto outras são totalmente descritas em linguagem natural e construídas na forma de argumentação. Uma argumentação, ou um argumento, é uma sequência de proposições onde se distingue uma conclusão e algumas hipóteses. A Lógica (com letra maiúscula, pois se trata de uma área de conhecimento humano) tem como objeto de estudo as argumentações válidas. Pode-se dizer que a Lógica é a área do conhecimento humano que estuda ou estabelece quando um argumento é bom ou válido. Em assuntos de Matemática, tais como o nosso estudo de Linguagens Formais e Autômatos, um argumento não é válido se ele não estabelece uma conclusão falsa a partir de premissas ou hipóteses verdadeiras. Argumentos, válidos ou não, são sequências de proposições, sendo que há uma proposição diferenciada - a conclusão -, e algumas proposições que são hipóteses. Proposições são sentenças que possuem valor de verdade, ou seja, podem ser falsas ou verdadeiras. Não é a toda sentença da linguagem

natural que podemos atribuir um valor de verdade. Por exemplo, as seguintes sentenças não podem ter valor de verdade: "Feche a porta!", "O programa parou?" e "Este carro azul estacionado à porta de sua casa". A primeira é um comando, a segunda uma pergunta e a terceira é uma descrição. Proposições são geralmente representadas por sentenças que declaram um estado de coisas (*state-of-affairs*), isto é, sentenças declarativas, tais como "O carro estacionado à porta de sua casa é azul", "A porta está fechada" e o "O programa parou". Podemos não saber se "A porta está fechada" é uma sentença verdadeira ou não, pois podemos não saber a que porta a sentença se refere. Mas, com certeza, saberemos dizer se a mesma é verdade ou não, uma vez que a referência à porta esteja totalmente determinada, o que incluirá conhecer o estado da mesma, isto é, se fechada ou não. A Lógica não tem a "verdade" como objeto de estudo. Ela se utiliza de um conceito arbitrado de verdade para estudar quando um argumento é válido ou bom. Argumentos são formados de uma ou mais premissas e de uma conclusão.

Sendo P_i as premissas e C a conclusão, dizemos que um argumento $P_1, P_2, \supset P_n \vdash$ é válido, se, e somente se, em qualquer situação em que cada uma das P_i´s é verdadeira, então C também é verdadeira.

Argumentos válidos são devido a sua forma, e não ao seu conteúdo, ou a verdade factual das sentenças que o formam. Por exemplo, o argumento "Todo ser humano é mortal, bombeiros são seres humanos, então bombeiros são mortais" e o argumento "Todo guerreiro é corajoso, covardes são guerreiros, então covardes são corajosos" são na realidade instâncias do mesmo argumento formal, que é "Todo B é A, c´s são B´s, então c´s são A´s". No entanto, a primeira instância do argumento tem uma conclusão verdadeira no mundo atual, enquanto a segunda tem uma conclusão falsa. Isto se deve ao fato de uma das premissas não poder ser verdadeira no estado de coisas que são atribuídas ao nosso dia a dia. Ambos os casos usam o mesmo argumento formal, que é válido ou bom. Um exemplo de um argumento inválido ou ruim é: Alguns franceses são europeus, alguns parisienses são europeus, então alguns parisienses são franceses. Note que a versão formal deste argumento é "Alguns A´s são B´s e alguns C´s são B´s, então alguns C´s são A´s". Interpretando A como o conjunto dos homens e C como o das mulheres e B como seres humanos, teremos um estado de coisas em que as premissas são verdadeiras e a conclusão falsa, portanto, o argumento é ruim ou

inválido. Desde a Grécia antiga, onde a disciplina de Lógica teve um florescimento significativo, os argumentos utilizados na matemática passaram a ser formados de padrões lógicos, ou argumentos formais válidos, bem estabelecidos de forma a que o simples encadeamento destes argumentos (válidos) garantem a conclusão verdadeira a partir de premissas (supostas) como verdadeiras. Esta atitude lógico-demonstrativa é que deu suporte à matemática como "ciência" das proposições necessárias. A forma com que se lida com as demonstrações matemáticas é a mesma desde Euclides (300 a.C). Por exemplo, uma proposição que é uma negação (não P), será sempre provada na forma "suponha P verdadeira e obtenha uma contradição, portanto P". Apenas a título de ilustração, vamos acompanhar uma das demonstrações mais antigas do conhecimento humano. Vamos provar que $\sqrt{2}$ não é um número racional. Esta prova se encontra nos Elementos de Euclides (livro X proposição 117). Vejamos uma versão moderna desta demonstração:

Suponha que $\sqrt{2}$ é racional, então existem a e b primos entre si, com b não nulo, tal que $\sqrt{2} = \frac{a}{b}$. Pela definição de $\sqrt{2}$, temos que $2 = \frac{a^2}{b^2}$, portanto $a^2 = 2b^2$, sendo um número par, pois quadrado de par é par. Como a é par, então $a = 2c$ para algum c e, portanto $4c^2 = 2b^2$, portanto, $2c^2 = b^2$. Concluímos que b^2 é par e, portanto b também é par. Como a e b são pares, não podem ser primos entre si. Contradição! Conclui-se que não existem tais a e b.

Desde Euclides, a matemática vem sendo construída por meio de axiomas, que representam teorias que nada mais são do que o conjunto de todos os teoremas que podem ser demonstrados a partir dos axiomas através de regras de inferência (argumentos formais) válidas.

Um exemplo em teoria dos conjuntos é a demonstração de que se $A \subseteq B$, então $A \cap C \subseteq B \cap C$. Esta demonstração ilustra como é o argumento típico para a prova de um condicional, ou seja, uma proposição na forma "Se P1 então P2". Vejamos:

Suponha que $A \subseteq B$, então todo $a \in A$ é tal que $a \in B$. Seja C um conjunto e considere um elemento $x \in A \cap C$. Pela definição de $\cap$ temos que $x \in A$ e $x \in C$. Pela afirmação acima $x \in A$ implica em $x \in B$, portanto temos que $x \in B$ e $x \in C$, daí $x \in B \cap C$.

Concluímos então que "Se $x \in A \cap C$, então $x \in B \cap C$". Portanto, temos $A \cap C \subseteq B \cap C$. Finalmente, concluímos que "Se $A \subseteq B$, então $A \cap C \subseteq B \cap C$". Neste curso vamos nos deparar algumas vezes com demonstrações como as que acompanhamos anteriormente.

Passamos agora a estudar algumas propriedades dos conjuntos infinitos. Suponha que o hotel Hilbert tenha uma quantidade infinita de quartos, numerados como $H_0, H_1, H_2, \ldots$ o hotel Cantor também tem uma quantidade infinita de quartos, numerados como $C_0, C_1, C_2, \ldots$. Ambos os hotéis estão lotados, há hospedes em todos os quartos. Devido a um vazamento de gás, o hotel Cantor é fechado e todos os hóspedes são realocados no hotel Hilbert. Para tanto, o gerente, Sr. Dedekind, rearranja os hóspedes da seguinte forma: hóspedes do hotel Hilbert que estão no quarto H_n devem se mudar para o quarto H_{2n} e hóspedes do hotel Cantor no quarto C_n devem se mudar para o hotel Hilbert no quarto H_{2n+1}.

Esse processo ilustra vários conceitos sobre conjuntos em primeiro lugar, o conjunto dos quartos em cada um dos hotéis é enumerável, porque existe um quarto para cada número natural. Um conjunto S é enumerável se existe uma função bijetora $f : \mathbb{N} \to S$.

Além disso, a solução dada pelo gerente do hotel mostra que a união de 2 conjuntos enumeráveis também é enumerável. Lembre-se: se os conjuntos S_1 e S_2 são enumeráveis então a sua união, $S_1 \cup S_2$ também é enumerável.

Um exemplo é o conjunto $\mathbb{Z}$ dos inteiros. Ele é enumerável porque você pode criar a função bijetora $f : \mathbb{N} \to \mathbb{Z}$ definindo $f(n) = n/2$ se n é par e $f(n) = -(n+1)/2$ se n é ímpar.

Reflita

Você acha que o conjunto dos racionais é enumerável? Lembre-se que um número racional pode ser escrito como o quociente de dois números inteiros.

Agora vamos pensar no conjunto de todas as cadeias de caracteres (*strings*) não vazias que podemos formar com os caracteres 'a', 'b' e 'c'. O nosso conjunto é $S = \{a, b, c, aa, ab, ac, ba, bb, bc, \ldots\}$. Observe que se você sabe escrever um programa de computador que imprime uma destas cadeias em cada linha, então podemos associar o natural

n à cadeia que aparece na n-ésima linha (supondo que as linhas são numeradas a partir do 0). Portanto, o conjunto anterior é enumerável.

Pesquise mais

Cantor publicou em 1891 uma prova de que o conjunto dos números reais não é enumerável usando a técnica da diagonalização. Pesquise na internet sobre essa demonstração. Ela é importante para a discussão do problema da decisão que será abordado no final do curso. O assunto pode ser encontrado no link disponível em: <http://bit.ly/2nDfVWu> Acesso em: 30 maio 2017.

Sem medo de errar

Você finalmente está preparando a apresentação para a etapa final do processo de seleção para a instituição de pesquisa meteorológica. Uma boa ideia é começar explicando a importância de Linguagens Formais e Autômatos. Você pode falar da Hierarquia de Chomsky e dizer que, para as linguagens geradas por gramáticas mais simples, existem reconhecedores que tem ampla aplicação em áreas como compiladores e reconhecimento de padrões, que é uma área de interesse da empresa. Por outro lado, o estudo de linguagens geradas por gramáticas mais complexas tem relação com o estudo dos limites tanto da computação como da computação eficiente (computabilidade e tratabilidade).

Lembre-se que, para conseguir fazer uma boa apresentação, você deve ter um bom entendimento dos conceitos. Se você tem alguma dúvida sobre algum conceito que está em sua apresentação, este é um bom momento para você pesquisar na internet ou procurar um professor para saná-la.

Você pode prosseguir a apresentação explicando que, uma vez que a computação não trata números reais, mas apenas números naturais, a base matemática para a teoria da computação como um todo, e para linguagens formais e autômatos em particular, é a matemática discreta.

Depois dessa introdução você pode passar a falar dos tópicos de matemática discreta que considera relevantes para o tratamento

de linguagens formais e autômatos. Você pode começar pela teoria dos conjuntos, usando-a para explicar os conceitos de relação e função, que devem ser apresentados com um nível de detalhe maior. Ao final do espaço dedicado à matemática discreta, você pode falar de conjuntos enumeráveis, que são fundamentais para entender conceitos de linguagens formais, como conjuntos recursivamente enumeráveis.

Avançando na prática

Matemática para a computação

Descrição da situação-problema

A empresa em que você trabalha tem um programa de captação de jovens talentos. Este programa se inicia com uma palestra que é ministrada em turmas de ensino médio, normalmente do primeiro ano. Cada membro de sua equipe vai apresentar um tema, tais como: Jogos de Entretenimento, Inteligência Artificial, Engenharia de Software e Matemática para a Computação. Sua equipe realiza um sorteio para ver quem apresenta que tema. Neste sorteio ficou a seu cargo o tema de Matemática para a Computação. O problema que você vai enfrentar é que sua palestra não pode ser chata, deve conter um desafio e uma curiosidade bastante relacionada com a computação. Seu desempenho, bem como dos outros membros de sua equipe, será avaliado por um questionário a ser respondido pela audiência. Para se preparar, você conversou com o professor de matemática e descobriu que o assunto que é estudado no início do primeiro ano é a Teoria de Conjuntos. Isto é, o que são conjuntos, o que são funções e relações entre conjuntos e o que é a cardinalidade de um conjunto. A partir daí, você lembrou que quase tudo o que se faz em computação passa pela programação de computadores, e que isto é sempre realizado através de uma linguagem de programação ou montagem. Isto pode sugerir que você pense em como relacionar linguagens de programação, ou linguagens em geral com computação. Você então se recorda das aulas de Linguagens Formais e Autômatos e Compiladores que você fez na faculdade e tem a lembrança que seu professor fez uma "brincadeira" sobre homens e barbeiros em uma cidade. Você forçou um pouco mais a memória e com o auxílio de suas notas de

aula, verificou não ser necessário mais que um conhecimento bem básico de Teoria de conjuntos para entender o tal paradoxo. A esta altura, você já lembrou que a "brincadeira" era um paradoxo, ou seja, uma aparente contradição. Este paradoxo tem consequências bastante importantes nos limites de uso de linguagens, inclusive de linguagens de programação, e você pode montar sua palestra pensando nisso.

Resolução da situação-problema

Para preparar a sua apresentação para o programa de captura de jovens talentos, você primeiro estuda o paradoxo do barbeiro: "Em uma cidade existe um barbeiro que barbeia todo homem que não se barbeia e somente estes. A questão é: Este barbeiro se barbeia?". É uma apresentação para o ensino médio, você deve detalhar o argumento que a partir do enunciado do paradoxo mostra que: "O barbeiro se barbeia, se, e somente se, ele não se barbeia". Qual o propósito disso? Qual o relacionamento deste paradoxo com o nosso curso?

Use um pouco da linguagem de teoria de conjuntos da seguinte forma: (1) Considere o conjunto H dos homens e uma função $b : H \to \wp(H)$, tal que, para todo homem $h \in H$, $b(h)$ é o conjunto de homens que são barbeados por h. Todos os homens que h barbeia estão neste conjunto. O paradoxo se forma quando você assume que qualquer conjunto arbitrário de homens tem que ser barbeado por um barbeiro específico. Isto é equivalente a dizer que b é uma bijeção, e que, portanto, existem tantos barbeiros quanto conjuntos de homens barbeados por um barbeiro só.

Usando de alguma retórica, você pergunta à audiência: em uma cidade podem existir tantos grupos de homens quanto homens? Em seguida, você lembra à audiência sobre o fato que em qualquer conjunto finito com n elementos a quantidade de subconjuntos possíveis é 2^n, portanto, não há como haver tantos subconjuntos de homens quanto homens. A grande surpresa é que isso também vale para conjuntos infinitos. Detalhe o argumento, agora só com base na existência de bijeção. Se tiver dificuldade, consulte as notas que usou em seu curso de linguagens formais e autômatos.

O fecho triunfal de sua apresentação acontecerá quando você passar a considerar programas em alguma linguagem formal tal

como uma linguagem de programação. Daí você dirá que um programa p **reconhece** outro programa q, como um paralelo com o "barbear" entre homens. Daí a função de barbear do caso com homens vai associar a cada programa o conjunto de programas que este reconhece, e somente este. Assim, repetindo os argumentos anteriores, você conclui que não existe programa que implemente a função b. Se existisse, tal programa reconheceria a si mesmo, se, e somente se, não reconhecesse a si mesmo. A mensagem de sua palestra é que existem tarefas impossíveis de serem realizadas por programas.

Faça valer a pena

1. Sejam A, B e C conjuntos. E considere a igualdade $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
Assinale a demonstração correta desta igualdade:
a) Como $A \cup (B \cap C)$ contém elementos de A, B e C e como $(A \cup B) \cap (A \cup C)$ também contém elementos de A, B e C então ambos têm elementos de A, B e C e, portanto, são conjuntos iguais.
b) $A \cup (B \cap C)$ tem elementos de A e de B ou de C, portanto, tem elementos de A e B ou de A e C, que são precisamente os elementos de $(A \cup B) \cap (A \cup C)$, portanto, os conjuntos são iguais.
c) Suponha que $x \in A \cup (B \cap C)$, portanto, $x \in A$ ou $x \in (B \cap C)$. Se $x \in B \cap C$, então x pertence a ambos, tanto B quanto C. Juntando tudo, temos que $x \in A$ ou $x \in B$ e que $x \in A$ ou $x \in C$ também. Neste caso $x \in (A \cup B) \cap (A \cup C)$, pois x é um elemento arbitrário.
d) $x \in A \cup (B \cap C)$ equivale a $x \in A$ ou $x \in (B \cap C)$. Por sua vez, $x \in B \cap C$ equivale a dizer que x pertence a ambos, ou seja, tanto B quanto C. Juntando tudo, temos que $x \in A$ ou $x \in B$ e que $x \in A$ ou $x \in C$ equivale a $x \in (A \cup B) \cap (A \cup C)$. Como x é arbitrário, então $x \in A \cup (B \cap C)$ é equivalente a $x \in (A \cup B) \cap (A \cup C)$.
e) Seja $x \in A \cup (B \cap C)$, então $x \in A$ ou $x \in B \cap C$, ou seja $x \in A$, ou, $x \in B$ e $x \in C$. Considere que $x \in (A \cup B) \cap (A \cup C)$, então $x \in A$ ou $x \in B$ e também que $x \in A$ ou $x \in C$. Com isso temos que $x \in B$ e $x \in C$ ou $x \in A$. Como x é arbitrário, provamos que os conjuntos são iguais.

2. Sejam A e B conjuntos.
Assinale a alternativa verdadeira:

a) $A \not\subset A \cup (A \cap A)$
b) Para todo o conjunto C $(B \cap C) \subseteq (A \cap C)$
c) $A \times (B \cup C) \not\subset (A \times B) \cup (A \times C)$
d) $\varnothing$ é uma relação de A para B.
e) Se $A \subsetneq B$ e B é finito, então existe uma função injetiva $f : B \to A$.

3. Dados os conjuntos $A = \{1, 2, 3\}$ e $B = \{1, 2\}$.
Assinale a alternativa verdadeira:
a) Existe somente uma função $f : B \to A$.
b) Existe somente uma função $f : A \to B$.
c) Existem 12 relações de A para B.
d) Existem 8 funções de A para B.
e) Existem 3 funções bijetoras de A para B.

Seção 1.2

Conceitos básicos de linguagens

Diálogo aberto

Lembre-se que você está preparando uma apresentação sobre Linguagens Formais e Autômatos para o processo de seleção para uma instituição de pesquisa meteorológica.

Nesta seção, você verá os conceitos fundamentais de Linguagens Formais e Autômatos, que serão usados durante todo o curso. Estes conceitos representam uma primeira definição formal do que é uma linguagem e podem ser usados em diversas áreas da ciência da computação, por exemplo, para a formalização de problemas. Esta primeira definição de linguagem deixa de fora o aspecto sintático de uma linguagem, o que será abordado na próxima seção, com a apresentação de gramáticas.

Você já preparou a primeira parte de sua apresentação, relativa aos elementos de matemática discreta e a mostrou a um antigo professor, que elogiou muito o seu trabalho, fazendo apenas algumas pequenas correções. Dando continuidade à preparação de sua apresentação, você passa a abordar os conceitos básicos de Linguagens Formais e Autômatos, como: alfabetos, palavras, cadeias e linguagens. Pense em como você pode apresentar o material de forma que se mostre útil para a instituição de pesquisa meteorológica. Você está indo muito bem e percebe que a apresentação será útil não só para valorizar sua imagem junto à empresa, como também para você organizar suas ideias para poder aplicá-las em seu trabalho. Mãos à obra!

Não pode faltar

Nesta seção vamos entrar no conteúdo mais específico de linguagens formais e autômatos, como: alfabetos, palavras, cadeias e linguagens.

Podemos definir uma linguagem como a expressão de ideias, usando símbolos (sejam eles escritos, orais, ou de outro tipo) que se agrupam segundo determinadas regras. Assim, uma linguagem tem aspectos léxicos (relacionados aos símbolos usados), sintáticos

(relacionados às regras) e semânticos (relacionados ao significado). O objetivo deste curso é nos aprofundarmos no aspecto sintático, em particular no problema da análise sintática, suas soluções para diferentes tipos de gramáticas, bem como as consequências destas soluções. Neste curso não será abordado o aspecto semântico de uma linguagem.

Mesmo sem considerar o aspecto semântico, os aspectos léxico e sintático de uma linguagem são bastante complexos. Em nossa primeira definição consideramos apenas o aspecto léxico. Dado um conjunto finito de símbolos, comumente chamado alfabeto, uma linguagem sobre esse alfabeto é um conjunto de sequências finitas de símbolos deste alfabeto. Vamos definir esses conceitos de forma mais cadenciada.

Assimile

Um alfabeto (chamado também de vocabulário) é um conjunto finito não vazio de símbolos.

A definição de alfabeto, bem como outras definições apresentadas aqui, pode ser encontrada em referências clássicas como: (MENEZES, 2000) e (HOPCROFT; MOTWANI; ULLMAN, 2002).

O alfabeto latino moderno é o seguinte conjunto de 26 símbolos: {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}. É comum representarmos o alfabeto pela letra Σ. Outros exemplos de alfabeto são:

$$\Sigma_1 = \{\alpha, \beta, \gamma, \delta, \ldots, \omega\}$$
$$\Sigma_2 = \{0,1\}$$

Com o primeiro você pode escrever palavras gregas e com o segundo você pode escrever palavras binárias (números na base 2).

Assimile

Uma cadeia de símbolos de um dado alfabeto (também chamada de *string*, palavra ou sentença) é uma sequência finita de símbolos deste alfabeto.

Para o alfabeto $\Sigma_1 = \{\alpha, \beta, \gamma, \delta, \ldots, \omega\}$ podemos escrever a cadeia "$\psi\omega$".

Para o alfabeto $\Sigma_2 = \{0,1\}$ podemos escrever a cadeia "10001".

A cadeia formada por uma sequência com nenhum símbolo é conhecida como a cadeia vazia. Representamos a cadeia vazia com o símbolo ϵ. Note que a cadeia vazia é uma cadeia, ou palavra, sobre qualquer alfabeto. Cadeias de símbolos, ou palavras, sendo sempre uma sequência finita de símbolos, possuem comprimento, que é a quantidade de símbolos que ocorrem na mesma.

Qualquer cadeia de símbolos tem um comprimento. Por exemplo, a cadeia "10001" tem comprimento 5. A cadeia vazia é normalmente representada em linguagens de programação como "".

Uma operação rotineira quando lidamos com material escrito é a justaposição de palavras de uma linguagem, produzindo novas palavras. Esta justaposição é uma operação tão básica que ela prescinde de uma possível gramática. Isto quer dizer que a operação de justaposição, ou de concatenação de palavras, pode ser realizada entre quaisquer palavras da linguagem. Em português não é qualquer justaposição de palavras que pode ser considerada uma palavra da língua portuguesa. Nossa definição de linguagem não pressupõe que esta tenha que ter gramática. Mais à frente veremos exemplos de linguagens que não possuem gramática. Assim, a operação de concatenação é tomada como sendo sempre definida para quaisquer duas palavras.

Assimile

Dadas duas cadeias, definimos sua concatenação como a justaposição de seus valores. Por exemplo, se ω_1 ="101" e ω_2 ="000", sua concatenação é "101000". Representamos a concatenação como $\omega_1 \circ \omega_2$ ou simplesmente $\omega_1\omega_2$.

Considere 3 cadeias ω_1, ω_2 e ω_3. Se formarmos a concatenação de ω_1 com ω_2 e depois concatenarmos ω_3 teremos a justaposição das 3 palavras como o resultado destas operações, a saber $\omega_1\omega_2\omega_3$. Isso é o mesmo que concatenar ω_2 e ω_3 e fazer em seguida a concatenação de ω_1 ou

resultado anterior. Em termos matemáticos, dizemos que a concatenação é um operador associativo, isto é, $\omega_1 \circ (\omega_2 \circ \omega_3) = (\omega_1 \circ \omega_2) \circ \omega_3$. A cadeia vazia também pode ser concatenada. Pense então:

Reflita

Que cadeia é $\epsilon \circ \omega$?

No entanto, prefixos de uma cadeia são as subsequências de símbolos do início da cadeia.

Assimile

Dadas duas cadeias, ω_1 e ω_2, dizemos que ω_1 é prefixo de ω_2 se existe uma cadeia ω_1 tal que $\omega_1 \circ \omega_3 = \omega_2$.

A cadeia "101" possui os seguintes prefixos: ϵ, "1", "10" e "101".

Os sufixos de uma cadeia são definidos de forma análoga, porém tomando as subsequências do final da cadeia. Deixamos a definição como exercício para o leitor. A cadeia "100" possui os seguintes sufixos: ϵ, "0", "00" e "100".

Finalmente podemos apresentar a definição de linguagens.

Assimile

Dado um alfabeto definimos uma linguagem sobre este alfabeto como um conjunto de cadeias sobre este alfabeto.

Para o alfabeto $\Sigma_2 = \{0,1\}$ temos infinitas linguagens possíveis, entre elas:

$$L_1 = \varnothing$$

$L_2 = \{\epsilon\}$
$L_3 = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

A primeira linguagem não possui cadeia. A segunda linguagem possui apenas a cadeia vazia, enquanto a última possui todas as cadeias possíveis com símbolos do alfabeto Σ_2. Observe que a linguagem vazia, $\varnothing$, e a linguagem que só tem a palavra vazia, $\{\epsilon\}$, são diferentes, por quê?

Reflita

Qual seria o alfabeto adequado para ser usado no português? E qual seria o alfabeto adequado para ser usado na linguagem de programação C?

Sabemos que podemos concatenar duas cadeias. Esta operação pode ser estendida para uma linguagem. Definimos a concatenação de duas linguagens como a linguagem cujas cadeias são todas as possíveis concatenações entre cadeias da primeira linguagem com cadeias da segunda linguagem.

Assimile

Dadas as linguagens L_1 e L_2, definimos sua concatenação como a linguagem $L_1 \circ L_2 = \{\omega_1 \circ \omega_2 \mid \omega_1 \in L_1 \text{ e } \omega_2 \in L_2\}$.

Se L_1 é uma linguagem finita com n cadeias e L_2 é uma linguagem finita com m cadeias, então quantos elementos possui $L_1 \circ L_2$?

Reflita

Que linguagem é $L_1 \circ \varnothing$?

Quando repetimos a operação de concatenação com a mesma linguagem usamos a notação de potência. Por exemplo, $L_1^2 = L_1 \circ L_1$ e $L_1^3 = L_1 \circ L_1 \circ L_1$.

Reflita

Qual é a melhor definição para L_1^0?

Definimos $L^0 = \varnothing$ e $L^{n+1} = L^n \circ L$.

Se fizermos a união de todas as potências de L, de L^0 em diante, obtemos o fecho de Kleene da linguagem L, representado por L^*.

Assimile

Definimos $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$

Algumas definições apresentadas neste livro podem ser um pouco diferentes das referências clássicas. Uma referência na qual a notação e as definições estão muito próximas às adotadas neste livro é Garcia (2017).

Para o alfabeto $L = \{0,1\}$ temos:

$$L^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

Usando a concatenação de conjuntos, a união e o fecho de Kleene, podemos especificar algumas linguagens simples.

Exemplificando

Considerando o alfabeto $\Sigma = \{0,1\}$ podemos especificar algumas linguagens formadas por cadeias que representam números na base binária de numeração.

$L =$ { números na base 2 que são múltiplos de 4 (terminam com 00) } = $\{0,1\}^{\circ}\{00\}$

$L =$ { todos os números na base 2, sem permitir 0's desnecessários à esquerda } = $(\{1\}^{\circ}\{0,1\}^{*}) \cup \{0\}$

$L =$ {todos os números na base 2, que têm um número ímpar de bits ligados (número ímpar de caracteres 1)} = $\{0\}^{*} \circ \{1\} \circ \{0\}^{*} \circ \{1\}^{*} \circ \{0\}^{*} \circ \{1\} \circ \{0\}^{*}$

Uma variação do fecho de Kleene é usar o símbolo +. Definimos $L^{+} = L^{1} \cup L^{2} \cup L^{3} \cup \ldots$

Podemos definir o operador * usando o operador + e vice-versa. Podemos definir L^{*} como a união de L^{0} com L^{+}, enquanto que L^{+}, por sua vez, pode ser definida como $L^{\circ}L^{*}$.

Assimile

$L^{*} = L^{0} \cup L^{+}$

$L^{+} = L^{\circ}L^{*}$

Considere agora a seguinte linguagem: $L = \{00, 000, 00000, 0000000, \ldots\}$. Esse é um exemplo de linguagem infinita onde a listagem de alguns poucos elementos da mesma não deixa claro o que queremos especificar. Neste caso queremos listar as cadeias que possuem apenas o símbolo "0", repetido uma quantidade p de vezes, sendo p um número primo. Como fazer para especificar de forma finita uma linguagem infinita? Uma possibilidade é fazer um programa de computador para listar todos os elementos da linguagem. Será que isto é sempre possível? Vamos nos ocupar desta questão na última unidade do curso.

Pesquise mais

Pesquise na internet a definição da estrutura matemática chamada "monoide" e verifique se um conjunto L de cadeias, junto com a operação de concatenação, juntamente com a palavra (cadeia) vazia, forma um monoide.

Vamos pensar no alfabeto $\Sigma = \{n, +, \times\}$. Vamos entender n como representando um número qualquer, $+$ como a soma, e $\times$ como a multiplicação. Queremos definir uma linguagem L sobre Σ como sendo a linguagem de todas as 'expressões' bem formadas usando-se essas duas operações:

$$L = \{n, n+n, n \times n, n+n+n, n+n \times n, n \times n+n, n \times n \times n, \ldots\}$$

Reflita

Na especificação da linguagem acima, espera-se que o leitor possa adivinhar o que se quer dizer usando-se o símbolo " ".

Você é capaz de pensar em uma forma de se especificar precisamente as cadeias que devem estar em L?

Neste momento, temos uma definição de linguagem que diz que a linguagem é simplesmente um conjunto de sentenças (cadeias). Isso ignora totalmente o aspecto estrutural da linguagem. Na próxima seção vamos introduzir o conceito de gramática e de linguagem gerada por uma dada gramática, com isso, vamos estar especificando a sintaxe destas linguagens e, ao mesmo tempo, apresentando uma forma de especificar finitamente linguagens infinitas.

Sem medo de errar

Em sua apresentação você deve abordar os seguintes assuntos:

- Alfabeto. Dando a motivação, definição e exemplos do conceito.
- Cadeias.

• Concatenação de cadeias.

• Sufixos e prefixos de cadeias (se houver espaço na apresentação).

• Linguagens. Este é o conceito mais importante, deve ser dada a motivação, definição e exemplos do mesmo.

• União de linguagens e concatenação de linguagens.

• Fecho de Kleene (L^*) e a linguagem L^+ .

• Alguns exemplos de especificação de linguagens usando as operações acima.

• Finalmente, você pode falar de especificação finita de linguagens infinitas, bem como de sintaxe, para motivar o principal conceito a ser apresentado na próxima parte da apresentação: o conceito de gramática.

Tenha em mente que seu objetivo é ambicioso, primeiro você quer que uma audiência heterogênea entenda os conceitos apresentados. Além disso, você também quer que a audiência, sobretudo a gerência, entenda como pode utilizar estes conceitos em sua atividade.

Avançando na prática

Numerais romanos

Descrição da situação-problema

O sistema de numeração originário na Roma antiga, aproximadamente no século VIII a.C., é aquele baseado nas letras I, V, X, L, C, D e M. Este sistema foi amplamente utilizado desde a sua criação até o século XIV d.C. Ainda hoje existem usos modernos deste sistema para representar quantidades ou itens em uma ordenação, como a denominação dos séculos no ocidente. Sabe-se que o sistema caiu em desuso e foi substituído pelo sistema posicional com zero (Hindu-Arábico) por este ser uma representação que facilita em muito a aplicação de algoritmos de adição e multiplicação. No sistema romano, a justaposição de símbolos é mais complexa que no sistema decimal hindu-arábico. No sistema decimal, os símbolos 0,1,2,3,4,5,6,7,8 e 9 podem ser justapostos lado a lado em qualquer ordem e livres de quaisquer restrições, a exceção dos zeros à esquerda, que devem ser evitados, por razões de ordem prática. Qualquer sequência de algarismos é um número decimal. Outra propriedade interessante

dos numerais decimais é a sua capacidade de representar qualquer número. O mesmo não acontece com os numerais romanos. Em primeiro lugar não é qualquer sequência de letras I, V, X, L, C, D e M que é um numeral romano válido. Por exemplo, a sequência IIIV não é um numeral romano válido. Além disso, é sabido que os numerais romanos tradicionais não conseguem representar mais que MMMCMXLIX números naturais distintos. Existiram extensões do sistema romano que conseguiam passar disso, mas não chegavam a representação de bilhões. Neste livro vamos nos limitar ao número MMMCMXLIX mesmo.

Nesta seção, você aprendeu que qualquer conjunto de palavras sobre um alfabeto é uma linguagem formal. Assim, tanto a linguagem dos numerais decimais quanto a dos numerais romanos são linguagens formais. Uma é uma linguagem infinita e outra é uma linguagem finita. Se não levarmos em consideração os "zeros à esquerda" que devem ser evitados na notação decimal, podemos dizer que os numerais decimais são a linguagem definida pelo conjunto $\{0,1,2,3,4,5,6,7,8,9,\}^+$. Ou seja, usamos um dos operadores aprendidos, o chamado fecho de Kleene, para definir o conjunto de todos os numerais decimais de uma forma compacta. Lembre-se que a linguagem em questão é infinita.

Você consegue representar a restrição de não haver zeros à esquerda através de conjuntos de símbolos e as operações entre linguagens formais apresentadas nesta seção?

Pense em como fazer o mesmo para a linguagem dos numerais romanos, pois apesar da linguagem ser finita, ela tem uma quantidade grande de palavras, 3999 para sermos exatos. Obviamente uma representação mais compacta da linguagem dos numerais romanos é também muito bem-vinda. Como fazer isso? Relembre da notação de numerais romanos, através de uma busca na internet. Depois de ter certeza que sabe como funciona a numeração romana padrão, mostre como se define o conjunto dos numerais romanos, através de operações $\cap$, $\cup$, $^\circ$, * e +, sobre conjunto de símbolos I, V, X, L, C, D e M.

Resolução da situação-problema

Para se restringir os numerais decimais com a impossibilidade de termos zeros à esquerda, basta impor que os símbolos mais à

esquerda de qualquer palavra da linguagem sejam 1,2,3,4,5,6,7,8 e 9. Desta forma

$$DECIMAIS = (\{1,2,3,4,5,6,7,8,9\}^{\circ}\{0,1,2,3,4,5,6,7,8,9,\}^{+}) \cup (\{1,2,3,4,5,6,7,8,9\})$$

E temos então incluído a restrição de não haver zeros à esquerda. Note que o conjunto acima também pode ser especificado como

$$DECIMAIS = \{1,2,3,4,5,6,7,8,9\}^{\circ}\{0,1,2,3,4,5,6,7,8,9,\}^{*}$$

Como relação aos numerais romanos, vamos estruturar nosso conhecimento: (1) as palavras I, II e III são as únicas que podem ser escritas só com I; (2) imediatamente à esquerda de um V só pode ocorrer um I; (3) À direita de um V podem ocorrer até no máximo 3 Is; (4) À direita de um X podem ocorrer no máximo 3 Is e à sua esquerda somente um I; (5) a regra 4 também vale em relação a L, C, D e M. Em resumo, todo numeral romano tem um núcleo de maior valor, por exemplo o núcleo de MMXVII é MM, o núcleo de CDXXIV é CD. Antes do núcleo pode ocorrer um, e somente um, símbolo de menor valor e depois do núcleo de maior valor pode aparecer um núcleo de valor menor.

Observe que a explicação em linguagem natural, mesmo organizada, fica complicada. Vamos fazer usando as operações entre conjuntos. Para facilitar a leitura vamos denotar cada novo conjunto especificado.

$$NI = \{I, II, III, IV, V, VI, VII, VIII, IX\}$$
$$AX = \{X, XX, XXX, XL, L, LX, LXX, LXXX, XC\}$$
$$AC = \{C, CC, CCC, CD, D, DC, DCC, DCCC, CM$$
$$NX = AX \cup (AX^{\circ}NI)$$
$$NC = AC \cup (AC^{\circ}NX)$$
$$AM = \{M, MM, MMM\}$$
$$NM = AM \cup (AM^{\circ}NC)$$

O conjunto NM é a linguagem das cadeias que são numerais romanos até a numeração de 3999.

Faça valer a pena

1. Considere a linguagem $L = \{aab, a\}$ sobre o alfabeto $\Sigma = \{a, b\}$.
Marque a alternativa correta:
a) $L^0 = \varnothing$
b) $L^2 = \{aabaab, aa\}$
c) $L^3 = \{aaa, aaaab, aaabaab, aabaabaab, aabaa, aabaaba\}$
d) $L^4 \subset L^5$
e) $L^4 \subset L^*$

2. Suponha que L_1 e L_2 sejam linguagens sobre o alfabeto $\Sigma = \{a, b\}$.
Assinale a alternativa verdadeira:
a) Se $L_1 \circ L_2 = \varnothing$, então $L_1 = \varnothing$.
b) Se $L_1 \circ L_1 = \varnothing$, então $L_1 = \varnothing$.
c) Se $L_1 = \{\epsilon\}$, então $L_1 \circ L_2 = \{\epsilon\}$.
d) Se $L_1 = \{\epsilon\}$, então $L_1 \circ L_2 = \varnothing$.
e) Se $L_1 \subseteq L_2$, então $L_1^* = L_2^*$.

3. Considere a cadeia $\omega = ababa$.
Assinale a cadeia que pode ser formada concatenando-se dois prefixos de ω:
a) $abba$
b) $abaabb$
c) bb
d) ba
e) a

Seção 1.3

Linguagens formais e gramáticas

Diálogo aberto

Nesta seção, você verá os conceitos de gramática, derivação e linguagem gerada por uma gramática. Estes conceitos completam os conceitos vistos na seção anterior, permitindo a formalização de sintaxe de uma linguagem. Com estes conceitos será formulado o problema da análise sintática: dada uma cadeia e uma gramática, a cadeia está na linguagem gerada por esta gramática? Trata-se de um problema difícil, vamos passar a maior parte do restante do livro investigando a solução deste problema para diversos tipos de gramáticas, com um nível de complexidade crescente. Esses diferentes tipos de gramáticas serão vistos ainda nesta seção, quando abordarmos a Hierarquia de Chomsky.

Lembre-se que você está preparando uma apresentação sobre Linguagens Formais e Autômatos para o processo de seleção para uma instituição de pesquisa meteorológica.

Você já preparou as duas primeiras partes de sua apresentação e está satisfeito com o resultado até aqui. Para concluir sua apresentação você deve introduzir os conceitos de Gramática e explicar seu funcionamento, isto é, explicar o que é a linguagem gerada por uma gramática. Você pode mencionar que a gramática formaliza a sintaxe da linguagem que gera. Finalmente, você pode concluir apresentando os diversos tipos de gramáticas e a Hierarquia de Chomsky.

Não pode faltar

O conceito de linguagem apresentado na seção anterior deixa a desejar, por não abordar o aspecto sintático de uma linguagem. Além disso, observamos que as linguagens mais interessantes para o nosso estudo são infinitas, e temos alguma dificuldade para especificar algumas destas linguagens de forma precisa. Nesta seção, vamos apresentar o conceito de gramática. Gramáticas compõem uma forma de especificar linguagens, podendo especificar linguagens

infinitas. Além disso, gramáticas mostram como cada cadeia de uma linguagem é gerada através de regras, o que modela o aspecto sintático da linguagem.

Como um primeiro exemplo de gramática, suponha que estamos interessados em especificar a linguagem L_D dos números, numerais para sermos mais precisos, não negativos na base decimal. Estamos interessados não só nos inteiros, como também nos números com ponto decimal. Assim como nas principais linguagens de programação, usaremos ponto decimal (e não vírgula) para separar a parte decimal do número. Exemplos de números nesta linguagem são $L_D = \{0,1,2,\ldots 9,10,11,\ldots,99,100,\ldots,0.1,0.2,\ldots\}$. Neste exemplo fica clara a dificuldade de especificação de uma linguagem infinita. O alfabeto sobre o qual L_D está definida é $\Sigma = \{0,1,2,3,4,5,6,7,8,9,.\}$, ou seja, os 10 dígitos decimais mais o ponto. Uma forma de especificarmos isso é através das regras a seguir:

- $N \rightarrow L$ (um número N pode ser uma lista de dígitos L)
- $N \rightarrow L.L$ (N pode ser uma lista de dígitos L seguida de outra lista L)
- $L \rightarrow D$ (uma lista de dígitos L pode ser um dígito D)
- $L \rightarrow LD$ (uma lista de dígitos L pode ser outra lista L seguida de um dígito D)
- $D \rightarrow 0$ (uma dígito D pode ser o dígito 0)
- $D \rightarrow 1$ (uma dígito D pode ser o dígito 1)
- $D \rightarrow 2$ (uma dígito D pode ser o dígito 2)
- $D \rightarrow 3$ (uma dígito D pode ser o dígito 3)
- $D \rightarrow 4$ (uma dígito D pode ser o dígito 4)
- $D \rightarrow 5$ (uma dígito D pode ser o dígito 5)
- $D \rightarrow 6$ (uma dígito D pode ser o dígito 6)
- $D \rightarrow 7$ (uma dígito D pode ser o dígito 6)
- $D \rightarrow 8$ (uma dígito D pode ser o dígito 8)
- $D \rightarrow 9$ (uma dígito D pode ser o dígito 9)

Neste caso, podemos concluir que 1.23 é uma cadeia especificada por esta gramática, porque: N é um número (numeral); devido à regra $N \rightarrow L.L$, $L.L$ é um número (numeral decimal); mas devido à regra $L \rightarrow D$, $D.L$ é um numeral decimal. Seguimos a sequência,

sem indicar as regras: Se $D.L$ é um numeral, $1.L$ é um numeral, portanto $1.LD$ é um numeral, assim $1.DD$ é um numeral, portanto $1.2D$ é um numeral, logo 1.23 também é um numeral.

Neste caso dizemos que N gera 1.23. O processo pode ser representado pelo símbolo $\Rightarrow$, da seguinte forma:

$$N \Rightarrow L.L \Rightarrow D.L \Rightarrow 1.L \Rightarrow 1.LD \Rightarrow 1.DD \Rightarrow 1.2D \Rightarrow 1.23$$

A essa especificação damos o nome de gramática. Observando a especificação com mais detalhes, podemos observar que ela possui os seguintes elementos:

• Um alfabeto sobre o qual a linguagem é definida, chamamos os elementos deste alfabeto de símbolos terminais: $T = \{0,1,2,3,4,5,6,7,8,9,.\}$

• Um conjunto de símbolos auxiliares, aos quais chamamos de símbolos não terminais ou variáveis: $V = \{N, L, D\}$;

• Um conjunto de regras que indicam como os símbolos são substituídos para gerar uma cadeia da linguagem: $P = \{N \to L, N \to L.L,\ L \to D,\ L \to LD,\ D \to 0,\ D \to 1,\ D \to 2,\ D \to 3, D \to 4,\ D \to 5,\ D \to 6,\ D \to 7,\ D \to 8,\ D \to 9\}$. Cada regra da forma $L \to \omega$ indica que podemos substituir qualquer ocorrência de L por ω na palavra (cadeia) que está sendo gerada.

• Uma das variáveis de V designada por símbolo inicial, símbolo do qual se iniciam as derivações, neste caso: N.

Reflita

Nesta aula apresentamos o conceito de par ordenado, por meio do qual teremos a igualdade $(x_1, x_2) = (y_1, y_2)$ se, e somente se, $x_1 = y_1$ e $x_2 = y_2$. Como você generalizaria este conceito para triplas ou quádruplas ordenadas?

De uma forma mais geral do que triplas ou quádruplas ordenadas, chamamos de tupla a $(x_1, x_2, \ldots, x_n)$ e dizemos que $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ se, e somente se, para todo o $i \in \{1, 2, \ldots,\}$, $x_i = y_i$. Com este conceito de tuplas, podemos passar a uma definição mais formal de gramática. (MENEZES, 2000)

Assimile

Uma gramática G é uma tupla (V,T,P,S) onde:

• V é um conjunto finito não vazio de variáveis.

• T é um conjunto finito não vazio de símbolos terminais.

• P é um conjunto finito de regras de produção da forma: $\alpha \to \beta$ onde $\alpha, \beta \in (V \cup T)^*$, onde α tem ao menos uma variável.

• $S \in V$ é o símbolo inicial.

Na Seção 1.1 estudamos o conceito de relação. O funcionamento da gramática se dá através da relação $\Rightarrow_G$ ou simplesmente $\Rightarrow$. O domínio e o contradomínio desta relação é o conjunto $(V \cup T)^*$. Se $G = (V,T,P,S)$, dizemos que $\alpha \Rightarrow_G \beta$ quando $\alpha = \delta_1\alpha_1\delta_2$ e $\beta = \delta_1\beta_1\delta_2$ e a regra $\alpha_1 \to \beta_1$ está em P. Ou seja, se uma regra da gramática descreve como podemos trocar α_1, um pedaço de α igual ao lado esquerdo de uma regra, pelo lado direito da mesma regra, obtendo β. Neste caso dizemos que α deriva β. Observe que α_1 pode ocorrer em mais de uma posição em α e que a aplicação da regra $\alpha_1 \to \beta_1$ pode ser aplicada em qualquer destas posições. Devemos aplicar as regras até que não haja mais não variáveis.

Exemplificando

Vamos considerar uma gramática análoga à gramática vista para números decimais. Entretanto, esta gramática só possui os dígitos 0 e 1, gerando os números de ponto flutuante na base binária. $G = (V,T,P,N)$, onde:

• $T = \{0,1\}$

• $V = \{N,L,D\}$;

• $P = \{N \to L, \quad N \to L.L, L \to D, L \to LD, D \to 0, D \to 1\}$.

Para esta gramática temos os seguintes exemplos de $\Rightarrow_G$:

- $L.L \Rightarrow_G L.LD$
- $L.L \Rightarrow_G LD.L$

Quando a gramática G está subentendida no contexto, nós a suprimimos da notação, escrevendo simplesmente:

- $L.LD \Rightarrow L.L0$

É muito útil escrever mais de um passo da relação $\Rightarrow_G$ com um único símbolo $\Rightarrow_G^*$, por exemplo, se $N \Rightarrow_G L.L \Rightarrow_G D.L \Rightarrow_G 1.L$, podemos escrever $N \Rightarrow_G^* 1.L$. O * indica que $\Rightarrow_G^*$ também é reflexiva, ou seja, $\alpha \Rightarrow^* \alpha$.

Assimile

Dada uma gramática G é uma tupla (V,T,P,S), dizemos que a relação $\Rightarrow_G^* \subseteq (V \cup T)^* \times (V \cup T)^*$ é a menor relação tal que:

- Se $\alpha \Rightarrow_G \beta$ então $\alpha \Rightarrow_G^* \beta$.
- Para todo $\omega \in (V \cup T)^*$, $\omega \Rightarrow_G^* \omega$.
- Para todos $\alpha, \beta, \gamma \in (V \cup T)^*$, se $\alpha \Rightarrow_G^* \beta$ e $\beta \Rightarrow_G^* \gamma$ então $\alpha \Rightarrow_G^* \gamma$.

Esta notação é lida como "deriva em zero ou mais passos". Por exemplo, $N \Rightarrow_G^* 1.L$ é pronunciado "N deriva em zero ou mais passos $1.L$".

Pesquise mais

A relação $\Rightarrow_G^*$ pode ser definida de forma sucinta como $\Rightarrow_G^*$ é o fecho reflexivo-transitivo de $\Rightarrow_G$. Pesquise na internet sobre o que é um fecho transitivo-reflexivo no site disponível em:<http://bit.ly/2r6piTj>. Acesso em: 6 jun. 2017.

Considere a gramática $G = (V, T, P, S)$ onde:

- $V = \{S, B\}$
- $T = \{a\}$
- $P = \{S \rightarrow aB, B \rightarrow \epsilon\}$

Esta gramática gera uma única sentença, a saber, a sentença "a". A sentença pode ser obtida pela derivação: $S \Rightarrow aB \Rightarrow a$. Observe que no segundo passo foi aplicada a regra $B \rightarrow \epsilon$. Esta regra indica que B pode ser substituído pela cadeia vazia. Essa gramática é bastante simples e gostaríamos de descrevê-la de forma mais simples do que a apresentada acima. Podemos fazer isso adotando algumas convenções. Convencionamos que as variáveis serão sempre letras maiúsculas e que os demais símbolos serão terminais. Além disso, o símbolo inicial é o lado esquerdo da primeira regra apresentada. De posse desta convenção, podemos apresentar a mesma gramática de uma forma muito mais concisa. Poderíamos definir a gramática anterior em duas linhas, simplesmente como:

- $S \rightarrow aB,$
- $B \rightarrow \epsilon$

Podemos agora definir a linguagem gerada pela gramática $G = (V, T, P, S)$ como o conjunto de todas as sentenças $\alpha \in T^*$ tais que $S \Rightarrow_G^* \alpha$. Escrevemos esta linguagem como $L(G)$.

Assimile

Dada uma gramática G é uma tupla (V, T, P, S), dizemos que a linguagem gerada por G é:

- $L(G) = \{\omega \in T^* \mid S \Rightarrow_G^* \omega\}$.

Uma vez que definimos o que é gramática e o que é a linguagem gerada por esta gramática, podemos nos perguntar: dada uma gramática G e a uma sentença ω, como determinar se $\omega \in L(G)$? Este problema é o problema da análise sintática. Podemos dizer que todo o conteúdo do livro daqui em diante está relacionado com este problema. Vamos resolvê-lo por partes, para diferentes "tipos" de gramática, de acordo com a sua complexidade.

Você deve ter percebido que as regras de uma gramática podem ter diferentes formatos. Por exemplo, considere o formato onde do lado esquerdo só há uma variável, ou seja, da forma $N \to \gamma$. Tais regras indicam que qualquer ocorrência de N na palavra que está sendo gerada pode ser substituída por γ. Ou seja, o N pode ser substituído por γ em qualquer posição que ele ocorra. Regras com este formato são chamadas de regras livres de contexto, isto é, a regra indica que se pode substituir N por γ independentemente do contexto em que N aparece.

A regra $A \to aA$ pode ser aplicada na forma sentencial (cadeia de símbolos terminais e não terminais) $ABACAB$ em cada uma das 3 posições do símbolo não terminal A. Para percebermos a diferença das regras livres de contexto, vamos contrastar com a regra $AB \to aAB$. Esta regra, em função do B no lado esquerdo, só pode ser aplicada à primeira e à terceira ocorrência de A em $ABACAB$, pois a segunda ocorrência não tem um B justaposto à direita. De fato, a aplicação se dá sobre o AB e só há duas ocorrências de AB na palavra $ABACAB$. Portanto, ao aplicarmos esta regra, que não é livre de contexto, à primeira ocorrência de AB na palavra $ABACAB$ temos $aABACAB$ e caso apliquemos à segunda ocorrência de AB teremos $ABACaAB$. Note que podemos continuar a aplicar esta regra às palavras resultantes indefinidamente. A este tipo de regra, que possui mais de um símbolo à esquerda da seta, denominamos regras sensíveis ao contexto, ou dependentes de contexto, sempre que o lado direito tiver mais ou tantos símbolos que o lado esquerdo. Caso contrário, ou seja, se o lado direito possuir menos símbolos que o esquerdo, temos o tipo mais geral de regra. Ou seja, uma regra da forma $\alpha \to \beta$ é sensível ao contexto se, e somente se, $|\alpha| \leq |\beta|$. Por hora, basta entendermos que isto contempla regras da forma $\alpha_1 A \alpha_2 \to \alpha_1 \gamma \alpha_2$, com γ não sendo a palavra vazia. Veja, esta última regra basicamente diz que A é γ no contexto de α_1 à esquerda e α_2 à direita.

Existem casos particulares de regras livres de contexto que são importantes por serem mais simples e ainda permitem a geração de linguagens infinitas. Regras nas formas $A \to aB$, $A \to \epsilon$ ou $A \to b$ são chamadas de regulares. Entende-se que A e B representam duas variáveis quaisquer (podendo ser a mesma), enquanto a e b representam dois terminais quaisquer.

Considere a gramática:

- $A \rightarrow aA$
- $A \rightarrow b$

Podemos ver que $A \Rightarrow aA$, portanto $A \Rightarrow^* aaA$, $A \Rightarrow^* aaaA$ etc. Usando a notação a^k para indicar uma sequência de k a´s, teremos que $A \Rightarrow^* a^k A$, com $k > 0$ e, portanto, a linguagem gerada por A é formada pelas palavras $a^k b$, com $0 < k$, pois uma vez aplicada a regra $A \rightarrow b$ não teremos mais variáveis na palavra. Existe, entretanto, um fato bastante relevante nas derivações de $a^k b$: A cadeia sendo gerada só possui uma ocorrência de variável, e esta está sempre no final da cadeia. Dada uma gramática G = (V, T, P, S), segue um resumo dos possíveis tipos de regras com as respectivas denominações:

• Regra regular: $A \rightarrow b$, $A \rightarrow \epsilon$ ou $A \rightarrow aB$, com $A, B \in V$ e $a, b \in T$;

• Regra livre de contexto: $A \rightarrow \gamma$, com $A \in V$ e $\gamma \in (V \cup T)^*$;

• Regra sensível ao contexto: $\alpha \rightarrow \beta$, com $\alpha, \beta \in (V \cup T)^*$ e $|\alpha| \leq |\beta|$; ou $S \rightarrow \epsilon$, se S não aparece do lado direito de nenhuma regra;

• Regra geral: Não possui restrição além daquela na definição de gramática, que obriga a existir ao menos uma variável no lado esquerdo da regra.

Das denominações acima podemos constatar que toda regra regular é livre de contexto, toda livre de contexto cujo lado direito é não vazio é sensível ao contexto e toda sensível ao contexto é geral (GARCIA, 2017). Por outro lado, existem regras gerais que não são sensíveis ao contexto, regras sensíveis ao contexto que não são livres de contexto e finalmente regras livres de contexto que não são regulares. Temos uma hierarquia de regras. Vejamos como esta hierarquia de regras dá origem a uma hierarquia de linguagens.

Uma gramática é dita ser de certo tipo (regular, livre de contexto, sensível ao contexto, geral), se, e somente se, todas as suas regras são daquele tipo. Uma linguagem é de certo tipo Ti, se, e somente se, existe uma gramática do tipo Ti que gera a linguagem. Dizer então que uma linguagem não é de certo tipo é equivalente a dizer que não existe gramática daquele tipo capaz de gerar a linguagem.

Exemplificando

Seja a gramática G_1:

- $A \rightarrow aA$
- $A \rightarrow b$

G_1 só possui regras regulares, portanto é uma gramática regular. A linguagem $L = \{a^k b, k > 0\}$ é gerada pela gramática regular G_1, portanto L é regular. Seja a gramática G_2:

- $A \rightarrow aaA$
- $A \rightarrow aA$
- $A \rightarrow b$

G_2 possui uma regra que não é regular, portanto a gramática G_2 não é regular. Entretanto, a linguagem $L(G_2)$ é regular, pois $L(G_2) = L(G_1)$, e basta que exista uma gramática regular que a gere para que a linguagem seja regular.

A hierarquia de Chomsky é justamente a afirmação que as linguagens formam uma hierarquia a partir dos tipos das gramáticas que são capazes de gerá-las. Cada tipo gramatical também nos indicará o mecanismo/autômato capaz de resolver o problema da análise sintática para aquele tipo de gramática. Na sua forma menos detalhada, a hierarquia de Chomsky é mostrada de acordo com a Tabela 1.1.

Tabela 1.1 | Tipos de Gramáticas

Tipos de Gramáticas	Regras	Exemplos de Linguagens geradas por estas gramáticas
Regulares	$A \to b$, $A \to \epsilon$ ou $A \to aB$ $A, B \in V^*$, $a, b \in T^*$	$a^n b^m, n, m > 0$
Livres de Contexto	$A \to \gamma$ $\gamma \in (V \cup T)^*$	$a^n b^n, n > 0$
Sensíveis ao Contexto	$\alpha \to \beta$, com $\alpha, \beta \in (V \cup T)^*$ e $\lvert \alpha \rvert \leq \lvert \beta \rvert$; ou $S \to \epsilon$, se S não aparece do lado direito de nenhuma regra;	$a^n b^n c^n, n > 0$
Irrestrita ou geral	$\alpha \to \beta$, com $\alpha, \beta \in (V \cup T)^*$ α tem pelo menos símbolo de V.	----------

A Tabela 1.1 apresenta quatro tipos de gramáticas, o objetivo é sinalizar o que será visto nas próximas unidades, nelas serão vistos cada um destes tipos com maiores detalhes.

Em particular, na próxima unidade serão abordadas Linguagens Regulares, Autômatos Finitos e Expressões Regulares, assuntos mais concretos e que possuem diversas aplicações. Continue com sua dedicação à leitura e aos exercícios.

Sem medo de errar

Para resolver à situação-problema proposta, você deve abordar os seguintes assuntos na última parte de sua apresentação:

- Gramáticas como solução de descrição de linguagens, e também como mecanismo de especificação de sintaxe.
- Relação de derivação em um passo.
- Relação de derivação em zero ou mais passos.
- Linguagem gerada por uma gramática.
- Neste ponto você deve dar exemplos de gramáticas e das respectivas linguagens geradas.
- Hierarquia de Chomsky. Gramáticas regulares, livres de contexto, sensíveis ao contexto e gramáticas sem restrição.
- Finalmente, você deve abordar o problema da análise sintática.

Avançando na prática

Dissertação sobre gramáticas

Descrição da situação-problema

Você quer ser aceito em um curso de mestrado em Ciência da Computação para trabalhar com o tema de linguagens formais e autômatos. Você chegou à última fase do processo seletivo e para ser aceito deve escrever uma dissertação sobre linguagens formais e gramáticas geradoras. Nesta dissertação você deve explicar o fato de que existem linguagens que não são geradas por gramáticas e exibir uma destas linguagens. Depois de escrever essa dissertação você deve defendê-la oralmente para o professor responsável pelo processo seletivo. Você pede ajuda ao seu professor de linguagens formais, que dá a explicação que segue:

Você aprendeu que gramáticas são formalismos/ferramentas capazes de gerar, ou descrever, linguagens formais. Observamos também que gramáticas são objetos formais finitos. De um ponto de vista bem básico, uma gramática é uma palavra em um alfabeto que inclui o símbolo $\rightarrow$ e o alfabeto da própria linguagem que ela gera. Para simplificar nossa discussão, vamos considerar somente gramáticas que geram palavras sobre o alfabeto $\{0,1\}$.

Podemos estabelecer uma notação para as variáveis das gramáticas e com isso fixamos o alfabeto para escrever gramáticas em um conjunto fixo de símbolos. Por exemplos se temos n variáveis na gramática, usamos as variáveis com nomes $<V1>$, $<V11>$, $<V111>$, ..., $<V1^n>$. Vamos usar a ausência de símbolos para o ϵ. Observe o uso dos $<>$ para delimitar a notação de variáveis e não misturar o símbolo 1 do alfabeto da linguagem descrita/gerada pela gramática com a notação para variáveis. Só para exemplificar, uma gramática como:

$$S \rightarrow 1S0, S \rightarrow \epsilon$$

é representada pela cadeia $<V1>;<V1>\rightarrow 1<V1>0;<V1>\rightarrow$. O primeiro $<V1>$ indica que ele é o símbolo inicial da gramática. Em seguida vêm as regras, com $V1$ no lugar de S.

Você se convenceu que qualquer gramática que gera uma linguagem sobre $\{0,1\}$ pode ser escrita nesta forma? Demonstre que sim descrevendo a palavra associada a uma gramática tal como $S; S \rightarrow ASBC;\ S \rightarrow; AB \rightarrow BA;\ BC \rightarrow CB;\ AC \rightarrow CA;$ $A \rightarrow 10;\ B \rightarrow 00;\ C \rightarrow 11$, com variáveis A, B e C, no formato recém-discutido.

Acreditamos que você já se convenceu que podemos codificar (esse é o termo) cada gramática que gera uma linguagem sobre $\{0,1\}$ através de uma palavra sobre o alfabeto $\{<,>,V,1,0,\rightarrow,;\}$. Baseado no que aprendemos na Seção 1, existem tantas gramáticas quantos números naturais. Cada palavra sobre o alfabeto destas gramáticas pode ser vista como um numeral natural na base 7, a quantidade de símbolos no alfabeto. Por outro lado, para cada número natural k, temos a gramática $<V1>;<V1>\rightarrow 1^k$, podemos contar números naturais com gramáticas e gramáticas com números naturais. Existe então uma quantidade infinita

enumerável de gramáticas que geram linguagens sobre $\{0,1\}$.

Sua dissertação pode mostrar que existem linguagens não geradas por uma gramática através de um argumento meramente quantitativo: qual a cardinalidade do conjunto de linguagens formais sobre $\{0,1\}$? Ou seja, qual a cardinalidade do conjunto $\{L / L \subseteq \{0,1\}^*\}$. Lembre-se do teorema de Cantor visto na Seção 1.

Finalmente, para exibir uma linguagem particular que não é gerada por uma gramática, lembre-se do argumento utilizado na demonstração do teorema de Cantor, ou seja, a diagonalização. A linguagem formal $L_d = \{G / G$ não gera $G\}$ tem gramática geradora? Isto é, se uma gramática pode ser codificada por uma cadeia que é gerada por ela, então ela está neste conjunto, caso contrário não está.

Resolução da situação-problema

1- Observe que o teorema de Cantor diz que para qualquer conjunto S, a cardinalidade de $P(S)$ é estritamente maior que S. Portanto $\{L / L \subseteq \{0,1\}^*\}$ tem mais linguagens que gramáticas codificadas sobre $\{<,>,V,1,0,\rightarrow,;\}$. Afinal, a cardinalidade de $\{0,1\}^*$ é a do conjunto dos numerais binários.

2- Portanto existem linguagens formais que não possuem gramáticas geradoras.

3- No paradoxo do barbeiro, que é a base para o argumento da diagonalização, mudamos homens para gramáticas e a oração H_1 barbeia H_2 para H_1 gera H_2. O que temos então no conjunto $\{G / G$ não gera $G\}$ é o conjunto de todas as gramáticas que não geram seu código, e somente estas. Como na prova de Cantor, este conjunto de código de gramáticas não pode ter uma gramática geradora. Suponha que exista uma gramática Y que gere $L_d =$ {G / G não gera G}. Se Y gera seu código então significa que Y não está em L_d, portanto Y não gera seu código. Se Y não gera seu código significa que Y está em L_d, portanto Y gera seu código. Isto é uma contradição, logo não pode existir a gramática Y que gere L_d. Acompanhe o argumento passo a passo e você o entenderá.

Veja que a gramática que pedimos para codificar:

$S; S \rightarrow ASBC; \quad S \rightarrow; AB \rightarrow BA; \quad BC \rightarrow CB; \quad AC \rightarrow CA;$
$A \rightarrow 10; B \rightarrow 00; C \rightarrow 11$,

Pode ser codificada em $\{<,>V,1,0,\rightarrow,;\}$ como:

$< V1 >; < V1 > \rightarrow < V11 >< V1 >< V111 >< V1111 >;$
$< V1 > \rightarrow ; < V11 >< V111 > \rightarrow < V111 >< V11 >;$
$< V111 >< V1111 > \rightarrow < V1111 >< V111 >;$
$< V11 >< V1111 > \rightarrow < V1111 >< V11 >;$
$< V11 > \rightarrow 10; < V111 > \rightarrow 00; < V1111 > \rightarrow 11$

Faça valer a pena

1. Considere a gramática $G = (V, T, P, S)$ onde:
- $V = \{S, A, B\}$;
- $T = \{0, 1\}$;
- $P = \{S \rightarrow 0S,$
- $S \rightarrow 1A,\ S \rightarrow \epsilon, A \rightarrow 1S,\ A \rightarrow 0B,\ B \rightarrow 0A,\ B \rightarrow 1B\}$

Marque a cadeia gerada pela gramática G:
a) 1110111
b) 1111001
c) 1111011
d) 1111100
e) 1111101

2. Considere a gramática $G = (V, T, P, S)$ onde:
- $V = \{S, A, B\}$;
- $T = \{0, 1\}$;
- $P = \{S \rightarrow 0S,$
- $S \rightarrow 1A,\ S \rightarrow \epsilon, A \rightarrow 1S,\ A \rightarrow 0B,\ B \rightarrow 0A,\ B \rightarrow 1B\}$

Marque a alternativa correta:
a) $S \Rightarrow_G S$
b) $S \Rightarrow_G 1111011$
c) $SA \Rightarrow_G 01$
d) $SA \Rightarrow_G A$
e) $SA \Rightarrow_G B$

3. Sabemos que a linguagem gerada por uma gramática, $L(G)$, é o conjunto de todas as cadeias geradas por G.
Assinale a alternativa que contém a gramática que gera uma linguagem finita:
a) $S \rightarrow aS,$
$S \rightarrow \epsilon$

b) $S \rightarrow SS$,
$S \rightarrow \epsilon$

c) $S \rightarrow aS$,
$S \rightarrow a$

d) $S \rightarrow aA$,
$A \rightarrow a$

e) $S \rightarrow aSa$,
$S \rightarrow \epsilon$

Referências

CANTOR, G. Ueber eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. **Journal für die Reine und Angewandte Mathematik**, v. 77, p. 258-262, 1874. doi:10.1515/crll.1874.77.258.

GARCIA, A. **Linguagens Regulares e Livres de Contexto**. 1. ed. Rio de Janeiro: Edição do Autor (eBook Kindle), 2017.

HOPCROFT, J.; MOTWANI, R.; ULLMAN, J. **Introdução à Teoria de Autômatos, Linguagens e Computação**. 1. ed. Boston: Elsevier, 2002.

MENEZES, P. B. **Linguagens Formais e Autômatos**. Porto Alegre: Sagra Luzzatto, 2000.

_____. **Matemática Discreta para Computação e Informática**. 4. ed. Porto Alegre: Bookman, 2013.

PREPARATA, F. P.; YEH, R. T. **Introduction to Discrete Structures**. 1. ed. [S.l.]: Addison-Wesley, 1973.

Linguagens, gramáticas e expressões

Convite ao estudo

Sua apresentação sobre linguagens formais e autômatos foi muito boa. Você conseguiu o emprego na instituição que faz pesquisa meteorológica. A empresa possui uma massa de dados de décadas sobre as condições climáticas, relativa a centenas de localizações geográficas. Seu chefe pede que você localize certos padrões na massa de dados.

Na unidade anterior, você foi apresentado aos conceitos de gramática e de linguagem gerada por uma gramática. Em geral, dada uma gramática e uma cadeia, é difícil saber se a gramática gera ou não esta cadeia em particular. Por isso, na última seção da unidade anterior foi apresentada a Hierarquia de Chomsky, na qual diversos tipos de gramática são apresentados em ordem crescente de complexidade. Nesta unidade, vamos nos aprofundar em tudo que se refere a linguagens regulares. Na primeira seção falaremos sobre gramáticas regulares, linguagens regulares e alguns resultados de fechamento. Na segunda seção apresentaremos autômatos finitos, que são autômatos capazes de reconhecer linguagens regulares, e, na terceira e última seção desta unidade, apresentaremos expressões regulares, que são um formalismo com o mesmo poder de expressividade que as gramáticas regulares.

Todos os conhecimentos são de aplicação imediata em sua atuação profissional, por este motivo, é importante manter a dedicação ao aprendizado.

Seção 2.1

Linguagens regulares

Diálogo aberto

Como dito anteriormente, sua apresentação de linguagens formais e autômatos foi um sucesso e você conseguiu o emprego na instituição na qual havia se submetido ao processo de seleção.

Nesta seção, você estudará com mais detalhes as linguagens regulares e as gramáticas regulares, por tratar-se de um conhecimento mais aplicado. Em particular, existem diversas técnicas para o reconhecimento de padrões descritos por linguagens regulares e estas técnicas serão vistas no restante desta unidade.

A empresa que o contratou possui uma massa de dados de décadas sobre as condições climáticas, relativa a centenas de localizações geográficas. Seu chefe pede que você localize certos padrões na massa de dados. Como desenvolvedor pleno de sua equipe, você ficou responsável por levantar os tipos de padrões buscados para avaliar a melhor solução para programar a localização destes. Você levantou que, em um primeiro momento, a única tarefa a ser executada será encontrar números negativos (que representam temperaturas negativas) em arquivos do tipo '.csv' (commaseparatedvalues). Por exemplo "-12.33". Em função disso, qual solução você propõe?

Para programar sua solução serão particularmente úteis os conteúdos das Seções 2 (autômatos finitos) e 3 (expressões regulares) desta unidade. Entretanto, neste momento, você irá apenas propor a solução e especificá-la em seu relatório criando uma gramática regular. Esta é uma excelente oportunidade para colocar em prática os conhecimentos adquiridos nesta seção.

Não pode faltar

Uma das aplicações mais básicas e importantes relacionadas a uma linguagem formal é a possibilidade de reconhecer-se mecânica ou automaticamente as palavras que pertencem a ela. Este é o papel do reconhecedor ou analisador sintático. Um reconhecedor para uma linguagem formal $L \subseteq \Sigma^*$ é um procedimento que ao ler

qualquer palavra $\omega \in \Sigma^*$ indica se $\omega \in L$ ou se $\omega \notin L$. Na realidade, o analisador sintático faz um pouco além disso, mas isto é assunto de outra unidade.

Uma das primeiras tarefas no reconhecimento de uma linguagem, seja ela natural ou não, trata da identificação das palavras ou unidades básicas que fazem parte dela. Já sabemos que uma palavra é uma cadeia de caracteres ou símbolos. A exigência básica é que estes sejam reconhecíveis. Isto é, para reconhecer a linguagem deve haver um mecanismo básico ou primitivo, capaz de ler cada símbolo individualmente e reconhecer se eles são diferentes de outros símbolos do mesmo alfabeto ou qualquer outro alfabeto que o inclua. Por exemplo, no alfabeto binário $\{0,1\}$, este mecanismo básico deve ser capaz de distinguir o 0 do 1, além de distinguir 0 ou 1 de a, por exemplo, um símbolo que não pertence a $\{0,1\}$.

Todo alfabeto deve dispor deste mecanismo de identificação de seus elementos. Dado um símbolo s e um alfabeto Σ, saber se $s \in \Sigma$ e saber se $s \neq s_2 \in \Sigma$ deve ser um procedimento automático. Como se faz para reconhecer uma palavra? A resposta imediata é verificando-a símbolo a símbolo. Por exemplo, sabemos que $12324A563$ não é um numeral decimal, dado que a palavra que o representa não é formada somente com dígitos decimais. Cada símbolo lido em $12324A563$ deve ser um dígito. Esta palavra é curta, e, portanto, não parece ser necessária nenhuma disciplina na sua leitura para encontrar o A que foge à regra de escrita de numerais decimais. No entanto, uma palavra com 10^100 caracteres justapostos necessita de alguma disciplina de leitura. Como ter certeza que não há caracteres que não pertencem a $\{0,1,2,3,4,5,6,7,8,9\}$? Somente verificando sistematicamente. Se a linguagem fosse dos números pares até 10^{100} teríamos que verificar também se o último dígito está em $\{0,2,4,6,8\}$. Cada linguagem formal traz muitas possibilidades. Para tornar o problema mais interessante ainda, lembramos que as linguagens mais usadas no dia a dia e em computação são infinitas. Portanto, um procedimento de verificação é essencial neste caso, dado que não há como especificar ou verificar pertinência que não seja via procedimento computacional.

Voltando ao reconhecimento de uma linguagem formal, façamos um paralelo com a forma com que nós, seres humanos, processamos um texto em linguagem natural. Ao lermos um texto, uma tarefa que

nos é exigida imediatamente é a segmentação do texto em palavras. Somente depois de reconhecermos as palavras é que as frases ou as orações são processadas - em processamento de linguagem natural por meio de programas de computador ou aplicativos, essa também é a primeira tarefa. Em linguagens de programação esta etapa recebe o nome de análise léxica e é exatamente isso, percorre-se o texto agrupando os símbolos em agregados (*clusters*) que denominamos, na terminologia de compiladores, de *itens léxicos*. Em termos de especificação da linguagem esses são os símbolos do nosso alfabeto. Exemplos de itens léxicos na linguagem de programação C são os identificadores, nomes de variáveis, nomes de funções e palavras reservadas tais como "*main*", "*function*", "*while*" etc. (AHO et al., 2006)

Uma curiosidade com relação à formação dos itens léxicos é o fato da leitura/escrita de um texto se dar em diferentes direções nas linguagens naturais. Mandarim e japonês são lidos/escritos de cima para baixo e em seguida da esquerda para a direita. Hebraico e árabe são escritos e lidos da direita para a esquerda, enquanto a maioria das línguas ocidentais é escrita da esquerda para a direita. No entanto, alguns nomes próprios que podem ser escritos em todas estas linguagens são os mesmos nomes, estejam eles na vertical, escritos da direita para a esquerda ou escritos da esquerda para a direita. Por exemplo, Saul em hebraico é escrito לואש(lwu'ahS), lido da direita para esquerda é praticamente o que vemos em grego como $\Sigma\alpha\upsilon\lambda$, da esquerda para a direita. A direção da escrita/leitura não altera todas as características da palavra, até mesmo quando o alfabeto utilizado muda (NAVEH, 2012). Observe como os símbolos em Saul se relacionam, mesmo estando em alfabetos distintos. Veremos nesta seção que a direção de leitura/escrita em uma linguagem formal não a altera como conjunto de palavras.

O tipo de gramática mais simples é a gramática regular. Uma gramática regular possibilita uma análise bem simples da sua linguagem gerada. Melhor ainda, esta análise é feita de forma sistemática a partir da própria gramática. Lembramos que na Unidade 1 definimos uma gramática regular como uma gramática G = (V,T,P,S), que só possui regras da forma $A \to b$, $A \to \epsilon$ ou $A \to aB$, com $A,B \in V$ e $a,b \in T$. Observe que A e B aqui representam não terminais quaisquer de V, podendo ser iguais entre si ou diferentes. Da mesma forma a e b representam terminais quaisquer do conjunto T. (HOPCROFT; ULLMAN, 1969) (GARCIA, 2017).

Assimile

Uma gramática G = (V,T,P,S) é regular se, e somente se, só possui regras da forma $A \to b$, $A \to \epsilon$ ou $A \to aB$, com $A,B \in V$ e $a,b \in T$.

Observe em particular a linguagem $L = \{aab, bba\}$ sobre o alfabeto $\Sigma = \{a,b\}$. Esta linguagem pode ser gerada pela seguinte gramática regular:

$S \to aA_1$,
$S \to bB_1$,
$A_1 \to aA_2$,
$A_2 \to b$,
$B_1 \to bB_2$,
$B_2 \to a$

Observe que a primeira cadeia pode ser gerada pela derivação:

$$S \Rightarrow aA_1 \Rightarrow aaA_2 \Rightarrow aab$$

Enquanto que a segunda cadeia pode ser gerada pela derivação:

$$S \Rightarrow bB_1 \Rightarrow bbB_2 \Rightarrow bba$$

Reflita

No exemplo anterior exibimos uma gramática regular que gera uma linguagem finita. Você pode dizer se para toda a linguagem finita existe uma gramática regular que a gere?

Lembramos que na Unidade 1 definimos que uma linguagem é regular se, e somente se, existir uma gramática regular que a gere. Assim, dada uma gramática G que não é regular, é possível que a linguagem $L(G)$ seja regular, bastando que, para isso, exista uma gramática regular G_2 tal que $L(G_2) = L(G)$.

Exemplificando

Considere a gramática:

$S \to A_1 b$,

$S \to B_1 a$,

$A_1 \to A_2 a$,

$A_2 \to a$,

$B_1 \to B_2 b$,

$B_2 \to b$

Deve estar claro que esta gramática não é regular, entretanto, temos que $L(G) = \{aab, bba\}$ e vimos anteriormente que esta linguagem é gerada por uma gramática regular. Portanto, $L(G)$ é regular.

A gramática do exemplo anterior é um exemplo de gramática linear à esquerda, onde as regras podem ser da forma $A \to b$, $A \to \epsilon$ ou $A \to Ba$, com $A, B \in V$ e $a, b \in T$. A linguagem gerada por gramáticas deste tipo é regular.

Da mesma forma, gramáticas lineares à direita são aquelas em que as regras são da forma $A \to b$, $A \to \epsilon$ ou $A \to aB$, com $A, B \in V$ e $a, b \in T$. Estas gramáticas são o que definimos como gramáticas regulares. Alguns autores consideram ambos os tipos (tanto as lineares à esquerda quanto as lineares à direita) gramáticas regulares, por exemplo, Menezes (2000).

Regras simples da forma $A \to B$ não acrescentam poder a essas gramáticas. Estas regras podem ser substituídas para obtermos uma gramática regular equivalente. Considere o exemplo:

$S \to A$,

$S \to a$,

$A \to aB$,

$A \to B$,

$B \to b$.

Para esta gramática observamos que $S \Rightarrow^* A$, $S \Rightarrow^* B$, $A \Rightarrow^* B$,

portanto, podemos substituir o lado direito das regras simples da forma $C \to D$ pelo lado direito das regras cujo lado esquerdo é D. Neste caso obtemos a seguinte gramática regular equivalente:

$S \to aB$,
$S \to b$,
$S \to a$,
$A \to aB$,
$A \to b$,
$B \to b$.

Esta mesma gramática pode ser apresentada de forma mais concisa quando juntamos as regras que têm o mesmo lado esquerdo:

$S \to aB \mid b \mid a$,
$A \to aB \mid b$,
$B \to b$.

Portanto, a linguagem gerada por uma gramática apenas com regras regulares e regras simples é regular. Este resultado é importante porque nos permite apresentar um resultado central para esta seção, a saber: se L_1 e L_2 são linguagens regulares, então $L_1 \cup L_2$ também é regular.

O motivo é o seguinte: se L_1 e L_2 são linguagens regulares então existem gramáticas regulares $G_1 = (V_1, T, P_1, S_1)$ e $G_2 = (V_2, T, P_2, S_2)$ tais que $L(G_1) = L_1$ e $L(G_2) = L_2$. Podemos renomear as variáveis de V_2 de modo que não tenham o mesmo nome que uma variável de V_1. Podemos agora criar um novo símbolo inicial S e criar a gramática $G_3 = (V_1 \cup V_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$. É fácil ver que G_3 é uma gramática apenas com regras regulares e regras simples e que $L(G_3) = L(G_1) \cup L(G_2)$, portanto, $L(G_1) \cup L(G_2)$ é regular.

Exemplificando

Considere a gramática G_1:

$S_1 \to bA_1$,

$A_1 \to a$.

e a gramática G_2:

$S_2 \to aA_2 \mid bB_2$,

$A_2 \to a$,

$B_2 \to bB_2 \mid b$.

Podemos construir a nova gramática G_3 tal que $L(G_3) = L(G_1) \cup L(G_2)$:

$S \to S_1 \mid S_2$,

$S_1 \to bA_1$,

$A_1 \to a$,

$S_2 \to aA_2 \mid bB_2$

$A_2 \to a$,

$B_2 \to bB_2 \mid b$.

Observe que G_3 é uma gramática apenas com regras regulares e simples, portanto, gera uma linguagem regular.

Voltemos a considerar a gramática G_2:

$S_2 \to aA_2$,

$S_2 \to bB_2$,

$A_2 \to a$,

$B_2 \to bB_2$,

$B_2 \to b$.

Queremos agora encontrar uma gramática que gere a linguagem $L(G_2)^*$. Uma vez que G_2 só tem regras da forma $A \to b$ e $A \to aB$, e que o símbolo inicial não aparece do lado direito, essa construção será

bem fácil. Basta colocar o símbolo inicial nas regras da forma $A \to b$ e acrescentar a regra $S_2 \to \epsilon$, obtendo a gramática:

$S_2 \to \epsilon$
$S_2 \to aA_2$,
$S_2 \to bB_2$,
$A_2 \to aS_2$,
$B_2 \to bB_2$,
$B_2 \to bS_2$.

Exemplificando

Vamos mostrar um exemplo um pouco mais sofisticado. A gramática a seguir, com símbolo inicial A_0, gera os números inteiros, escritos na base 2, que deixam resto 1 quando divididos por 3:

$A_0 \to 0A_0$, $A_0 \to 1A_1$,

$A_1 \to 0A_2$, $A_1 \to 1A_0$, $A_1 \to \epsilon$,

$A_2 \to 0A_1$, $A_2 \to 1A_2$.

Observe que $A_0 \Rightarrow^* wA_i$ se, e somente se, w é um número na base 2 que deixa resto i quando dividido por 3.

Na próxima seção vamos estudar como são os algoritmos que realizam o processamento, ou análise sintática, de uma linguagem regular. Veremos que este mecanismo pode ser obtido facilmente a partir das gramáticas regulares que geram as respectivas linguagens.

Pesquise mais

Pesquise como é o mecanismo de marcação para a escrita de comentários na sua linguagem de programação favorita. Verifique se este mecanismo pode ser descrito por uma gramática regular.

Acesse o material disponível em: <http://bit.ly/2qo5dDJ>. Acesso em: 10 jul. 2017.

Sem medo de errar

Uma vez que você já está familiarizado com gramáticas e linguagens regulares, você foi capaz de identificar que o padrão especificado – que são números de ponto flutuante negativos – pode ser facilmente descrito por gramáticas regulares. Você sabe que gramáticas regulares podem ser facilmente reconhecidas por ferramentas eficientes, que consomem tempo linear no tamanho da entrada. Portanto, você propõe especificar o padrão usando uma gramática regular e posteriormente irá programar seu reconhecimento usando um algoritmo eficiente, como o Autômato Finito Determinístico.

Para especificar a sua solução você primeiro identifica o alfabeto em que o padrão está escrito. Este será o conjunto de símbolos terminais de sua gramática $T = \{-,.,0,1,2,3,4,5,6,7,8,9\}$. Depois você parte para a especificação das regras. A seguir, uma especificação para o mesmo padrão para números de ponto flutuante negativos na base 2. A única diferença é que há apenas os dígitos 0 e 1, portanto, $T = \{-,.,0,1\}$. Neste caso temos as regras, com comentários à direita.

$S \to -A$ //Coloca o sinal de menos.

$A \to 0$ //O número pode ter apenas um dígito.

$A \to 1$

$A \to 0A$ //O número pode ter mais que um dígito.

$A \to 1A$

$A \to 0B$ //O número pode continuar com uma parte decimal.

$A \to 1B$

$B \to .C$

$C \to 0$ //A parte decimal pode ter um dígito.

$C \to 1$

$C \to 0C$ //A parte decimal pode ter mais do que um dígito.

$C \to 1C$

Avançando na prática

Linearidade à esquerda e à direita

Descrição da situação-problema

Discutimos o problema da direção em que se realiza o processamento de uma linguagem escrita. Estávamos discutindo o fato da direção de leitura/escrita ser ou da direita para a esquerda ou da esquerda para direita. Exemplificamos com um par de palavras o nome "Saul", que são essencialmente as mesmas, apesar da direção de processamento nestas serem diferentes. Em geral, parece ser possível considerarmos que uma mesma linguagem (regular) possa ser descrita, ou processada, tanto da esquerda para a direita quanto da direita para a esquerda. Observe a linguagem formada por todas as palavras formadas somente com a's e b's onde qualquer ocorrência de a acontece antes de todas as ocorrências de b.

Por exemplo, as duas gramáticas a seguir são capazes de gerar esta linguagem:

G1 =

$$S \rightarrow aS \mid a \mid b \mid bB$$
$$B \rightarrow b \mid bB$$

e

G2 =

$$S \rightarrow Sb \mid a \mid Aa \mid b$$
$$A \rightarrow Aa \mid a$$

G1 gera da esquerda para a direita e G2 o faz da direita para a esquerda. Gramáticas do tipo de G1 são as gramáticas regulares, também denominadas de gramáticas lineares à direita, enquanto G2 é uma gramática linear à esquerda.

Demonstre que para toda a gramática linear à esquerda existe uma gramática regular que gera a mesma linguagem. Isto praticamente nos faz concluir que a direção de processamento de uma linguagem não é essencial.

Resolução da situação-problema

Em função do que vimos nesta seção, sabemos que não precisamos levar em consideração gramáticas que geram linguagens finitas. Vamos então considerar um padrão que pode servir de modelo para qualquer linguagem infinita. Considere uma linguagem infinita gerada por uma gramática linear à esquerda de G_e. Se isso acontece é porque existe um não terminal A, tal que as seguintes derivações são possíveis em G_e:

$$A \Rightarrow^{+} A\gamma_1 | \ldots | A\gamma_k$$

$$A \Rightarrow^{+} \delta_1 | \ldots | \delta_m$$

Onde δ_i e γ_j são formados apenas por símbolos do alfabeto. Para ver a necessidade da existência de A com esta propriedade, suponha que tal A não existe e você será capaz de concluir que a linguagem gerada por G_e é finita.

Veja então que as regras descritas a seguir podem substituir as regras que tem A à esquerda e geram a mesma sublinguagem gerada por A. Considere B uma variável nova em G_e

$$A \to \delta_1 | \ldots | \delta_m | \delta_1 B | \ldots | \delta_m B$$

$$B \to \gamma_1 | \ldots | \gamma_k | \gamma_1 B | \gamma_k B$$

A gramática resultante ainda não é uma linear à direita, porque δ_i e γ_i podem ter mais que um símbolo terminal. Neste caso, basta aplicar um procedimento já visto em um exemplo desta seção. Por exemplo, a regra $A \to abA$ pode ser substituída pelas regras: $A \to aZ$ e $Z \to bA$.

Faça valer a pena

1. Lembramos que uma gramática regular é aquela na qual o lado esquerdo de cada regra possui exatamente um símbolo, e ele é não terminal. O lado direito da regra é vazio ou possui um único símbolo terminal, ou ainda possui um símbolo terminal seguido de um único símbolo não terminal. Levando em conta a definição dada e as convenções sobre apresentação de gramáticas estudadas na Unidade 1, assinale a gramática regular:

a) $S \to AA$,
$A \to aA$,
$A \to a$,
$A \to \epsilon$

b) $S \to aA$,
$A \to aAa$,
$A \to a$,
$A \to \epsilon$

c) $S \to aA$,
$A \to aA$,
$A \to aa$,
$A \to \epsilon$

d) $S \to aA$,
$A \to aA$,
$A \to aS$,
$A \to aa$,
$A \to \epsilon$

e) $S \to aA$,
$A \to aA$,
$A \to a$,
$A \to \epsilon$

2. Considere a gramática regular $G_1 = (V, T, P, S)$, cujas regras de derivação são apresentadas a seguir:
$S \to aA$,
$S \to bB$,
$A \to a$,
$B \to bB$,
$B \to b$.
Assinale a alternativa que apresenta uma gramática regular que gera a linguagem $L(G_1)^*$:

a) $S \to SS$, $S \to aA$, $S \to bB$, $A \to a$, $B \to bB$, $B \to b$.

b) $S \to \epsilon$, $S \to aA$, $S \to bB$, $A \to a$, $B \to bB$, $B \to b$.

c) $S \to \epsilon$, $S \to aA$, $S \to bB$, $A \to aS$, $B \to bB$, $B \to b$.

d) $S \to \epsilon$, $S \to aA$, $S \to bS$, $A \to aS$.

e) $S \to \epsilon$, $S \to aA$, $S \to bB$, $A \to aS$, $B \to bB$, $B \to Sb$.

3. Lembramos que uma gramática é regular se possui apenas regras regulares. Considerando esta definição, assinale a alternativa verdadeira:

a) A gramática $A \to aB$, $A \to Ba$, $B \to \epsilon$ é regular.

b) A gramática $A \to aB$, $A \to Ba$, $B \to a$ é regular.

c) A gramática $A \to aA$, $A \to a$ é regular.

d) A gramática $A \to aBb$, $B \to \epsilon$ é regular.

e) A gramática $A \to AA$, $A \to a$ é regular.

Seção 2.2

Autômatos finitos

Diálogo aberto

Você apresentou sua especificação dos padrões de números negativos prometendo que usaria técnicas de reconhecimento de linguagens regulares para desenvolver sua solução. Sua proposta agradou seus superiores e gerou uma expectativa positiva e sua próxima tarefa será liderar a programação da solução que você especificou.

Nesta seção, você conhecerá em detalhes os Autômatos Finitos Determinísticos (AFDs), que são reconhecedores eficientes de linguagens regulares. A partir da gramática regular que você já criou você deve gerar a especificação de um AFD e orientar sua equipe a programá-lo.

Não pode faltar

Vimos na seção anterior que o problema geral da análise sintática é muito difícil. Felizmente, em alguns casos particulares ele se torna mais simples. Nesta seção, vamos estudar um destes casos. Vamos tomar como exemplo a seguinte gramática G:

$S \rightarrow 0A \mid 1S$

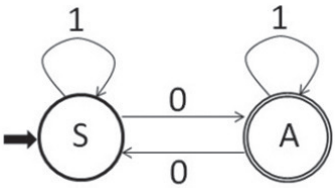
$A \rightarrow 0S \mid 1A \mid \epsilon$

Esta gramática gera as cadeias sobre o alfabeto $\Sigma = \{0,1\}$, que têm quantidade ímpar de caracteres '0'. Por exemplo, esta gramática gera a cadeia 01010 da seguinte forma:

$$S \Rightarrow 0A \Rightarrow 01A \Rightarrow 010S \Rightarrow 0101S \Rightarrow 01010A \Rightarrow 01010$$

Programar a análise sintática é resolver o problema oposto. Dada a cadeia 01010, podemos determinar se ela está na linguagem gerada pela gramática? Para programar a solução podemos guardar sempre qual é o único não terminal que está no final da cadeia sendo gerada. Se ao final o não terminal for 'A', como existe a regra $A \rightarrow \epsilon$, então a cadeia estará na linguagem. Esta solução pode ser representada pela Figura 2.1.

Figura 2.1 | Autômato finito determinístico



Fonte: elaborada pelo autor.

Chamamos esta solução de AFD – Autômato Finito Determinístico. A ideia é que cada bola representa um "estado". Começamos pelo estado marcado com a seta, no caso, o estado 'S'. Ao lermos um 0 vamos para o estado 'A'; ao lermos um '1', permanecemos no estado 'A'. Depois, ao lermos o segundo '0', voltamos ao estado 'S'. Ao terminarmos de ler a cadeia de entrada, se estivermos em um estado marcado com círculo duplo, neste caso no estado 'A', dizemos que a cadeia foi 'reconhecida'. Caso contrário, a cadeia não foi reconhecida pelo AFD (GARCIA, 2017).

Em um AFD, dado um estado e um símbolo do alfabeto de entrada, existe uma seta, ou transição para exatamente um único estado. Assim , o AFD está sempre em exatamente um estado, por isso é chamado de determinístico. Observamos que o AFD possui estados. No AFD da Figura 2.1 os estados são 'S' e 'A'. O estado 'S' é o estado inicial (marcado com uma seta), no qual o AFD inicia sua execução. O estado 'A' é um estado final e se o autômato termina em um estado final (marcado com círculo duplo), ele reconheceu com sucesso a cadeia lida. Uma forma alternativa de representar este mesmo AFD é a Tabela 2.1:

Tabela 2.1 | Representação tabular do AFD

	0	1
$\rightarrow$S	A	S
*A	S	A

Fonte: elaborada pelo autor.

Neste caso, o estado inicial é marcado com a seta. Os estados finais são marcados com o símbolo '*' e, para sabermos para onde vai a seta do estado q com entrada a basta consultar a linha correspondente ao estado q e a coluna correspondente ao símbolo de entrada a. As

setas - ou a tabela - representam uma função que, dado um estado e um símbolo da entrada, retorna um único estado.

Assimile

Formalmente um AFD M é uma tupla (Q,Σ,δ,q_0,F), onde:

- Q é um conjunto finito e não vazio de estados.
- Σ é o alfabeto de entrada.
- $\delta : Q\times\Sigma \to Q$ é a função de transição de estados.
- $q_0 \in Q$ é o estado inicial
- $F \subseteq Q$ é o conjunto de estados finais.

No AFD da Figura 2.1 temos que:

- $Q = (S, A)$;
- $\Sigma = \{0,1\}$;
- $\delta : Q\times\Sigma \to Q$ é a função na qual:
  $\delta(S,0) = A$, $\delta(S,1) = S$, $\delta(A,0) = S$ e $\delta(A,1) = A$;
- S é o estado inicial;
- $\{A\}$ é o conjunto de estados finais.

Para entendermos como o AFD funciona é útil estendermos a função $\delta : Q\times\Sigma \to Q$ para esta extensão, isto é, a função $\hat{\delta}$,representa a ação do AFD ao ler uma cadeia, ou seja, ao ler mais de um caractere. Sua definição é intuitiva: se ao lermos um 0 temos que $\delta(S,0) = A$, e o AFD fica no estado A. Ao lermos um segundo 0, temos $\delta(A,0) = S$ e o AFD retorna ao estado S. Isso significa que $\hat{\delta}(S,00) = S$. Formalmente podemos definir $\hat{\delta}$ em função de δ da seguinte forma:

- $\hat{\delta}(q,\epsilon) = q$, para todo $q \in Q$.
- $\hat{\delta}(q,aw) = \hat{\delta}(\delta(q,a),w)$, para todo $q \in Q, a \in \Sigma, w \in \Sigma^*$.

Exemplificando

Aplicando esta definição ao AFD da Figura 2.1, com entrada 00, temos o seguinte:

$$\hat{\delta}(S,00) = \hat{\delta}(\delta(S,0),0) = \hat{\delta}(A,0) = \hat{\delta}(\delta(A,0),\epsilon) = \hat{\delta}(S,\epsilon) = S$$

Portanto, a definição funciona como esperávamos.

Finalmente estamos aptos a definir a linguagem gerada por um AFD M. Esta linguagem é formada por todas as cadeias reconhecidas por M, isto é, todas as cadeias que levam M a um estado final.

Assimile

Dado um AFD M = (Q,Σ,δ,q_0,F), definimos a linguagem reconhecida por M como:

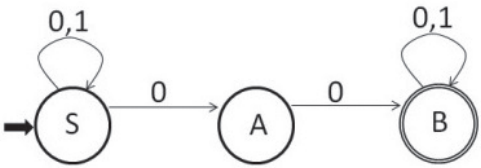
- $T(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0,w) \in F\}$

No caso do autômato da Figura 2.1 temos que T(M)={0,01,10,011,101, 110,000,011,...}, o conjunto de todas as cadeias com número ímpar de caracteres 0. Gostaríamos que o AFD fosse uma boa forma de resolver o problema da análise sintática para todas as linguagens regulares. Entretanto, não é claro como obter um AFD de algumas gramáticas regulares. Por exemplo, a gramática G_2 - descrita logo a seguir -, gera as cadeias sobre o alfabeto $\Sigma = \{0,1\}$ que possuem a subcadeia '00', isto é, que têm dois caracteres 0 seguidos. G_2 possui as regras:

- $S \rightarrow 0S \mid 1S \mid 0A$
- $A \rightarrow 0B$
- $B \rightarrow 0B \mid 1B \mid \epsilon$

Ao tentarmos transformar esta gramática em um autômato finito obtemos a Figura 2.2:

Figura 2.2 | Autômato Finito não Determinístico



Fonte: elaborada pelo autor.

Podemos dizer que nesta figura a função δ não retorna um estado, mas um conjunto de estados. Por exemplo, $\delta(S,0)=\{S,A\}$, enquanto que $\delta(A,1)=\varnothing$, ambos os casos violam a condição de que para cada estado e símbolo de entrada as setas nos levem a exatamente um estado. Neste caso, dizemos que a figura representa um AFND – Autômato Finito Não Determinístico. O AFND da Figura 2.2 também pode ser representado em outra forma , como apresentado na Tabela 2.2:

Tabela 2.2 | Representação Tabular do AFND

	0	1
$\rightarrow$S	{S, A}	{S}
A	B	$\varnothing$
*B	B	B

Fonte: elaborada pelo autor.

Podemos interpretar um AFND como podendo seguir diversos caminhos ao ler uma entrada. Se ao menos 1 dos caminhos levar o AFND a um estado final, dizemos que o AFND reconhece w.

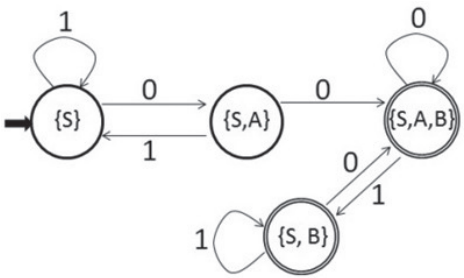
Pesquise mais

Observe que a função δ neste caso não retorna um estado, mas sim um conjunto de estados, em particular pode ser o conjunto vazio. Assim a assinatura da função deve ser $\delta : Q \times \Sigma \rightarrow \wp(Q)$.

O aluno pode encontrar a formalização de um AFND em: Garcia (2007), Menezes (2000) ou no site disponível em: <http://bit.ly/2seo19J>. Acesso em: 11 jul. 2017.

O AFND da Figura 2.2 pode ser transformado em um AFD. Dissemos que um AFND pode seguir diversos caminhos ao ler uma entrada w. Vamos fazer um AFD que guarda os estados de todos esses possíveis caminhos. Como o AFND em questão tem apenas três estados, cada um dos possíveis caminhos seguidos ao lermos uma cadeia w estará em um desses três estados, portanto, os caminhos combinados poderão estar em nenhum estado, ou somente no estado {S}, ou em uma combinação de estados, por exemplo, {S, A, B}. Na pior das hipóteses há $2^3 = 8$ estados possíveis para esses caminhos combinados. Podemos assim desenhar o AFD da Figura 2.3:

Figura 2.3 | AFD obtido a partir de um AFND



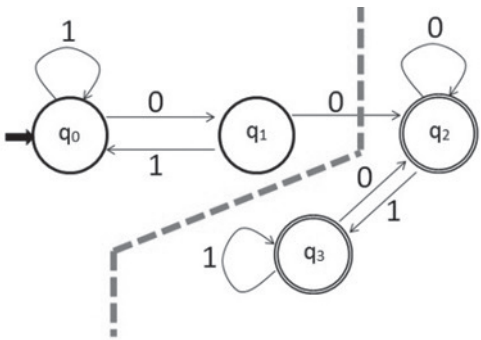
Fonte: elaborada pelo autor.

Conforme explicado, cada estado do AFD corresponde a todos os possíveis estados que o AFND estaria após ler determinada entrada. Em particular, o estado inicial é {S}, porque o AFND inicia apenas em S. Se δ^2 é a função de transição do AFD que desejamos obter, $\delta^2(\{S,A\},0) = \cup_{q\in\{S,A\}}\delta(q,0) = \delta(S,0)\cup\delta(A,0) = \{S,A\}\cup\{B\} = \{S,A,B\}$. Os estados finais do AFD correspondem aos conjuntos de estados do AFND que contém ao menos um estado final. No exemplo {S,A,B} e {S,B} são estados finais, porque contém o estado final B.

Dada uma gramática regular sabemos obter um AFND equivalente, a construção anterior nos mostra que dado um AFND podemos obter um AFD equivalente, portanto, para toda gramática regular G sabemos obter um AFD M equivalente, isto é, tal que $T(M) = L(G)$. Isso significa que podemos fazer a análise sintática de forma eficiente para todas as linguagens regulares. Se fizermos um programa que simule um AFD, este consome tempo proporcional ao tamanho da entrada, pois para cada símbolo da cadeia de entrada o AFD precisa fazer um movimento entre dois estados. O consumo de memória, por sua vez, é constante

e não depende do tamanho da entrada, basta armazenar a tabela de transição em uma matriz. Entretanto, no AFD obtido na Figura 2.3, é possível diminuir o número de estados sem afetar seu funcionamento. A Figura 2.4 exibe o mesmo AFD que a Figura 2.3. Seus estados foram renomeados para a forma mais usual $q_0, q_1, \ldots$, e desenhamos uma linha tracejada para facilitar o entendimento da melhoria que pode ser feita no AFD.

Figura 2.4 | AFD da figura 2.3 com estados renomeados



Fonte: elaborada pelo autor.

Na Figura 2.4 há uma seta que ultrapassa a linha tracejada da esquerda para a direita, e não há seta no sentido oposto. Isto significa que uma vez ultrapassada a seta, não há "volta", o que corresponde à especificação da linguagem, composta pelas cadeias sobre $\Sigma = \{0,1\}$ que possuem a subcadeia '00'. Uma vez que a subcadeia '00' ocorra na cadeia, qualquer continuação da cadeia terá esta característica.

Reflita

Usando as observações acima sobre a Figura 2.4, você seria capaz de produzir um AFD equivalente ao da Figura 2.4 com menos estados?

Em um AFD, os estados q e r são equivalentes, se para toda entrada possível o AFD se comporta da mesma forma (reconhece as mesmas entradas) estando em q ou estando em r.

Assimile

Dado um AFD M = (Q,Σ,δ,q_0,F), e estados $q,r \in Q$, definimos que q e r são equivalentes, ou $q \equiv r$, quando:

- $\forall w \in \Sigma^*, \delta(q,w) \in F \Leftrightarrow \delta(r,w) \in F$

Na Figura 2.4 é apresentado o caso que $q_2 \equiv q_3$ porque depois da linha tracejada qualquer entrada sempre leva a um estado final. Em geral, é mais fácil identificar estados que não são equivalentes do que estados que são equivalentes. Se fizermos a negação da definição de equivalência temos que:

$q \not\equiv r$ se, e somente se, $\exists w \in \Sigma^*, \delta(q,w) \in F \Leftrightarrow \delta(r,w) \notin F$

Esta condição pode ser dividida no caso em que w é a cadeia vazia e o caso onde w não é a cadeia vazia. Se w não é a cadeia vazia e $q \not\equiv r$, então $w = aw_2$, onde $a \in \Sigma$, e $\delta(q,a) \not\equiv \delta(r,a)$, isso significa que a condição de não equivalência pode ser reescrita em dois casos, isto é, $q \not\equiv r$ se um dos seguintes casos é verdadeiro:

1. $q \in F \Leftrightarrow r \notin F$
2. $\exists a \in \Sigma, \delta(q,a) \not\equiv \delta(r,a)$

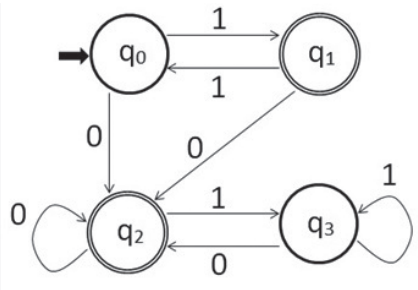
A partir disso podemos fazer um algoritmo de minimização que consiste em dois passos:

1. Marcar como não equivalente todo o par de estados (q,r) no qual $q \in F \Leftrightarrow r \notin F$;

2. Percorrer todo o par de estados restante, marcando-o como não equivalente quando $\exists a \in \Sigma, \delta(q,a) \not\equiv \delta(r,a)$.

Veremos isso através de um exemplo. Considere o AFD da Figura 2.5.

Figura 2.5 | AFD a ser minimizado



Fonte: elaborada pelo autor.

Vamos fazer uma matriz triangular conforme a Tabela 2.3, onde cada par de estados é considerado uma única vez. Nesta matriz vamos marcar com um 'X' os pares de estados que atendem à primeira condição: $q \in F \Leftrightarrow r \notin F$.

Tabela 2.3 | Primeira etapa da minimização do AFD

q_1	X		
q_2	X		
q_3		X	x
	q_0	q_1	q_2

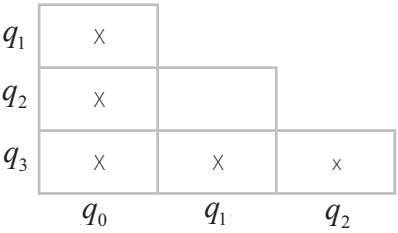
Fonte: elaborada pelo autor.

Após essa marcação falta analisar dois pares de estados: (q_0, q_3) e (q_1, q_2). Começaremos com (q_0, q_3). Neste caso, temos que:

- $\delta(q_0, 0) = \delta(q_3, 0)$
- $\delta(q_0, 1) = q_1 \not\equiv q_3 = \delta(q_3, 1)$

Devido ao movimento na segunda linha temos que $\exists a \in \Sigma, \delta(q_0, a) \not\equiv \delta(q_3, a)$, portanto $q_0 \not\equiv q_3$, podemos marcá-los na Tabela 2.4 como não equivalentes:

Tabela 2.4 | Segunda etapa da minimização do AFD



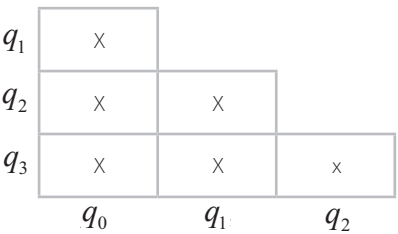
Fonte: elaborada pelo autor.

Finalmente, devemos analisar o par (q_1, q_2). Neste caso, temos que:

- $\delta(q_1, 0) = \delta(q_2, 0)$
- $\delta(q_1, 1) = q_0 \not\equiv q_3 = \delta(q_2, 1)$

Mais uma vez, o movimento da segunda linha faz com que o par não seja equivalente. Portanto, a Tabela 2.5 fica toda marcada com 'X', o que significa que não há estados equivalentes, portanto, o AFD não pode ser melhorado.

Tabela 2.5 | Terceira etapa da minimização do AFD



Fonte: elaborada pelo autor.

É interessante voltarmos ao final da primeira fase, onde a situação era dada pela Tabela 2.3. Se escolhêssemos analisar primeiro o estado (q_1, q_2), devido à condição $\delta(q_1, 1) = q_0$ e $\delta(q_2, 1) = q_3$, não saberíamos se (q_1, q_2) é equivalente ou não porque ainda não sabemos se (q_0, q_3) o é. Neste caso, devemos colocar um ponteiro de (q_0, q_3) para (q_1, q_2) nos alertando que se no futuro formos marcar (q_0, q_3) como não equivalentes, devemos seguir este ponteiro e marcar também (q_1, q_2). A situação pode ser desenhada na Tabela 2.6:

Tabela 2.6 | Dependência entre estados na minimização do AFD

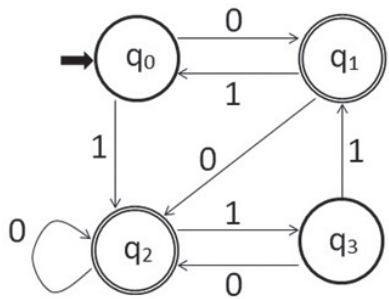
q_1	X		
q_2	X		
q_3	(q_1, q_2)	X	x
	q_0	q_1	q_2

Fonte: elaborada pelo autor.

Depois disso, ao visitarmos (q_0, q_3), toda a tabela ficará marcada como não equivalente, conforme obtivemos anteriormente. Note que alguns autores representam a marcação de ponteiros de forma diferente, a escolhida neste texto é aquela que é mais conveniente para a programação. É interessante mostrar mais um exemplo, onde existam estados equivalentes.

Vamos analisar um novo AFD na Figura 2.6:

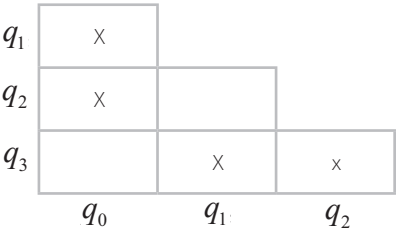
Figura 2.6 | AFD a ser minimizado



Fonte: elaborada pelo autor.

Ao minimizarmos este AFD, em primeiro lugar devemos marcar como não equivalentes os estados finais com os não finais e vice-versa.

Tabela 2.7 | Primeira etapa de minimização do AFD da Figura 2.6



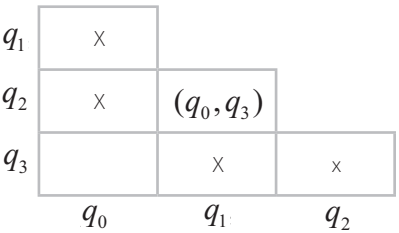
Fonte: elaborada pelo autor.

Falta analisarmos os pares: (q_0, q_3) e (q_1, q_2). Começaremos com (q_0, q_3). Neste caso, temos que:

- $\delta(q_0, 0) = q_1$ e $\delta(q_3, 0) = q_2$
- $\delta(q_0, 1) = q_2$ e $\delta(q_3, 1) = q_1$

Ambas as linhas dependem do par e devemos indicar a dependência por um ponteiro, obtendo a Tabela 2.8:

Tabela 2.8 | Segunda etapa de minimização do AFD da Figura 2.6



Fonte: elaborada pelo autor.

Finalmente analisamos (q_1, q_2):

- $\delta(q_1, 0) = \delta(q_2, 0)$
- $\delta(q_1, 1) = q_0$ e $\delta(q_2, 1) = q_3$

Obtemos uma dependência circular na Tabela 2.9:

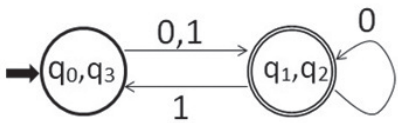
Tabela 2.9 | Terceira etapa de minimização do AFD da Figura 2.6

q_1	X		
q_2	X	(q_0, q_3)	
q_3	(q_1, q_2)	X	X
	q_0	q_1	q_2

Fonte: elaborada pelo autor.

Chegamos ao final do algoritmo e os pares (q_0, q_3) e (q_1, q_2) não foram marcados. Podemos considerar todos os pares não marcados como equivalentes. Portanto, podemos juntar (q_0, q_3) em um único estado, o mesmo se dá para (q_1, q_2). Desenhamos o autômato minimizado na Figura 2.7:

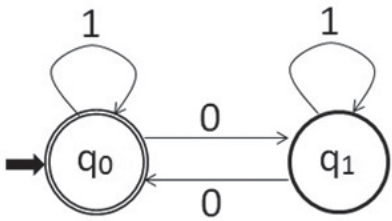
Figura 2.7 | AFD da Figura 2.6 minimizado



Fonte: elaborada pelo autor.

Vimos que todo o AFND pode ser transformado em um AFD equivalente e que existe um algoritmo para encontrar o menor AFD possível. Considere o AFD da Figura 2.1. Ele reconhece a linguagem $L_{i0} = \{w|$ w tem número ímpar de 0's. Como construir um AFD que reconheça o complemento desta linguagem? Basta simplesmente marcar os estados não finais como finais e vice-versa. Assim, as cadeias reconhecidas serão aquelas que não eram reconhecidas anteriormente. O AFD obtido, após renomearmos os estados, é o da Figura 2.8.

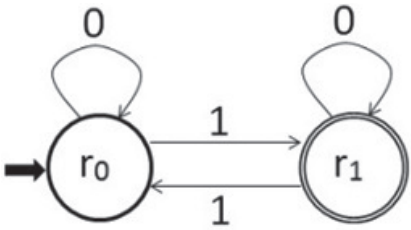
Figura 2.8 | AFD que reconhece o complemento da linguagem L_{i0}



Fonte: elaborada pelo autor.

Uma vez que podemos construir um AFD para toda a linguagem regular e vice-versa, a consequência desta construção é que as linguagens regulares são fechadas sob complemento, isto é, se L é uma linguagem regular, então seu complemento, $\overline{L}$, também o é. Podemos agora considerar um autômato finito determinístico bem semelhante, que reconhece as cadeias sobre $\Sigma = \{0,1\}$ que tem uma quantidade ímpar de 1's. A Figura 2.9 exibe este AFD.

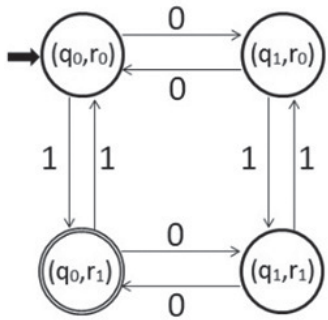
Figura 2.9 | AFD que reconhece cadeias com número ímpar de 1´s



Fonte: elaborada pelo autor.

Um problema importante é: dados os AFDs M_1 e M_2 como fazer um AFD que reconheça a linguagem $T(M_1) \cap T(M_2)$? Tomando os AFDs $M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ da Figura 2.8 e $M_2 = (Q_2, \Sigma, \delta_2, r_0, F_2)$ da Figura 2.9, como fazer um AFD que reconheça as cadeias que tem um número par de 0's e ímpar de 1's? Vamos fazer isso construindo o AFD produto, $M_3 = (Q_3, \Sigma, \delta_3, p_0, F_3) = M_1 \times M_2$, no qual o conjunto de estados é o produto cartesiano dos conjuntos de estados dos AFDs originais, isto é, $Q_3 = Q_1 \times Q_2$ e no qual a função de transição consiste em aplicar as funções de transição em paralelo, isto é, $\delta_3((q,r),a) = (\delta_1(q,a), \delta_2(r,a))$. Escolhemos como estado inicial $p_0 = (q_0, r_0)$ e marcamos como estados finais os estados (q,r) onde $q \in F_1$ e $r \in F_2$. O autômato obtido é o da Figura 2.10:

Figura 2.10 | AFD produto



Fonte: elaborada pelo autor.

Observe que o AFD obtido possui certas simetrias. Sempre que lemos um '0', passamos do lado esquerdo para o lado direito, ou do direito para o esquerdo. Sempre que lemos um '1', passamos da parte de cima para a parte de baixo, ou o contrário. Portanto, a parte da esquerda tem sempre uma quantidade par de 0's e a parte de baixo tem sempre uma quantidade ímpar de 1's.

Uma vez que uma linguagem é regular se, e somente se, é reconhecida por um AFD, a construção acima nos assegura que a classe das linguagens regulares é fechada sob a operação de interseção, isto é, dadas as linguagens regulares L_1 e L_2, a linguagem $L_1 \cap L_2$ também é uma linguagem regular. Na próxima seção, iremos abordar outro formalismo para especificar linguagens regulares que possuem muitas aplicações práticas, as expressões regulares.

Sem medo de errar

Lembramos que você já entregou a especificação do problema como uma gramática regular. Basta agora obter o AFND equivalente, que reconheça a linguagem gerada pela gramática, usando a metodologia desenvolvida na seção. Este AFND pode ser transformado em um AFD e uma vez obtida a tabela de transição do AFD você pode programá-lo de forma eficiente, conforme o seguinte pseudocódigo:

```
q = estado_inicial
enquanto ( (entrada = leCaractere())!= EOF) faça {
    q = delta[q,entrada]
}
retorne eh_estado_final(q)
```

Neste algoritmo, q representa o estado atual do AFD e delta[] é uma matriz que implementa a função de transição δ.

O tempo de execução deste algoritmo é proporcional ao tamanho de entrada, uma vez que executa o *loop* uma vez para cada caractere lido. Portanto, é uma maneira eficiente de programar o AFD obtido.

Avançando na prática

Divisibilidade de numerais binários

Descrição da situação-problema

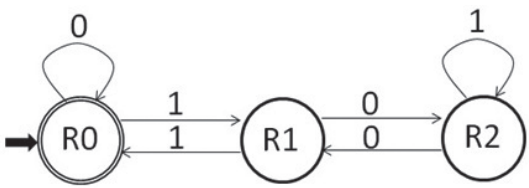
Suponha que você está trabalhando com linguagens sobre o alfabeto {0,1}, ou seja, suas cadeias de caracteres podem ser vistas como numerais binários. Para saber se um numeral binário representa um número par, basta verificar se ele termina com o "0", ou seja, com "0" mais à direita. Saber se o numeral representa um múltiplo de 4 é um pouco mais elaborado. Um número é múltiplo de 4 se puder ser dividido duas vezes por 2. Se o numeral deste número é da forma $b_n b_{n-1} \ldots b_0$ e ele é divisível por 2 então, para ser múltiplo de 4, b_0 e b_1 devem ser 0. Este problema de divisibilidade de números pode ser descrito por autômatos determinísticos? Ou seja, dado um n qualquer, $n \neq 0$, existe um AFD A que aceita todas as cadeias em binário que são divisíveis por n? Vamos focar em algo um pouco mais simples. Elabore uma resposta para $n = 3, 5, 7$. Depois, tente generalizar.

Resolução da situação-problema

Vamos começar com n=3. Considere os restos possíveis de divisão por 3: resto 0, resto 1 e resto 2. Vamos então ter um

autômato com três estados R0, R1 e R2. R0 é estado inicial e final. A ideia do funcionamento é que estaremos no estado R0 após lermos um número com resto 0 na divisão por 3. Neste estado, se o autômato ler 0, o novo número representa o dobro do anterior, portanto também deixa resto 0 na divisão por 3. Por exemplo, queremos que o autômato após ler 11, igual a 3 na base 10, esteja no estado R0. Neste estado, ao ler um 0 passará a ter lido 110, igual a 6 na base 10, o dobro do anterior, portanto deve continuar em R0. Se o autômato ler 1 então temos uma transição para R1, pois o novo número corresponde ao dobro do anterior mais 1, portanto, deixa resto 1. Quando estamos no estado R1 e lemos outro 1, então voltarmos para R0. Observe que $3 = 1 \times 2 + 1$. Cada estado tem transições neste tipo de raciocínio. Assim obtemos o autômato da Figura 2.11.

Figura 2.11 | AFD que reconhece os múltiplos de 3 na base 2.



Fonte: elaborada pelo autor.

A cadeia vazia é aceita, mas isso não será um problema, desde que você crie outro autômato com um estado a mais ou considere a cadeia vazia como uma notação alternativa para o 0, mantendo este autômato. Para n=5 usamos estados R0, R1,...,R4 e raciocínio análogo.

Faça valer a pena

1. Considere o autômato finito determinístico definido pela tabela:

	0	1
$\rightarrow {}^{*}q_0$	q_1	q_0
q_1	q_2	q_1
q_2	q_0	q_2

Assinale a alternativa correta:
a) Toda a cadeia reconhecida pelo autômato tem tamanho múltiplo de 3.
b) Toda a cadeia reconhecida pelo autômato tem tamanho múltiplo de 2.
c) Toda a cadeia reconhecida pelo autômato tem quantidades de caractere '1' múltipla de 3.
d) Toda a cadeia reconhecida pelo autômato tem quantidades de caractere '1' múltipla de 2.
e) Toda a cadeia reconhecida pelo autômato tem quantidades de caractere '0' múltipla de 3.

2. Considere o autômato finito definido pela tabela:

	0	1
$\rightarrow {}^{*}q_0$	q_1	q_3
$^{*}\ q_1$	q_1	q_3
$^{*}\ q_2$	q_1	q_0
q_3	q_2	q_0

Assinale a alternativa correta:
a) O autômato mínimo possui 2 estados porque $q_0 \equiv q_3$ e $q_1 \equiv q_2$.
b) O autômato mínimo possui 2 estados porque $q_1 \equiv q_2 \equiv q_3$.
c) O autômato mínimo possui 3 estados porque $q_1 \equiv q_2$.
d) O autômato mínimo possui 3 estados porque $q_0 \equiv q_3$.
e) O autômato mínimo possui 4 estados porque não há estados equivalentes.

3. Considere o AFD definido pela tabela:

	0	1
$\rightarrow$S	S	A
*A	B	S
B	A	B

Assinale a gramática regular que gera a linguagem reconhecida pelo AFD:

a) $S \rightarrow 0S \mid 1A \mid \epsilon$,
$A \rightarrow 0B \mid 1S$,
$B \rightarrow 0A \mid 1B$.

b) $S \rightarrow 0S \mid 1A$,
$A \rightarrow 0B \mid 1S \mid \epsilon$,
$B \rightarrow 0A \mid 1B$.

c) $S \rightarrow 0S \mid 1A$,
$A \rightarrow 0B \mid 1S$,
$B \rightarrow 0A \mid 1B \mid \epsilon$.

d) $S \rightarrow 0S \mid 1A$,
$A \rightarrow 0B \mid 1S$,
$B \rightarrow 0A \mid 1B$.

e) $S \rightarrow 0S \mid 1A \mid \epsilon$,
$A \rightarrow 0B \mid 1S \mid \epsilon$,
$B \rightarrow 0A \mid 1B \mid \epsilon$.

Seção 2.3

Expressões regulares

Diálogo aberto

O programa para detecção de padrões para localizar temperaturas negativas foi um sucesso e por isso você se tornou responsável pelos dados internacionais. Você deve enviar uma solução para outros países onde sua instituição atua. Cada país possui computadores com hardware diferente, mas todos rodam o sistema operacional Linux e você deve agora desenvolver uma solução portável usando os recursos deste sistema. Para isso você irá especificar a mesma linguagem especificada na seção anterior, desta vez escrevendo uma expressão regular que descreva a linguagem.

Nesta seção, você irá ver expressões regulares e sua relação com autômatos finitos e gramáticos regulares. As expressões regulares são usadas em diversas ferramentas do sistema operacional Linux, assim você planeja usá-las para desenvolver uma nova solução que poderá ser adotada internacionalmente em sua instituição. Esta é uma excelente oportunidade para colocar em prática os conhecimentos desta seção e se destacar ainda mais em seu trabalho.

Bons estudos!

Não pode faltar

Uma Expressão Regular (ER) é uma forma compacta de descrever linguagens regulares. Trata-se de uma definição que faz uso do fato de que a concatenação, união e o fecho de Kleene de linguagens regulares são regulares. Teremos então estas operações representadas pela justaposição (no caso da concatenação), o "+" (no caso da união) e o "*" (no caso do fecho de Kleene). Além disso, devemos também considerar as linguagens regulares mais simples. São elas: a linguagem vazia, a linguagem com a cadeia vazia e as linguagens com somente um símbolo.

Assimile

Uma expressão regular sobre o alfabeto $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ é de uma das seguintes formas:

• $\varnothing$; ϵ; σ_i, onde $\sigma_i \in \Sigma$.

• Se e é uma expressão regular, então e^* também é uma expressão regular. Se e_1 e e_2 são expressões regulares, então $e_1 e_2$ e $e_1 + e_2$ são expressões regulares. Se e é uma expressão regular, então (e) é uma expressão regular.

• Nada mais é uma expressão regular.

Os exemplos que seguem mostram para algumas expressões regulares as respectivas linguagens descritas:

• a representa a linguagem $\{a\}$, ou seja, a linguagem que só tem o símbolo a.

• $\varnothing$ representa a linguagem vazia, sem cadeias.

• $a + b + c$ representa a linguagem $\{a, b, c\}$.

• ϵ representa a linguagem $\{\epsilon\}$.

• a^* representa a linguagem $\{a\}^*$, ou seja, todos as cadeias formadas somente com o símbolo a.

• $(a + b)^*$ representa as cadeias no conjunto $\{a, b\}^*$, ou seja, todas as cadeias formadas de a's e b's.

• $a^* + b^*$ representa a linguagem das cadeias formadas só com a's ou só com b's.

Podemos verificar que algumas expressões regulares podem ser equivalentes a outras, no sentido de especificarem a mesma linguagem formal. Por exemplo, $(a^* + b^*)^*$ e $(a + b)^*$ especificam a mesma linguagem $\{a, b\}^*$.

Assimile

Existe uma forma recursiva de definir a linguagem que uma expressão regular descreve. Se e é uma expressão regular vamos usar $[e]$ para denotar a linguagem que e descreve.

Seja e uma expressão regular, define-se $[e]$ por casos, de acordo com as equações a seguir:

- $[\sigma_i] = \{\sigma_i\}$, com $\sigma_i \in \Sigma$.
- $[\varnothing] = \varnothing$.
- $[\epsilon] = \{\epsilon\}$.
- $[e_1 + e_2] = [e_1] \cup [e_2]$.
- $[e_1 e_2] = [e_1][e_2]$.
- $[e^*] = [e]^*$.

A definição desenvolvida na caixa *Assimile* permite que você atribua um conjunto de cadeias, e somente um, a qualquer expressão regular. Dizemos que $[e]$ é a linguagem formal descrita, especificada, pela expressão regular e.

Algumas questões que surgem naturalmente se toda linguagem regular é especificável por alguma expressão regular e vice-versa.

Na própria definição de $[\ ]$ responde, juntamente às propriedades de fechamento de linguagens regulares, à segunda questão. Vejamos: os três primeiros itens da definição, ou seja, o significado pretendido das expressões σ_i, $\varnothing$ e ϵ são linguagens regulares. Os outros itens utilizam operações sobre linguagens regulares que resultam em uma linguagem regular. Relembre que a concatenação de linguagens regulares, o fecho de Kleene de uma linguagem regular e a união de linguagens regulares são linguagens regulares. Portanto, toda expressão regular define uma linguagem regular. Este é um fato importante.

Assimile

Se e é uma expressão regular, então a linguagem $[e]$ é regular.

Neste ponto talvez tenha ficado claro que é muito simples escrever uma ER, entretanto, é difícil, até mesmo para um computador, verificar se duas ERs descrevem a mesma linguagem.

Pesquise mais

Considere o seguinte problema: dadas expressões regulares E_1 e E_2, elas são equivalentes? (descrevem a mesma linguagem).

Esse é um problema muito difícil, pesquise na internet as soluções existentes e porquê ele é considerado um problema difícil. Comece acessando o site disponível em: <http://bit.ly/2rVu3yr>. Acesso em: 12 jul. 2017.

O fato de as expressões regulares representarem linguagens regulares é importante, mas ele se torna mais útil se soubermos construir um reconhecedor (um AFD) a partir da expressão. Lembre-se das gramáticas, pois servem de guia para a construção de AFDs que reconhecem a linguagem gerada por elas. É bom descrever linguagens regulares de forma compacta, mas é melhor ainda se desta descrição pudermos sistematicamente construir um programa que verifique se as cadeias pertencem ou não à linguagem descrita pela expressão regular. De fato, a definição de $[\,]$ anterior pode ser modificada para em vez de associar um conjunto à cada expressão regular, ela associa um autômato que reconhece a linguagem descrita pela expressão. Faremos isso através do uso de autômatos com transição ϵ.

Um autômato finito com transição ϵ, $AF\epsilon$, é um AFND onde a função de transição inclui a leitura da cadeia vazia. Na realidade, é uma forma de realizar a transição de estados sem que sejam lidos símbolos da entrada. A função de transição δ de um AF_ϵ é tal que, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \wp(Q)$. O efeito de uma transição ϵ é não determinístico. A transição $\delta(q_1, \epsilon) = \{q_2\}$ indica que na leitura de ϵ, estando o autômato no estado q_1, este muda para o estado q_2.

Desde que a leitura de ϵ é opcional, pois a cadeia vazia é subcadeia de qualquer outra cadeia, também pode-se considerar que o autômato permanece no estado q_1. A extensão da função δ para cadeias de símbolos, $\hat{\delta}$, é então definida como:

- $\hat{\delta}(q,\epsilon) = \delta(q,\epsilon)$
- $\hat{\delta}(q,\sigma) = \delta(q,\sigma) \cup \delta(q,\epsilon)$
- $\hat{\delta}(q,\sigma\omega) = \cup_{q \in \delta(q,\sigma) \cup \delta(q,\epsilon)} \hat{\delta}(q,\omega)$

Define-se a linguagem aceita por um AF_ϵ $\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta \rangle$ como $\{\omega \in \Sigma^* / \hat{\delta}(q,\omega) \cap F\}$, onde F é o conjunto de estados finais de $\mathcal{A}$.

Um ponto importante é observar que todo AF_ϵ aceita uma linguagem regular. Isto pode ser argumentado ao verificarmos que as transições não são essenciais. Considere um AF_ϵ $\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta \rangle$ com $\delta(q,\epsilon) = S$. Podemos provar que a linguagem aceita por este AF_ϵ é regular, eliminando a transição ϵ. A transição ϵ significa que o autômato uma vez no estado q pode transitar a qualquer um dos estados $s \in S$. Pode-se eliminar esta transição ϵ considerando para cada símbolo $\sigma \in \Sigma$ o conjunto $\{q' \in Q / q \in \delta(q',\sigma)\}$ dos estados q' que transitam para q via leitura de σ. Se definimos uma nova função de transição δ_{nova}, tal que, $\delta_{nova}(q',\sigma) = \delta(q',\sigma) \cup \delta(q,\epsilon)$. O autômato $\mathcal{B} = \langle Q, \Sigma, q_0, F, \delta_{nova} \rangle$ é um autômato não determinístico sem transições ϵ que aceita a mesma linguagem que $\mathcal{A}$.

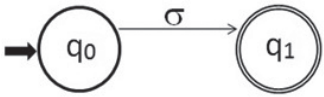
Assimile

Todo autômato finito com transições ϵ reconhece uma linguagem regular.

Pode-se agora formalizar a associação de uma expressão regular ao autômato finito que reconhece a linguagem descrita por ela. Seja e uma expressão regular, define-se o autômato $\mathcal{A}_e$, $[e]$, por casos, de acordo com as equações a seguir:

- Se $e = \sigma_i$ então $\mathcal{A}_e = \mathcal{A}_{\sigma i}$, exibido na Figura 2.12, com $\sigma_i \in \Sigma$;

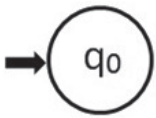
Figura 2.12 | $\mathcal{A}_e = \mathcal{A}_{\sigma i}$



Fonte: elaborada pelo autor.

• Se $e = \varnothing$ então $\mathcal{A}_e = \mathcal{A}_{\varnothing}$, exibido na Figura 2.13;

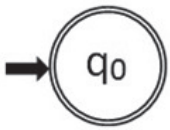
Figura 2.13 | $\mathcal{A}_e = \mathcal{A}_{\varnothing}$



Fonte: elaborada pelo autor.

• Se $e = \epsilon$ então $\mathcal{A}_e = \mathcal{A}_{\epsilon}$, exibido na Figura 2.14;

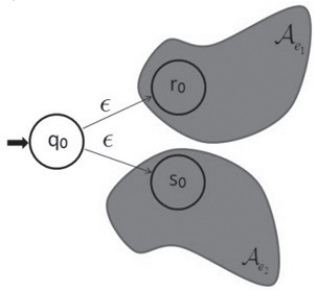
Figura 2.14 | $\mathcal{A}_e = \mathcal{A}_{\epsilon}$



Fonte: elaborada pelo autor.

• Se $e = e_1 + e_2$ então $\mathcal{A}_e = \mathcal{A}_{e_1} \cup \mathcal{A}_{e_2}$, exibido na Figura 2.15, onde é criado um novo estado inicial, que tem uma transição ϵ indo para cada um dos estados iniciais de $\mathcal{A}_{e_1}$ e $\mathcal{A}_{e_2}$.

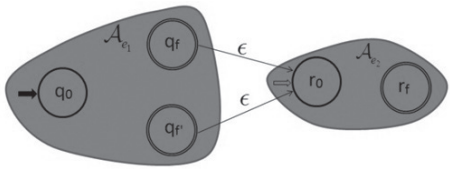
Figura 2.15 | $\mathcal{A}_e = \mathcal{A}_{e_1} \cup \mathcal{A}_{e_2}$



Fonte: elaborada pelo autor.

• Se $e = e_1 e_2$ então $\mathcal{A}_e = \mathcal{A}_{e_1} \mathcal{A}_{e_2}$, exibido na Figura 2.16, onde cada estado final de $\mathcal{A}_{e_1}$ tem uma transição ϵ indo para o estado inicial de $\mathcal{A}_{e_2}$;

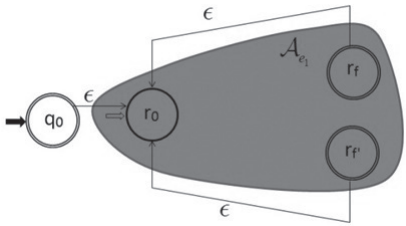
Figura 2.16 | $\mathcal{A}_e = \mathcal{A}_{e_1} \mathcal{A}_{e_2}$



Fonte: elaborada pelo autor.

• Se $e = e_1^*$ então $\mathcal{A}_e = \mathcal{A}_{e_1}^*$, exibido na Figura 2.17, onde é criado um novo estado inicial (e final) q_0, com uma transição ϵ indo para o estado inicial de $\mathcal{A}_{e_1}$ e são incluídas transições $\mathcal{A}_{e_1}$ dos estados finais de para o inicial de $\mathcal{A}_{e_1}$.

Figura 2.17 | $\mathcal{A}_e = \mathcal{A}_{e_1}^*$



Fonte: elaborada pelo autor.

Para qualquer expressão regular e, o autômato AF_ϵ $\mathcal{A}_e$ reconhece $[e]$. Eliminando-se transições ϵ, como explicado acima, obtém-se um AFND que reconhece $[e]$. Você pode encontrar descrição mais detalhada da construção acima em Menezes (2000) e uma versão sem usar o conceito de AF_ϵ em Garcia (2017).

Concluímos, então, que existem três formas de se formalizar uma linguagem regular. Via gramáticas regulares, via autômatos finitos (determinísticos ou não) e via expressões regulares. Além disso, para cada formalismo mostramos como efetivamente obter outra especificação formal em qualquer um dos outros dois formalismos. Ou seja, dada uma gramática regular, mostramos como construir uma expressão regular

e um autômato, que respectivamente descrevem e reconhecem a linguagem gerada por esta. O mesmo se dá a partir de qualquer expressão regular. Vimos como, a partir de uma expressão regular, obter um AFND equivalente. O processo inverso também é possível. Já sabemos que de um AFND podemos obter uma gramática regular equivalente. A seguir vamos mostrar como, a partir desta gramática, obter uma expressão regular equivalente.

Vamos começar analisando a seguinte gramática regular $G_1 S \to aS \mid b$

Vamos representar como $[S]$ o conjunto de todas as sentenças geradas pelo símbolo S, ou seja $[S] = \{w \in \{0,1\}^* \mid S \Rightarrow^* w\}$. Neste caso, como S é o símbolo inicial da gramática $[S] = L(G_1)$, ou seja, os conjuntos são iguais. Como consequência das regras da gramática G_1, podemos escrever a seguinte igualdade de conjuntos: $[S] = a[S] \cup b$. Ou usando o símbolo de expressões regulares para a união: $[S] = a[S] + b$. A solução desta equação é simplesmente: $[S] = a^* b$.

Assimile

A solução anterior é uma ilustração do Lema de Arden. Se conhecemos as linguagens L_1 e L_2 e definimos a linguagem L pela igualdade $L = L_1 L + L_2$, então a menor linguagem que atende à igualdade é $L = L_1^* L_2$, e esta solução é única quando $\epsilon \notin L_1$.

No caso de gramática com mais de uma variável (mais de um símbolo não terminal), podemos encontrar a expressão regular de forma parecida com a resolução de um sistema linear. Por exemplo, considere a gramática G_2:

$S \to bS \mid aA$,

$A \to bA \mid aS \mid \epsilon$.

Podemos reescrever a gramática como a seguinte igualdade entre conjuntos:

$[S] = b[S] + a[A]$,

$[A] = b[A] + a[S] + \epsilon$.

A segunda igualdade é o mesmo que $[A] = (a[S] + \epsilon) + b[A]$.

Portanto, pelo Lema de Arden, pode ser reescrita sem recorrência como $[A] = b^*(a[S] + \epsilon)$.

Esta forma para $[A]$ pode ser substituída na primeira equação $[S] = b[S] + ab^*(a[S] + \epsilon)$.

Isso pode ser rearranjado como $[S] = (b + ab^*a)[S] + ab^*$. Aplicando mais uma vez o Lema de Arden, chegamos à solução final $[S] = (b + ab^*a)^* ab^*$. Logo, a ER equivalente à gramática G_2 é $(b + ab^*a)^* ab^*$.

Exemplificando

Vamos exemplificar o método com uma gramática mais difícil, com três símbolos não terminais. Considere a seguinte gramática:

$$S \to 0A \mid 1S \; A \to 0B \mid 1S \; B \to 0S \mid 1A \mid \epsilon$$

Para obtermos a solução primeiro obtemos as igualdades:

$$[S] = 0[A] + 1[S]$$

$$[A] = 0[B] + 1[S]$$

$$[B] = 0[S] + 1[A] + \epsilon$$

Depois substituímos a 3ª equação na segunda:

$$[A] = 0(0[S] + 1[A] + \epsilon) + 1[S]$$

Colocamos os termos em $[A]$ separados:

$$[A] = 01[A] + 00[S] + 0 + 1[S]$$

Aplicamos o Lema de Arden: $[A] = (01)^*(00[S] + 0 + 1[S])$

Substituímos $[A]$ na primeira equação:

$$[S] = 0((01)^*(00[S] + 0 + 1[S])) + 1[S]$$

$$[S] = 0(01)^*(00[S] + 0 + 1[S]) + 1[S]$$

$$[S] = 0(01)^*00[S] + 0(01)^*0 + 0(01)^*1[S] + 1[S]$$

$$[S] = (0(01)^*00 + 0(01)^*1 + 1)[S] + 0(01)^*0$$

$$[S] = (0(01)^*00 + 0(01)^*1 + 1)^*0(01)^*0$$

Logo a expressão regular equivalente é:

$$[S] = (0(01)^*00 + 0(01)^*1 + 1)^*0(01)^*0$$

Expressões regulares POSIX

No sistema operacional Unix (e similares) diversos comandos utilizam expressões regulares, como : grep, sed e awk. Sua especificação foi normatizada em uma parte do padrão POSIX, mantido pela IEEE. Muitas linguagens de programação também passaram a aceitar linguagens regulares, por exemplo, Java, JavaScript, Perl, C#, R, Lua, entre outras, seguindo diferentes variantes de notação. É importante ressaltar que tanto as expressões regulares no sistema Linux, como as disponíveis nas linguagens supracitadas, apesar de manterem o nome histórico de 'expressões regulares', podem atualmente reconhecer linguagens não regulares, isto é, linguagens que não podem ser especificadas por uma gramática regular. O leitor interessado pode encontrar uma introdução a expressões regulares padrão POSIX em Hopcroft; Motwani; Ullman (2002) e um tratamento detalhado em Goyvaerts; Levitahn (2011). Nesta obra mostraremos algumas poucas funcionalidades do comando grep do Unix, ele pode ser chamado da forma:

grep <expressão_regular_procurada><arquivo>

Cada vez que o comando encontra a expressão procurada, ele imprime a linha do arquivo na qual a expressão se encontra. Em geral, cada caractere representa a linguagem apenas com aquele caractere, a concatenação é simplesmente a justaposição dos símbolos, a união é feita usando-se o símbolo '|', enquanto que o símbolo '*' representa a repetição 0 ou mais vezes. O comando possui algumas abreviações úteis:

[[:alpha:]] representa qualquer caractere alfabético (letra)

[[:alnum:]] representa qualquer caractere alfanumérico (letra ou dígito)

[0-9] representa qualquer caractere numérico (entre 0 e 9)

. representa qualquer caractere

Por exemplo, considere o arquivo 'lista_de_nomes.txt', com sete linhas, cujo conteúdo é o nome seguido de um separador, seguido do ano do nascimento:

Alice Siqueira Lima; 1991
Diogo Garcia Palmeira; 1989
José Silva Marques; 1987
Mariana Cavalcanti Mangabeira; 1988
Oscar Souza Silveira; 1993
Raul Nogueira dos Santos; 1984
Roberta Miranda Oliveira; 1983

Ao rodarmos o comando:
grep eiralista_de_nomes.txt
São impressas toas as linhas que possuem o string 'eira':
Alice Siqueira Lima; 1991
Diogo Garcia Palmeira; 1989
Mariana Cavalcanti Mangabeira; 1988
Oscar Souza Silveira; 1993
Raul Nogueira dos Santos; 1984
Roberta Miranda Oliveira; 1983

Ao rodarmos o comando:
grep eira..[0-9]lista_de_nomes.txt
São impressas todas as linhas que possuem o string 'eira', seguido de dois caracteres quaisquer (em particular ';' e espaço em branco), seguido de um dígito entre 0 e 9, resultando em:
Diogo Garcia Palmeira; 1989
Mariana Cavalcanti Mangabeira; 1988
Oscar Souza Silveira; 1993
Roberta Miranda Oliveira; 1983

Sem medo de errar

Na Seção 2.1 você fez a gramática que especifica as constantes numéricas que representam temperaturas negativas. Fazer isso com expressões regulares é muito mais simples. Em particular, mostramos a solução usando expressões regulares POSIX, o que facilita sua utilização no sistema Linux. A solução com expressões regulares, portável para a plataforma Linux é dada a seguir:

A parte inteira negativa pode ser encontrada pela expressão-[0-9][0-9]*

A parte decimal, que é opcional, pode ser casada pela expressão(.[0-9]*)?

1.1 Finalmente, para encontrarmos o padrão desejado, basta concatenarmos as expressões, obtendo-[0-9][0-9]*(.[0-9]*)?

1.2 Para usar a expressão em um comando Linux será necessário fazer o 'escape' de alguns caracteres.

Avançando na prática

Verificar equivalência de expressões regulares

Descrição da situação-problema

Você já sabe que toda linguagem reconhecida por autômato finito, determinístico ou não, pode ser expressa por expressões regulares. Por exemplo, a linguagem das cadeias com a's e b's e um número par de a's é expressa por $\epsilon + (b^*ab^*ab^*)^*$. Outras expressões regulares que descrevem a mesma linguagem são $(b^*ab^*a)^*b^*$, $((b^*ab^*)(b^*ab^*))^*, b^*(ab^*a)^*b^*$ etc. Sabemos que a quantidade de expressões regulares que descrevem uma linguagem regular é infinita, pois se E é uma expressão regular que descreve alguma linguagem, então E+E, E+E+E etc. são expressões regulares que a descrevem também. Dado que a quantidade de expressões que descreve qualquer linguagem regular é ilimitada, pode ser útil saber se duas expressões regulares são equivalentes, ou seja, se descrevem a mesma linguagem.

Descreva um algoritmo que recebe como entrada duas expressões regulares e retorne como resultado a informação de que são ou não expressões regulares equivalentes. Por exemplo, para as expressões regulares mencionadas anteriormente, seu algoritmo deve retornar "Equivalentes".

Dica: Projete seu algoritmo para determinar se dois AFD reconhecem a mesma linguagem ou não.

Resolução da situação-problema

Podemos, a partir das expressões regulares E_1 e E_2, que queremos verificar se são equivalentes gerar AF_ϵs A_1 e B_1. A partir de A_1 e B_1, sabemos obter os AFDs A_2 e B_2, equivalentes, respectivamente a A_1 e B_1, portanto equivalentes a E_1 e E_2.

Para resolvermos o problema, basta, portanto, criarmos um algoritmo para saber se os AFDs A_2 e B_2 são equivalentes. Para um AFD, sabemos construir o AFD complemento e o AFD produto. Portanto sabemos construirmos o AFD C que reconhece a linguagem $(T(A_2) \cap \overline{T(B_2)}) \cup T(B_2) \cap \overline{T(A_2)})$.

Dado este AFD C, para saber se $T(C)$ é vazia ou não, basta testar se qualquer cadeia de tamanho até a quantidade de estados de C será aceita/rejeitada.

O algoritmo descrito acima recebe expressões regulares e as transforma em AFDs para então verificar se a linguagem descrita pelos últimos é equivalente, devemos ter uma ideia da eficiência deste procedimento. Vimos que a transformação de uma expressão regular em um AF_ϵ é relativamente eficiente. Cada subexpressão dá origem a um subautômato do AF_ϵ resultante. Este subautômato tem um tamanho similar à subexpressão que lhe deu origem. No entanto, a transformação de em AFD não é nada eficiente. Se n é a quantidade de estados do autômato AF_ϵ, a quantidade de estados do AFD associado será, no pior caso, 2^n. Se nosso alfabeto possui 2 símbolos, o algoritmo de equivalência de expressões regulares da proposição acima terá que testar $2^{2^n} - 1$ cadeias, mesmo sendo em quantidade finita, este procedimento pode ser inviável. Para pequenos valores de n, como n=8, testar $2^{256} - 1$ cadeias leva tempo maior do que a idade do universo.

Faça valer a pena

1. Seja a linguagem $L = \{w \in \{a,b\}^* \mid w$ possui quantidade par de caracteres $a\}$. Assinale a expressão regular que descreve a linguagem L:

a) $((b^*ab)(b^*ab^*))^*$.

b) $((b^*ab^*)(b^*a^*b^*))^*$.

c) $((ab^*)(b^*ab^*))^*$.

d) $((b^*ab^*)(b^*a))^*$.

e) $b^*(ab^*a)^*b^*$.

2. Considere o AFD definido pela seguinte tabela:

	0	1
$\rightarrow {}^*q_0$	q_1	q_1
q_1	q_0	q_1

onde o estado inicial é marcado com a seta "$\rightarrow$" e os estados finais marcados com "*".

Assinale a expressão regular que representa a linguagem aceita por este AFD:

a) $(0^*1^*0^*1^*)^*0$.

b) $(0^*1^*0^*1^*0)^*$.

c) $((0+1)^*1^*0)^*$.

d) $((0+1)1^*0)^*$.

e) $((0+1)^*1^*0^*)^*$.

3. Seja L_1 a linguagem representada pela expressão regular $((0+1)1^*0)^*$. Assinale a expressão regular que representa a linguagem $\overline{L_1}$, o complemento de L_1:

a) $((0+1)(1^*0))^*(0+1)1^*$.

b) 000.

c) $((0+1)(10))^*(0+1)1^*$.

d) $((0+1)(10^*))^*(0+1)1^*$.

e) $((0+1)(10^*))^*(0+1)1$.

Referências

AHO, A. et al. **Compilers**: Principles, Techniques, and Tools. Nova York: Addison-Wesley, 2006.

GARCIA, A. **Linguagens regulares e livres de contexto**. 1. ed. Rio de Janeiro: Edição do Autor (eBook Kindle), 2017.

GOYVAERTS, J.; LEVITHAN, S. **Expressões regulares cookbook**. 1. ed. São Paulo: Novatec, 2011.

HOPCROFT, J.; ULLMAN, J. **Formal Languages and Their Relation to Automata**. Nova York: Addison-Wesley, 1969.

HOPCROFT, J.; MOTWANI, R.; ULLMAN, J. **Introdução à teoria de autômatos, linguagens e computação**. 1. ed. Boston: Elsevier, 2002.

MENEZES, P. B. **Linguagens formais e autômatos**. Porto Alegre: Sagra Luzzatto, 2000.

NAVEH, J. **Early History of the Alphabet**. Skokie: Varda Books, 2012.

Linguagens e gramáticas livres do contexto e autômatos com pilha

Convite ao estudo

Nesta unidade, vamos estudar gramáticas livres de contexto, linguagens livres de contexto e o autômato com pilha. O autômato com pilha é um reconhecedor para linguagens livres de contexto e tem um papel semelhante àquele que o AFND tem para linguagens regulares.

Devido ao seu sucesso em solucionar o problema de localização de padrões (no caso, temperaturas negativas) em arquivos ".csv", você ficou responsável pela elaboração da solução a nível mundial. Você foi informado que a maioria dos países usa a plataforma Linux e alguns poucos, como o Brasil, usam Windows. Por este motivo você desenvolveu uma nova solução usando expressões regulares. Infelizmente, ao apresentar sua solução, você foi informado que em muitos países os dados não são armazenados em formato ".csv", mas sim em formato ".xml". As gramáticas regulares não podem ajudá-lo neste problema, no entanto, as gramáticas livres de contexto podem, já que possuem maior poder de expressão, e serão abordadas nesta unidade.

Na Seção 3.1, vamos estudar gramáticas livres de contexto árvores de derivações e gramáticas ambíguas. Na Seção 3.2, vamos estudar linguagens livres de contexto e suas propriedades. Finalmente, na Seção 3.3, vamos estudar os autômatos com pilhas, reconhecedores de linguagens livres de contexto. Todo este conhecimento é útil para os novos desafios de internacionalização de sua solução de reconhecimento de padrões, portanto continue seus estudos com afinco.

Seção 3.1

Gramáticas livres de contexto

Diálogo aberto

Recordamos que ao apresentar sua proposta de internacionalização para sua solução de busca de padrões, você foi surpreendido por um novo requisito: em muitos países os dados não são armazenados em formato ".csv", mas sim em formato ".xml".

Os arquivos ".xml" apresentam dois níveis de sintaxe. No primeiro nível o documento é dito "bem-formado". No segundo nível, dependendo de uma especificação de hierarquia de Tags, chamada DTD, o documento é dito "válido".

O tag xml usado para identificar um local é <local>... </local>. Pode haver locais dentro de locais e cada local pode ter um tag <temp> ...</temp> com sua temperatura média. Não há nenhum padrão sobre quantas vezes um tag <local> é subdividido. Um exemplo é:

<local id="USA">
<temp>12</temp>
<local id="NJ">
<temp>-23</temp>
</local>
<local id="CA">
<temp>18</temp>
<local id="Los Angeles">
<temp>22</temp>
</local>
</local>
</local>

Neste momento, você deve modelar o problema criando um relatório que inclui a especificação de uma Gramática Livre de Contexto que gere este formato de arquivo explicando como ela será usada posteriormente. Esta é a chance do seu trabalho ser reconhecido internacionalmente em sua empresa.

Bons estudos!

Sem medo de errar

Ao final da Unidade 1, definimos alguns tipos de gramática. A mais simples foi a gramática regular, estudada em profundidade na Unidade 2. Na unidade que se inicia, estudaremos um segundo tipo de gramática, as gramáticas livres de contexto. Na Seção 1.3, vimos a definição de gramáticas livres de contexto (GLC), que são aquelas nas quais todas as regras têm exatamente um símbolo não terminal (e nenhum outro símbolo) do lado esquerdo (GARCIA, 2017).

Assimile

Uma gramática G = (V,T,P,S) é livre de contexto (GLC) se, e somente se, todas as regras são da forma $A \to w$ com $A \in V$ e $w \in (V \cup T)^*$.

Um exemplo clássico de GLC é a gramática que gera cadeias com os caracteres "(" e ")" de forma que cada abertura de parênteses corresponde a um fechamento de parênteses posterior.

Exemplificando

Considere a linguagem que possui os parênteses corretamente balanceados. L_{par} = $\{\epsilon, (), (()), ()(), ((())), (())(), ()(()), (())(), ()(()), \ldots\}$

Esta linguagem é gerada pela gramática a seguir:

$S \to \epsilon$

$S \to SS$

$S \to (S)$

O leitor deve perceber que, pela definição apresentada, toda gramática regular também é uma gramática livre de contexto. Ao final da Unidade 1, definimos que uma linguagem é regular se, e somente se, existe uma gramática regular que a gere. Da mesma forma, uma linguagem é livre de contexto (LLC) se, e somente se, existe uma gramática livre de contexto que a gere. Portanto, toda linguagem regular L possui uma gramática regular que a gera, gramática esta

que também é livre de contexto, portanto, por definição a linguagem L também é livre de contexto.

Entretanto, a recíproca não é verdadeira. Considere a linguagem $L_p = \{ab, aabb, aaabbb, \ldots\}$. Esta linguagem pode ser facilmente gerada pela seguinte gramática livre de contexto G_p:

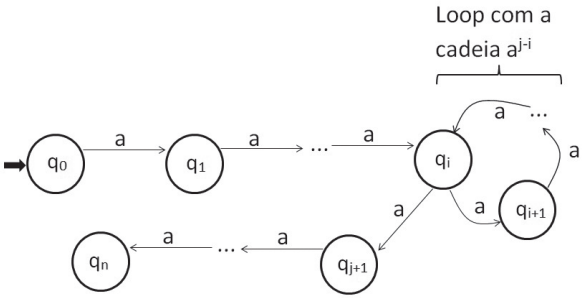
$$S \rightarrow aSb \mid ab$$

Recordamos nossa convenção de que as variáveis são escritas em letras maiúsculas, sendo os demais símbolos terminais, e que o símbolo inicial é a primeira variável que aparece ao escrevermos a gramática. A gramática G_p gera facilmente a cadeia $aaabbb$, através da derivação:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

Para constatarmos que L_p não é uma linguagem regular necessitamos de um raciocínio sofisticado. Apresentamos aqui o raciocínio para completar a explicação. Não é esperado que o aluno saiba reproduzi-lo. Suponhamos que L_p seja regular. Neste caso haveria um AFD $M = (Q, \Sigma, \delta, q_0, F)$, com uma quantidade finita de estados, a qual chamaremos n, que reconheceria L_p. Portanto, entre os $n+1$ estados $q_0 = \hat{\delta}(q_0, \epsilon)$, $q_1 = \hat{\delta}(q_0, a)$, $q_2 = \hat{\delta}(q_0, aa)$, $q_n = \hat{\delta}(q_0, a^n)$ haveria dois deles necessariamente iguais, por exemplo, $\hat{\delta}(q_0, a^i)$ e $\hat{\delta}(q_0, a^j)$, com $i < j$. Entretanto, nesse caso os estados $\hat{\delta}(\hat{\delta}(q_0, a^i), b^i)$ e $\hat{\delta}(\hat{\delta}(q_0, a^j), b^i)$ seriam iguais, mas o primeiro é final e o segundo não, o que é impossível, logo nossa hipótese está errada. Portanto L_p não é regular. Pode-se observar que existe um loop entre o q_i e q_j (que são o mesmo estado), esta situação é ilustrada na Figura 3.1.

Figura 3.1 | Repetição de estados em um suposto AFD



Fonte: elaborada pelos autores.

Esta explicação demonstra o maior poder de expressividade das gramáticas livres de contexto. Normalmente usamos gramáticas livres de contexto, ou algum formalismo equivalente, para especificar a sintaxe de uma linguagem de programação. Por exemplo, a especificação sintática da linguagem Java (GOSLING et al., 2015) é feita usando uma gramática livre de contexto (com uma notação um pouco diferente). A gramática livre de contexto G_{exp}, a seguir, gera as expressões aritméticas formadas com as operações soma e multiplicação e o número '1'.

$S \to S + S$,
$S \to S \times S$,
$S \to (S)$,
$S \to 1$.

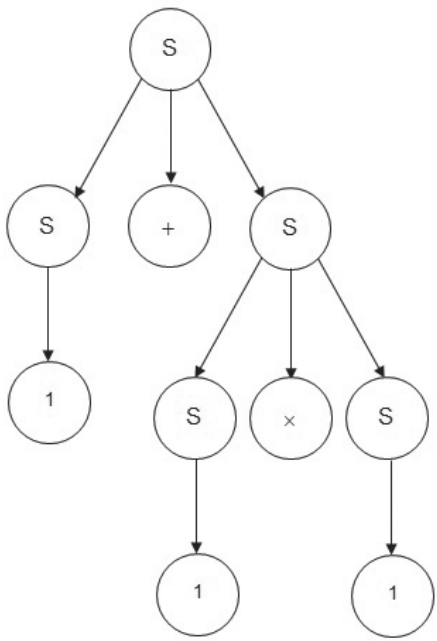
Uma mesma cadeia pode ter diversas derivações. Por exemplo, a cadeia $1+1\times 1$ possui, entre outras, as três seguintes derivações:

$S \Rightarrow S+S \Rightarrow 1+S \Rightarrow 1+S\times S \Rightarrow 1+1\times S \Rightarrow 1+1\times 1$.
$S \Rightarrow S+S \Rightarrow S+S\times S \Rightarrow S+S\times 1 \Rightarrow S+1\times 1 \Rightarrow 1+1\times 1$.
$S \Rightarrow S+S \Rightarrow S+S\times S \Rightarrow 1+S\times S \Rightarrow 1+1\times S \Rightarrow 1+1\times 1$.

Observamos que na primeira derivação substituímos sempre a variável mais à esquerda na forma sentencial (a chamamos de derivação mais à esquerda), na segunda derivação substituímos sempre a variável mais à direita (a chamamos de derivação mais à direita), enquanto que na terceira derivação não há uma ordem preferencial de substituição (essa derivação não tem um nome especial).

De uma forma geral, chamamos de derivação mais à esquerda (DME) àquela na qual substituímos sempre a variável mais à esquerda na forma sentencial (MENEZES, 2000). Analogamente chamamos de derivação mais à direita (DMD) àquela na qual substituímos sempre a variável mais à direita na forma sentencial. Podemos observar ainda que as três derivações acima correspondem à estrutura da Figura 3.2, a qual representa estas derivações na forma de árvore de derivação.

Figura 3.2 | Árvore de derivação para a expressão 1+1×1



Fonte: elaborada pelos autores.

Observe que as três derivações apresentadas anteriormente correspondem a percorrer a árvore apresentada na Figura 3.2 em diferentes ordens. Enquanto que as derivações apresentadas a seguirapresentam a estrutura da árvore da Figura 3.3.

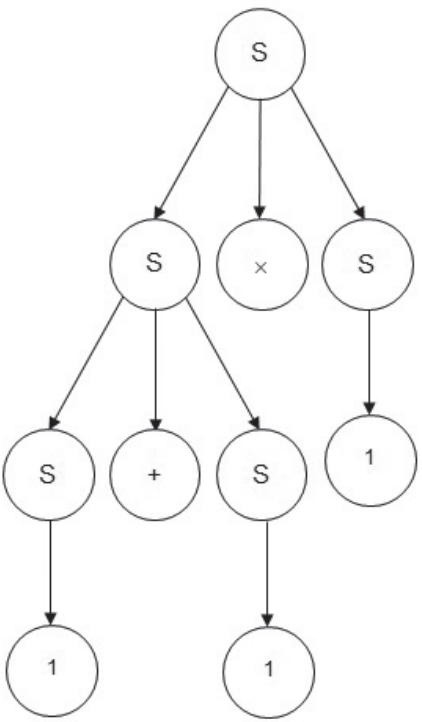
$S \Rightarrow S \times S \Rightarrow S + S \times S \Rightarrow 1 + S \times S \Rightarrow 1 + 1 \times S \Rightarrow 1 + 1 \times 1$. (Esta é uma DME)

$S \Rightarrow S \times S \Rightarrow S \times 1 \Rightarrow S + S \times 1 \Rightarrow S + 1 \times 1 \Rightarrow 1 + 1 \times 1$. (Esta é uma DMD)

$$S \Rightarrow S \times S \Rightarrow S + S \times S \Rightarrow 1 + S \times S \Rightarrow 1 + 1 \times S \Rightarrow 1 + 1 \times 1.$$

Uma árvore de derivação quando é lida de cima para baixo e da esquerda para a direita é gerada a partir de uma DME. Portanto, uma árvore de derivação está associada a uma, e somente uma, derivação mais à esquerda. O mesmo se dá para derivações mais à direita, a saber, uma árvore de derivação está associada a uma, e somente uma, DMD. Entretanto, uma mesma árvore de derivação pode ter muitas derivações associadas e não há limite para este número.

Figura 3.3 | Árvore de derivação alternativa para a expressão 1+1×1



Fonte: elaborada pelos autores.

As Figuras 3.2 e 3.3 apresentam duas árvores de derivação distintas para uma mesma cadeia, a saber, $1+1\times1$. Neste caso, dizemos que a gramática é ambígua.

Assimile

Uma gramática livre de contexto G é ambígua se existe uma cadeia w que possui mais de uma árvore de derivação de acordo com G.

Uma vez que existe uma correspondência um para um entre árvores de derivação e DME, podemos dizer, de forma equivalente, que uma GLC G é ambígua se existe uma cadeia w que possui mais de uma DME de acordo com G. O mesmo se dá com DMDs.

Vimos que a gramática G_{exp} é ambígua. A seguir veremos que existe uma GLC que gera a mesma linguagem, mas não é ambígua, entretanto não existe um algoritmo para obter uma gramática equivalente não ambígua que funcione para qualquer GLC. A gramática livre de contexto a seguir também gera as expressões aritméticas formadas com as operações soma e multiplicação e o número '1', mas não é ambígua.

$E \to E + T \mid T,$

$T \to T \times F \mid F,$

$F \to (E),$

$F \to 1.$

Uma derivação da sentença $1+1\times 1$ é:

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow 1+T \Rightarrow 1+T\times F \Rightarrow 1+F\times F \Rightarrow 1+1\times F \Rightarrow 1+1\times 1$$

Dada uma GLC ambígua, nem sempre é possível construir uma gramática não ambígua equivalente. Considere as linguagens $L_1 = \{a^n b^m c^m d^n \mid n,m > 0\}$ e $L_2 = \{a^n b^n c^m d^m \mid n,m > 0\}$. A linguagem $L = L_1 \cup L_2$. L é livre de contexto, mas para todas as gramáticas livres de contexto que a geram, as cadeias em $L_1 \cap L_2$ possuem mais do que uma árvore de derivação (HOPCROFT; MOTWANI; ULLMAN, 2002). Ou seja, todas as gramáticas que geram L são ambíguas. Neste caso, dizemos que a gramática L é inerentemente ambígua.

Pesquise mais

Por ser mais simples, a especificação de algumas linguagens de programação pode preferir apresentar a sintaxe do comando if-then-else de forma ambígua (mesmo que neste caso seja possível fazê-lo de forma não ambígua). Por exemplo, na especificação de C, temos as variáveis C (Comando), E (Expressão) e D (comando de Decisão), entre outras regras da gramática, temos as regras a seguir:

$$C \rightarrow D$$

$$D \rightarrow if\ (E)\ C$$

$$D \rightarrow if\ (E)\ C\ else\ C$$

A fim de poder gerar cadeias de símbolos terminais e facilitar o entendimento, acrescentamos as regras:

$$C \rightarrow c$$

$$E \rightarrow e$$

Verifique que esta gramática é ambígua e veja como ela pode ser reescrita de forma não ambígua. Isso é feito, por exemplo, na especificação da linguagem Java (GOSLING et al., 2015). Também há descrições da solução na internet, no site disponível em: <http://bit.ly/2sradv3>e <http://bit.ly/2sEDKOR>. Acessos em: 18 ago. 2017.

Nesta seção, vimos gramáticas livres de contexto, árvores de derivação e ambiguidade. Estes conceitos são amplamente usados em cursos de compiladores e até mesmo de linguística. Na próxima seção veremos propriedades importantes das linguagens geradas por este tipo de gramática.

Sem medo de errar

No arquivo que você deve modelar, o tag xml usado para identificar um local é <local>... </local>. Podem ter locais dentro de locais e cada local pode ter um tag <temp> ...</temp> com sua temperatura média. Não há nenhum padrão sobre quantas vezes um tag<local> é

subdividido. Foi pedido que você modele o arquivo criando uma GLC que gere o formato de arquivo.

A seguinte GLC é uma solução para o problema. Nela, os não terminais (variáveis) são as letras maiúsculas e o símbolo inicial é L:

L→<local id="X">JTJ</local>

T→<temp>S</temp>

J→LJ | ε

Falta especificar o símbolo S. Para tanto, basta usar a gramática apresentada na solução da situação-problema na Seção 2.1. Uma vez que ela é regular, ela também é livre de contexto.

Um exemplo de derivação é:

L ⇒ <local id="X">JTJ</local> <local id="X">TJ</local> ⇒ <local id="X">T</local> ⇒ <local id="X"><temp>S</temp></local> ⇒ <local id="X"><temp>-A</temp></local> ⇒ <local id="X"><temp>-1A</temp></local> ⇒ <local id="X"><temp>-11</temp></local>

Avançando na prática

Árvores de derivação representando conexões

Descrição da situação-problema

Na empresa em que você trabalha, um novo presidente achou por bem criar uma equipe de pesquisa e desenvolvimento (P&D). Essa é uma grande oportunidade para você sair da rotina da sua atividade atual de gerenciamento de produto. A empresa então abriu inscrições para o processo de seleção de funcionários para a sua nova equipe de P&D. A seleção passa pela implementação de uma solução para um problema de um cliente e apresentação desta solução. Descrevemos aqui o problema.

Seu cliente é um fabricante de novo tipo de distribuição elétrica para interior de edificações. Esta empresa fabrica três tipos de equipamentos de distribuição: A, B e C. Cada equipamento é formado de duas partes. O equipamento A é formado de Ai e Ao, B de Bi e Bo e C de Ci e Co. Os equipamentos do tipo C são os que fornecem energia diretamente a cada unidade da edificação (escritório ou apartamento). Os equipamentos do tipo A entregam energia para um grupo de equipamentos do tipo B, que por sua

vez entregam para grupos do tipo C. Não há limite preestabelecido para o tamanho destes grupos. Entretanto, por se tratar de um circuito elétrico, a entrada de energia do equipamento do um tipo X (X=A,B ou C) se dá via a parte Xi e a saída se dá via a parte Xo. Assim, para cada entrada de eletricidade via uma parte Xi, deve haver uma saída via Xo. No caso de edifícios residenciais, o equipamento do tipo A deve estar ligado à entrada/saída de eletricidade da edificação. O equipamento do tipo B deve estar ligado à entrada/saída de eletricidade de um andar, enquanto o do tipo C corresponde a um apartamento. Desta forma, a alocação de equipamentos para um edifício de 3 andares, com 2 apartamentos por andar pode ser especificado como AiBiCiCoCiCoBoBiCiCoCi CoBoBiCiCoCiCoBoAo.

A empresa adquiriu um robô de carga e quer usá-lo para depositar os equipamentos nas unidades e andares das edificações. Os comandos do robô são: dep(Y), prox_andar e prox_apto, onde dep(Y) deposita o equipamento/parte Y onde o robô executar o comando, prox_andar e prox_apto deslocam o robô para o próximo andar e para o próximo apartamento, respectivamente. Além destes, há o comando "para". O problema que você deve resolver é projetar o sistema de depósito dos equipamentos/parte em cada unidade/andar/edificação para que, quando os eletricistas forem instalar os equipamentos, eles já estejam corretamente alocados às unidades e caixa de distribuição dos respectivos andares. Para simplificar a solução, você pode considerar que o comando dep(Bi) ou dep(Bo) alocam Bi e Bo, respectivamente, nas caixas de distribuição no andar, o mesmo com respeito a Ci e Co em relação às caixas dos apartamentos.

Resolução da situação-problema

Sua solução passa por programar o robô na entrada do edifício e então deixá-lo realizar a tarefa. Existem dois cuidados aqui: um é que o robô execute o conjunto correto de comandos para que a distribuição dos equipamentos esteja adequada com a edificação. A outra é como gerar este conjunto de comandos. Se você leu com atenção a descrição do problema deve ter prestado atenção a parte que trata de como especificar a alocação de equipamentos em função da edificação. A cadeia

AiBiCiCoCiCoBoBiCiCoCiCoBoBiCiCoCiCoBoAo

que serviu de exemplo para o edifício de 3 andares com 2 apartamentos por andar, pode inspirar o uso deste tipo de expediente para descrever edificações de qualquer tipo. Por exemplo, até mesmo edifícios com configurações diferentes entre os andares podem ser descritos por cadeias desta forma. Por exemplo: "AiBiCiCoBoBiCiCoCiCoBoBiCiCoCiCoCiCoBoAo" descreve uma edificação com 1 apartamento no primeiro andar, dois no segundo e três no terceiro andar.

Você então decide usar cadeias pertencentes à linguagem descrita pela gramática logo a seguir para representar a alocação dos equipamentos da empresa cliente. Além disso, para gerar os programas para o robô distribuidor de equipamentos, você faz o seguinte esquema para traduzir cadeias de distribuição de equipamentos para programas:

Ai gera "dep(Ai); prox_andar"
Bi gera "dep(Bi)"
Cigera "dep(Ci)"
Co gera "prox_apto"
Bo gera "prox_andar"
Ao gera. "Para"

Gramática:

$$S \rightarrow AiZAo$$
$$Z \rightarrow BiTBo \mid BiTBoZ$$
$$T \rightarrow CiCo \mid CiCoT$$

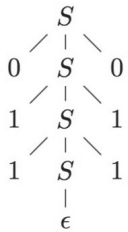
Entre as vantagens da sua solução está a escalabilidade. Sua linguagem formal de alocações dá conta de qualquer tipo de edificação e você ainda é capaz de estendê-la para ser adequada a condomínios (conjunto de edificações) ou até mesmo bairros planejados (conjuntos de condomínios). Basta considerar uma variável D para condomínios com D1 e Do e agregando edificações em condomínios na forma "DiAi....AoAi.....Ao.....Do". Para bairros o esquema é análogo. Mais à frente, quando falarmos de análise sintática e autômatos de pilha, veremos que a forma de gerar comandos

para o robô é via um autômato com saída. Isso é basicamente uma arquitetura de compiladores para linguagens de programação.

Faça valer a pena

1. Seja a linguagem gerada pela gramática $S \to ZT \mid aWd;\ Z \to aZb \mid ab;$ $T \to cTd \mid cd;\ W \to aWd \mid R;\ R \to bRc \mid bc$. Esta gramática gera a linguagem $\{a^m b^n c^n d^m \ / \ m,n > 0\} \cup \{a^m b^m c^n d^n \ / \ m,n > 0\}$.
Considere cadeias na forma $a^n b^n c^n d^n$, $n > 0$. Assinale a resposta correta sobre o número máximo de árvores de derivação possíveis de se obter para estas cadeias com a gramática descrita no texto-base.
a) Nenhuma árvore, pois as cadeias não pertencem à linguagem gerada pela gramática.
b) Somente uma para cada cadeia, pois somente é possível uma derivação mais à esquerda para cada uma destas cadeias.
c) Para cada cadeia existem duas árvores diferentes, pois é possível fazer derivações mais à direita e mais à esquerda com esta gramática.
d) n árvores de derivação diferentes. Para cada cadeia $a^n b^n c^n d^n$ existe uma árvore de derivação para cada aplicação de regra que coloca a na esquerda e d mais à direita.
e) Somente duas para cada cadeia. Existem duas derivações mais à esquerda diferentes para cada cadeia $a^n b^n c^n d^n$, $n > 0$.

2. Seja a seguinte árvore de derivação.



Indique a alternativa que contém as regras gramaticais usadas na derivação e a cadeia que foi derivada.
a) Regras da gramática = $S \to 0S0 \mid 1S1 \mid \epsilon$; cadeia=011110.
b) Regras da gramática = $S \to 0S \mid S0 \mid 1S \mid S1 \mid \epsilon$ cadeia = 011110.
c) Regras da gramática = $S \to 11 \mid 00 \mid \epsilon$ cadeia = 001111.
d) Regras da gramática = $S \to 0S1 \mid 1S0 \mid \epsilon$; cadeia = 011110.
e) Regras da gramática = $S \to 0S1 \mid 1S0 \mid \epsilon$; cadeia = 110011.

3. Seja a linguagem das cadeias na forma $a^n b^m c^{n+m}$, $n, m > 0$.
Assinale a alternativa correta sobre a linguagem descrita no texto-base.
a) A gramática $S \to aS \mid bB;\ B \to bB \mid cC;\ C \to cC \mid c$ gera a linguagem anterior.
b) A gramática $S \to RC;\ R \to aRb \mid ab;\ C \to cC \mid c$ gera a linguagem anterior.
c) A gramática $S \to aSbcc \mid abcc$ gera a linguagem anterior.
d) A gramática $S \to aSc \mid aSb \mid ab$ gera uma quantidade infinita de cadeias da linguagem anterior.
e) A gramática $S \to aSc \mid bSc \mid bc$ gera a linguagem anterior.

Seção 3.2

Linguagens livres de contexto

Diálogo aberto

Recordamos que após concluirmos o estudo de linguagens regulares e seus reconhecedores, vimos casos de linguagens que não são regulares. Para um destes exemplos apresentamos uma gramática não regular que o gera, e definimos as gramáticas livres de contexto.

Nesta seção serão vistas as linguagens livres de contexto, algumas propriedades de fechamento destas, bem como algumas transformações de gramáticas livres de contexto que possuem aplicações à análise sintática.

Em sua participação no projeto de internacionalização da solução de busca de padrões você apresentou a GLC que construiu na última seção. Durante a apresentação você constatou que sua proposta atende à grande maioria dos países. Entretanto, você encontrou um país que usava tags diferentes. Você solicitou um exemplo de arquivo e recebeu o seguinte exemplo:

<paika id="Suomi">
<lamp>-12</lamp>
<paika id="Turku">
<lamp>-23</lamp>
</paika>
<paika id="Uusimaa">
<lamp>-18</lamp>
<paika id="Helsinki">
<lamp>-12</lamp>
</paika>
</paika>
</paika>

Este arquivo tem o nome de local e temperatura traduzidos para o idioma local. Você deve alterar sua GLC para contemplar estes novos arquivos, gerando arquivos de ambos os formatos.

Esta é uma grande oportunidade para ter seu trabalho reconhecido internacionalmente.

Bons estudos!

Não pode faltar

Na seção anterior vimos que se uma linguagem L é gerada por uma gramática livre de contexto (GLC) G, dizemos que L é uma linguagem livre de contexto (LLC). Um exemplo clássico de LLC é a linguagem sobre o alfabeto $\Sigma = \{a, b\}$, definida como $L_R = \{w \mid w^R = w\}$, onde w^R significa a cadeia w revertida (de trás para frente). L_R é uma LLC porque é gerada pela GLC: $S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$.

Outro exemplo de LLC é a linguagem $\overline{L_R} = \{w\#w^R \neq w\}$, complemento da linguagem L_R. O leitor deve tentar construir uma GLC que gere $\overline{L_R}$ antes de ler a solução a seguir, pois se trata de um exercício interessante.

Pesquise mais

Na Unidade 2 vimos que, se L é uma linguagem regular, então seu complemento, $\overline{L}$, também o é. Afirmamos que, tanto a linguagem L_R, como o seu complemento $\overline{L_R}$ são LLC.

Isto gera um questionamento importante: dada uma LLC L, a linguagem $\overline{L}$ necessariamente uma LLC? Pesquise a resposta em livros ou na internet, acessando também o link disponível em: <http://bit.ly/2tkgjP2>. Acesso em: 18 ago. 2017.

Apresentamos agora a GLC que gera a linguagem $\overline{L_R}$:

$S \rightarrow aSa \mid bSb \mid aAb \mid bAa$

$A \rightarrow aAa \mid bAb \mid aAb \mid bAa \mid a \mid b \mid \epsilon$

Para entender o funcionamento desta gramática, observe que toda a forma sentencial (cadeia com terminais e variáveis) gerada possui exatamente uma variável: A ou S. Além disso, uma derivação a partir de S apresentará até determinada regra apenas S como variável na forma sentencial, mas depois da aplicação de uma das regras $S \rightarrow aAb$ ou $S \rightarrow bAa$, as formas sentenciais terão apenas

A como variável. O funcionamento da gramática é tal que as formas sentenciais que têm a variável S são simétricas, ou seja, são iguais quando lidas de trás para frente, enquanto que as formas sentenciais que têm a variável A são assimétricas, são diferentes quando lidas de trás para a frente.

Exemplificando

Dado o alfabeto $\Sigma = \{a, b\}$, considere a linguagem sobre Σ definida como $L_1 = \{a^n b^{2n} \mid n \geq 1\}$. Podemos dizer que L_1 é uma LLC porque é gerada pela GLC a seguir:

$S \rightarrow aSbb$

$S \rightarrow abb$

Por exemplo, para gerar $aabbbb$, podemos fazê-lo através da derivação: $S \Rightarrow aSbb \Rightarrow aabbbb$

A partir da definição de LLC podemos deduzir que as LLC são fechadas em relação à união. Isso significa que dadas duas LLC, L_1 e L_2, a linguagem $L_1 \cup L_2$ sempre é livre de contexto (MENEZES, 2000).

Isso porque se L_1 é gerada por uma GLC G_1 com símbolo inicial S_1 e L_2 é gerada por uma GLC G_2 com símbolo inicial S_2. Podemos criar um novo símbolo inicial, S, acrescentar as regras $S \rightarrow S_1 \mid S_2$ à união das regras das duas gramáticas e obter assim uma GLC G_3 que gera $L_1 \cup L_2$. Quando fazemos essa construção, temos que ter o cuidado de que G_1 e G_2 não podem ter variáveis com o mesmo nome. Entretanto, isso não é problema, porque podemos renomear as variáveis quando necessário.

Exemplificando

Vimos que $L_1 = \{a^n b^{2n} \mid n \geq 1\}$ é gerada pela GLC:

$S_1 \rightarrow aS_1bb \mid abb$

De forma análoga $L_2 = \{a^{2n} nb^n \mid n \geq 1\}$ é gerada pela GLC:

$S_2 \rightarrow aaS_2b \mid aab$

Portanto, a linguagem $L_1 \cup L_2$ pode ser gerada pela gramática

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow aS_1bb \mid abb$

$S_2 \rightarrow aaS_2b \mid aab$

Vamos supor agora que L_1 e L_2 sejam LLC. Podemos afirmar que sua concatenação, $L_1 \circ L_2 = \{\omega_1 \circ \omega_2 \mid \omega_1 \in L_1 \text{ e } \omega_2 \in L_2\}$, é LLC? Afirmamos que sim. O motivo é que se L_1 é gerada por uma GLC G_1 com símbolo inicial S_1 e L_2 é gerada por uma GLC G_2 com símbolo inicial S_2. Podemos criar um novo símbolo inicial, S, acrescentar as regras $S \rightarrow S_1S_2$ à união das regras das duas gramáticas e obter assim uma GLC G_3 que gera $L_1 \circ L_2$. Assim como discutimos no caso da união de linguagens, G_1 e G_2 não podem ter variáveis com o mesmo nome, portanto devemos renomear as variáveis quando necessário.

Exemplificando

$L_1 = \{a^n b^{2n} \mid n \geq 1\}$ é gerada pela GLC:

$S_1 \rightarrow aS_1bb \mid abb$

$L_2 = \{a^{2n} nb^n \mid n \geq 1\}$ é gerada pela GLC:

$S_2 \rightarrow aaS_2b \mid aab$

Portanto, a linguagem $L_1 \circ L_2$ pode ser gerada pela gramática

$S \rightarrow S_1S_2$

$S_1 \rightarrow aS_1bb \mid abb$

$S_2 \rightarrow aaS_2b \mid aab$

Um resultado semelhante é que se L_1 é uma LLC então L_1^* também o é. Mais uma vez, este fato pode ser verificado através de uma construção com gramáticas. Se L_1^* é gerada por uma GLC G_1 com símbolo inicial S_1, para criarmos uma gramática que gera L_1^* basta criarmos um novo símbolo inicial S, e acrescentarmos as regras $S \rightarrow SS_1 \mid \epsilon$ à gramática original.

Passamos a um exemplo. Seja $L_1 = \{a^n b^{2n} \mid n \geq 1\}$. A linguagem é gerada pela GLC:

$S_1 \rightarrow aS_1bb \mid abb$

Portanto, a linguagem L_1^* pode ser gerada pela gramática:

$S \rightarrow SS_1 \mid \epsilon$

$S_1 \rightarrow aS_1bb \mid abb$

A definição de GLC usada neste livro é igual àquela encontrada em Menezes (2000), Hopcroft; Motwani; Ullman (2002) e Garcia (2017), portanto, é uma definição bem estabelecida. Ainda assim, em obras antigas, como Hopcroft; Ullman (1969), encontra-se uma definição diferente, onde a cadeia vazia não é permitida no lado direito da regra, com a possível exceção da regra $S \rightarrow \epsilon$, onde S é o símbolo inicial e S não ocorre no lado direito de uma regra. Neste livro chamaremos tais gramáticas de Gramáticas Livres de Contexto Sem Regras Nulas (GSRN).

Assimile

Dizemos que uma GLC $G = (V, T, P, S)$ é uma Gramática Livre de Contexto Sem Regras Nulas (GSRN) se todas as suas regras são da forma: $A \rightarrow \alpha$

Onde: $A \in V$ e $\alpha \in (V \cup T)^+$.

Se S não está do lado direito de uma regra, é permitida também a regra: $S \rightarrow \epsilon$

Na próxima unidade vamos dar uma definição de Gramáticas Sensíveis ao Contexto com uma restrição semelhante. No momento, estamos interessados em mostrar que a definição de Hopcroft e Ullman (1969) não altera o poder de expressividade das GLC, isto é, se uma linguagem L é gerada por uma GLC G, então a mesma linguagem é gerada por uma GSRN G^1.

Existe um algoritmo para dada uma GLC G obtermos uma GSRN G^1 equivalente. O primeiro passo deste algoritmo é identificar as variáveis que podem gerar a cadeia vazia. Chamaremos a estas de símbolos nulificáveis (GARCIA, 2017). Por exemplo, a gramática:

$S \rightarrow aA$

$A \rightarrow aB \mid B$

$B \rightarrow b \mid \epsilon$

Possui as variáveis {S, A, B} e variáveis nulificáveis {A, B} porque $B \Rightarrow^* \epsilon$ e também $A \Rightarrow^* \epsilon$.

O algoritmo da Figura 3.4 permite o cálculo dos símbolos nulificáveis.

Figura 3.4 | Cálculo dos Símbolos Nulificáveis

```
NULIFICAVEIS(V,T,P,S) {
        CONJ1 = ∅
        CONJ2 = { v∈V|v→ε∈ P}
        Enquanto CONJ1≠ CONJ2 {
                CONJ1 = CONJ2
                CONJ2 = CONJ2∪ {X | X→X1,X2,...,Xn∈P e
X1,X2,...,Xn∈ CONJ2}
        }
        Retorne CONJUNTO2
}
```

Fonte: elaborada pelos autores.

Reflita

O algoritmo apresentado na Figura 3.4 possui variáveis $CONJ_1$ e $CONJ_2$ cujos valores são conjuntos de elementos do conjunto V (conjunto de símbolos não terminais). Portanto $CONJ_1$ e $CONJ_2$ são subconjuntos

de V. O algoritmo possui um laço do tipo 'enquanto'. Este laço será executado até que os conjuntos $CONJ_1$ e $CONJ_2$ sejam iguais. Se o conjunto V tem n elementos, quantas vezes o laço será executado na pior das hipóteses?

Vamos exemplificar a execução do algoritmo com a gramática:

$$S \rightarrow AB$$
$$A \rightarrow a \mid BC$$
$$B \rightarrow b \mid \epsilon$$
$$C \rightarrow c \mid BBB$$

A Tabela 3.1 mostra os valores das variáveis $CONJ_1$ e $CONJ_2$ em cada passo de execução do algoritmo da Figura 3.4.

Tabela 3.1 | Execução do cálculo dos símbolos nulificáveis

$CONJ_1$	$CONJ_2$	$CONJ_1 \neq CONJ_2$
$\varnothing$	{B}	Verdadeiro
{B}	{B,C}	Verdadeiro
{B,C}	{A,B,C}	Verdadeiro
{A,B,C}	{A,B,C,S}	Verdadeiro
{A,B,C,S}	{A,B,C,S}	Falso

Fonte: elaborada pelos autores.

Uma vez que sabemos quais são os símbolos nulificáveis de uma GLC podemos construir uma GSRN equivalente. A ideia é, cada vez que um símbolo nulificável ocorre do lado direito de uma regra, acrescentar uma regra nova à GLC original sem a ocorrência deste símbolo. Por exemplo, na gramática anterior havia a regra $S \rightarrow AB$, a esta regra serão acrescentadas as regras $S \rightarrow A$ (substituindo B pela cadeia vazia) e $S \rightarrow B$ (substituindo A pela cadeia vazia). Evitamos acrescentar a regra $S \rightarrow \epsilon$ (substituindo tanto A como B pela cadeia vazia) porque queremos gerar uma GSRN. Essa ideia simples não funciona caso a cadeia vazia seja gerada pela gramática, isto é, se S é um símbolo nulificável. Neste caso, devemos acrescentar um novo símbolo inicial que gere a cadeia vazia. O algoritmo detalhado é dado pela Figura 3.5:

Figura 3.5 | Cálculo dos símbolos nulificáveis

```
GSRN(V,T,P,S) {
        R = P - { v→ε } // R inicializado com todas as regras não nulas
        Para cada v em NULIFICAVEIS(V,T,P,S)
                Para cada X→w1vw2 em R  // Para cada ocorrência de v à direita
                        Se w1w2≠ε em R
                                R=R∪{ X→w1w2} // Adicionar regra sem o v
        Se S está em NULIFICAVEIS(V,T,P,S)
                Retorne(V∪{ S'},T, R∪{ S'→S, S'→ε },S') // Adicionar novo símbolo inicial
        Retorne(V,T,R,S);
}
```

Fonte: elaborada pelos autores.

A Tabela 3.2 mostra o valor do conjunto de regras R durante a execução do algoritmo da Figura 3.4. Observe que para esta gramática em particular, a saída se dá pelo primeiro comando **Retorne**. Portanto, se a coluna da direita dá o valor final da variável R, o valor retornado é R∪{ S'→S, S'→ε }.

Tabela 3.2 | Execução do cálculo dos símbolos nulificáveis

INICIALIZAR R	Valor de R após 1ª execução do laço: A em NULIFICÁVEIS(V,T,P,S)	Valor de R após 2ª execução do laço: B em NULIFICÁVEIS(V,T,P,S)	Valor de R após 3ª execução do laço: C em NULIFICÁVEIS(V,T,P,S)
S→AB	S→AB S→B	S→AB S→A S→B	S→AB S→AB S→B
A→BC		A→BC A→C	A→BC A→B A→C
C→BBB	C→BBB	C→BBB C→BB C→BB	C→BBB C→BB C→B
A→a	A→a	A→a	A→a
B→b	B→b	B→b	B→b
C→c	C→c	C→c	C→c

Fonte: elaborada pelos autores.

Ou seja, dada como entrada a GLC:

$S \rightarrow AB$

$A \rightarrow a \mid BC$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c \mid BBB$

O algoritmo irá retornar a GSRN equivalente:

$S' \rightarrow S \mid \epsilon$

$S \rightarrow AB \mid A \mid B$

$A \rightarrow a \mid BC \mid B \mid C$

$B \rightarrow b$

$C \rightarrow c \mid BBB \mid BB \mid B$

Sem medo de errar

Recordamos que a gramática feita para resolver o caso em que os tags xml estavam em inglês era:

L→<local id="X">JTJ</local>

T→<temp>S</temp>

J→LJ | ε

Onde S gera as temperaturas (inteiros). Nesse passo você está interessado em tratar os arquivos como os gerados pela gramática acima, juntamente com os arquivos que possuem tags apresentados na descrição da situação-problema.

Você pode acrescentar as regras para os novos tags (que estão no idioma finlandês), obtendo:

F→L|M

A regra citada anteriormente permite uma alternativa entre a variável L e a variável M.

L→<local id="X">JTJ</local>

T→<temp>S</temp>

J→LJ | ε

O trecho anterior se manteve, a variável L gera os tags em inglês.

M→<paika id="X">KUK/paika>

U→<lamp>S</lamp>

K→MK | ε

O trecho é novo a variável M gera os tags em finlandês.

Avançando na prática

Análise sintática de linguagens livres de contexto

Descrição da situação-problema

As gramáticas livres de contexto são usadas em algoritmos de análise sintática. Um dos tipos mais simples de algoritmo de análise sintática é a análise descendente recursiva. Na sua empresa o DBA criou heurísticas para a identificação de consultas que são danosas quando executadas no banco de dados. São aquelas famosas queries que travam o banco de dados. Seu DBA conseguiu programar de forma bem satisfatória essas heurísticas. Entretanto, o programa escrito por ele pede como entrada a árvore de análise sintática da consulta (query) escritas em uma linguagem semelhante à lógica de primeira ordem. O gerente de TI da empresa delegou para você a tarefa de desenvolver um protótipo para converter as consultas nesta linguagem em árvores de análise sintática. De imediato, lembrando das suas aulas de linguagens formais e autômatos, você percebeu que a linguagem de consulta é livre de contexto. De fato, para facilitar e aumentar a velocidade de desenvolvimento, o DBA permitiu que você definisse um subconjunto desta linguagem que inclui a operação binária de união ($\vee$), uma operação unária para a negação ($\neg$) e dois quantificadores: $\forall$ e $\exists$. Usando id como um símbolo terminal único para nome de tabela ou atributo, a linguagem para a qual você deve desenvolver o parser (analisador sintático) é definida pela gramática:

$$Q \rightarrow \forall\ id\ Q \mid \exists\ id\ Q \mid \neg Q \mid Q \vee Q \mid id.$$

Para implementar a análise descendente recursiva a partir de uma GLC, você deve, a partir de cada regra na forma $V \rightarrow aV_1V_2 \ldots V_n$, define uma função ANALISA_V() que tem acesso à linha de leitura (entrada) e que retorna a árvore de análise sintática desta entrada esperando uma derivação de cadeia válida a partir de V. O código de ANALISA_V() é então como descrito a seguir:

```
arv ANALISA_V() {
        chr = le_caractere()
        if (chr == 'a') {
                v1 = ANALISA_V1()
                v2= ANALISA_V2()
                ...
                vn = ANALISA_Vn
                returncria_arv("V", "a", v1,v2,...,vn)
        }
        print(" erro analisando V")
}
```

No caso de regras da forma $V \rightarrow a$. A função é:

```
arv ANALISA_V() {
        chr = le_caractere()
        if (chr == 'a') {
                returncria_arv("V", "a")
        }
        print(" erro analisando V")
}
```

A gramática deve estar escrita de modo que todas as regras tenham uma das formas acima.

Resolução da situação-problema

Para a solução do problema você deve converter todas as regras da gramática para uma das formas:

$V \rightarrow aV_1V_2 \ldots V_n$, ou

$V \rightarrow a$.

Uma sugestão para a gramática da linguagem proposta pelo DBA é então obtida observando-se que as três primeiras regras:

$$Q \rightarrow \forall\ id\ Q \mid \exists\ id\ Q \mid \neg Q$$

Já estão na primeira forma. A última regra já está na segunda forma. Restando apenas a regra:

$Q \to Q \vee Q$

Q Essa regra pode ser transformada substituindo-se o primeiro por todas as possibilidades das outras regras:

$Q \to \forall\, id\, Q \vee Q \,|\, \exists\, id\, Q \vee Q \,|\, \neg Q \vee Q \,|\, id \vee Q$.

Portanto, a gramática final fica:

$Q \to \forall\, id\, Q \,|\, \exists\, id\, Q \,|\, \neg Q \,|\, id \,|\, \forall\, id\, Q \vee Q \,|\, \exists\, id\, Q \vee Q \,|\, \neg Q \vee Q \,|\, id \vee Q.$

Faça valer a pena

1. Vimos nesta unidade que se L_1 e L_2 são LLC, então $L_1 \cup L_2$ também é uma LLC, enquanto que $L_1 \cap L_2$ pode ser ou não uma LLC, dependendo das linguagens em questão. Sejam $L_1 = \{a^n b^n \,|\, \text{n é par}\}$ e $L_2 = \{a^n b^n \,|\, \text{n é múltiplo de 3}\}$. Em particular a cadeia vazia pertence a $L_1 \cap L_2$.

A linguagem L_1 pode ser gerada pela gramática G_1 dada por:

$S \to \epsilon \,|\, aaSbb$

A linguagem L_2 pode ser gerada pela gramática G_2 dada por:

$S \to \epsilon \,|\, aaaaaaSbbbbbb$

Com base no texto-base, assinale a alternativa verdadeira:

a) $L_1 \cap L_2$ é LLC porque é gerada por G.
b) L_1 não é LLC.
c) L_2 não é LLC.
d) $L_1 \cup L_2$ não é LLC.
e) $L_1 \cup L_2$ é LLC porque é gerada por G.

2. Vimos nesta unidade que se L_1 é uma LLC, então L_1^* também o é. Seja $L_1 = \{a^n b^n \,|\, \text{n é par}\}$

A linguagem L_1 pode ser gerada pela gramática G_1 dada por:

$S \to \epsilon \,|\, aaSbb$

Com base no texto, assinale a alternativa verdadeira:

a) L_1^* é LLC porque é gerada pela gramática: $S \to SS \,|\, \epsilon \,|\, aaSbb$.
b) L_1^* não é LLC.
c) L_1 não é LLC.
d) L_1^* é regular porque é a linguagem das cadeias que possuem número par de caracteres a e número par de caracteres b.
e) L_1^* é LLC porque é gerada pela gramática: $S \to AS \,|\, \epsilon ; A \to \epsilon \,|\, aaAbb$.

3. Nesta unidade vimos que os símbolos nulificáveis de uma gramática são as variáveis que podem gerar a cadeia vazia. Considere a gramática G dada por:

$S \rightarrow AC$
$A \rightarrow a \mid aBC$
$B \rightarrow b \mid \epsilon$
$C \rightarrow c \mid BB$

Com base no texto, assinale o conjunto de símbolos nulificáveis de G:

a) $\{B, C\}$
b) $\{A, B\}$
c) $\{A, C\}$
d) $\{A, B, C\}$
e) $\{S, A, B, C\}$

Seção 3.3

Autômatos com pilha

Diálogo aberto

Nesta unidade, já vimos linguagens e gramáticas livres de contexto. Em particular vimos os conceitos de derivações mais à esquerda, mais à direita, árvores de derivação e propriedades de fechamento de linguagens livres de contexto.

No caso de linguagens regulares, resolvemos o problema da análise sintática usando o AFD, também desejamos resolver o problema da análise sintática para linguagens livres de contexto. Para tanto, será apresentado nesta seção o Autômato com Pilha.

Ao apresentar sua solução para o problema de localização de temperaturas negativas a outros países você foi informado que em muitos países os dados não são armazenados em formato ".csv", mas sim em formato ".xml". Nas últimas seções você apresentou duas versões de GLC que modelam a sintaxe dos arquivos ".xml" usados para armazenar os dados de temperatura. Agora você é o responsável pela equipe que irá implementar a nova solução de localização de padrões e disponibilizá-la como plataforma mundial para sua instituição. Para guiar o processo de implementação você deve apresentar um autômato com pilha que reconheça os arquivos ".xml". Você pode supor que os números são um símbolo terminal especial "&N", que são identificados em um pré-processamento. Esta é uma ótima oportunidade para continuar o seu excelente trabalho em detecção de padrões, que está sendo reconhecido internacionalmente.

Bons estudos!

Não pode faltar

Nesta seção, vamos apresentar o autômato com pilha (AP), um tipo de autômato que reconhece as linguagens livres de contexto (HOPCROFT; MOTWANI; ULLMAN, 2002).

Considere o alfabeto $\Sigma = \{a, b\}$ e L_1 a linguagem sobre Σ definida como $L_1 = \{a^n b^n \mid n \geq 1\}$. Na Seção 3.2 da Unidade 2 vimos

que L_1 não é regular, enquanto que na Seção 1 da unidade atual vimos que esta linguagem é livre de contexto. Uma vez que L_1 não é regular, não existe um AFD que reconheça L_1. Vamos propor um algoritmo para reconhecer cadeias de L_1 que estende o comportamento de um $AF\epsilon$. O algoritmo da Figura 3.1 inclui uma pilha ao código apresentado na solução da situação-problema da Seção 2.2 da Unidade 2. A função *nova_pilha(c)* retorna uma pilha com um único símbolo empilhado, aquele passado como o parâmetro *c*. A função *pilha.topo()* não só retorna o elemento que está no topo da pilha, como também o desempilha, enquanto a função *pilha.empilha(w)* empilha a cadeia de caracteres (*w)* (possivelmente vazia).

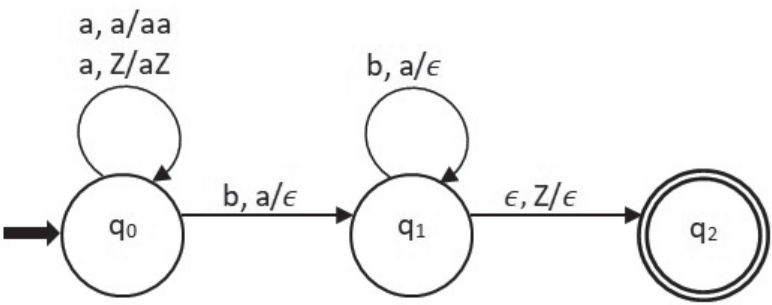
Figura 3.6 | Reconhecedor para a linguagem L_1

```
estado = 0
pilha = nova_pilha('Z')
Enquanto (1) { /* Loop Infinito até sair no retorne */
        entrada = leCaractere()
        t = pilha.topo()
        se (estado==0 && entrada=='a' &&(topo=='a'|| topo=='Z')) {
                estado=0
                pilha.empilha("a")
        } senão se (estado==0 && entrada=='b'&& topo=='a') {
                estado=1
                pilha.empilha("") /* Empilha a cadeia vazia */
        } senão se (estado==1 && entrada=='b'&& topo=='a') {
                estado=1
                pilha.empilha("") /* Empilha a cadeia vazia */
        } senão se (estado==1 && entrada==EOF && topo=='Z') {
                Retorne 1 /* 'Z' foi desempilhado, a pilha esta vazia */
        } senão {
                Retorne 0
        }
}
```

Fonte: elaborada pelos autores.

A ideia do algoritmo da Figura 3.6 é empilhar o símbolo '*a*' enquanto a entrada for este símbolo. Quando entrar o primeiro 'b', começamos a desempilhar o símbolo '*a*', além disso, mudamos do estado 0 para o estado 1, assim o algoritmo não voltará a empilhar o símbolo '*a*', caso este volte a ocorrer na entrada, o programa terminará entrando no "*Retorne 0*". Observe que cada teste de condição depende de 3 variáveis: estado, entrada e topo da pilha. Vamos representar o algoritmo da Figura 3.6 através da Figura 3.7.

Figura 3.7 | Autômato com pilha para a linguagem L_1

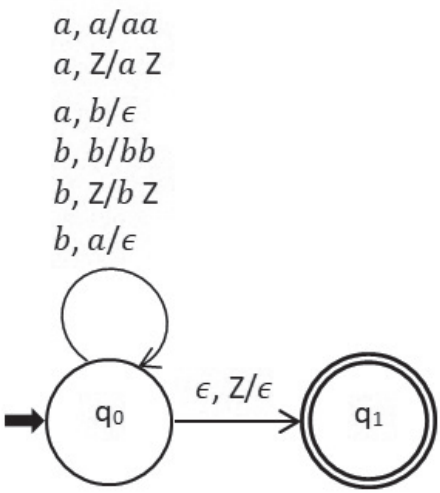


Fonte: elaborada pelos autores.

No autômato da Figura 3.7, a pilha é inicializada com o símbolo Z e a transição entre os estados q_0 e q_1 tem o texto $b, a/\epsilon$, correspondente ao segundo **if** do programa da Figura 3.6, isto é, estando no estado q_0, ao ler um b da entrada e tendo um símbolo a no topo da pilha, irá desempilhar este símbolo, empilhar ϵ, e ir para o estado q_1. Da mesma forma, a transição do estado q_0 para ele próprio, corresponde ao primeiro if do programa da Figura 3.1, estando no estado q_0, ao ler um a da entrada e tendo um símbolo Z no topo da pilha, irá desempilhar este símbolo, empilhar aZ (o mesmo que empilhar primeiro Z e depois a), e permanecer no estado q_0. O efeito prático neste caso é empilhar um a acima do Z. No programa da Figura 3.1 não havia o estado q_2, no quarto **if** do programa havia o comando "Retorne 1". Na Figura 3.7, esta condição de aceitação é indicada fazendo com que o autômato, sem consumir nenhum elemento da entrada e com o símbolo Z no topo da pilha, passe do estado q_1 para o estado final q_2. A aceitação se dá se o autômato parar em um estado final após ler toda a entrada.

Vamos considerar agora uma linguagem um pouco diferente. Seja $L_2 = \{w \in \{a,b\}^* \mid w$ tem a mesma quantidade de caracteres a e de caracteres $b\}$. Essa linguagem é gerada pela GLC: $S \to aB \mid b \mid \epsilon$, $A \to bS \mid aBB$, $B \to aS \mid bAA$. O autômato com pilha que reconhece a linguagem L_2 é dado na Figura 3.8.

Figura 3.8 | Autômato com pilha para a linguagem L_2



Fonte: elaborada pelos autores.

Observe que o autômato da Figura 3.8. Ele possui estados q_0 e q_1. O estado inicial é q_0, marcado com uma seta. O estado final é q_1, marcado com um círculo duplo. A regra de transição $a, Z / aZ$ significa que ao ler um a da entrada, se o símbolo Z está no topo da pilha, irá desempilhar este símbolo (lembre-se que a função topo da Figura 3.6 desempilha um símbolo) e irá empilhar aZ, ou seja, colocar o Z de volta e empilhar o a sobre ele. Observamos que ao ler um a o autômato pode fazer duas coisas: se o topo da pilha for a ou Z ele irá empilhar o a, se o topo da pilha for b, irá apenas desempilhar o b. O comportamento ao ler o símbolo b é análogo. Assim a pilha irá sempre ter apenas símbolos a (sobre o símbolo inicial Z) ou apenas símbolos b. E o número de símbolos na pilha indica quantas vezes aquele símbolo apareceu a mais na entrada. Ao final, se a pilha está apenas com o símbolo inicial, significa que não houve excesso de nenhum símbolo, permitindo a transição que leva ao estado final q_1.

Assimile

Um *autômato com pilha* (AP) é uma tupla P = (Q, Σ, Γ, δ, q_0, Z, F), onde:

1. Q é um conjunto finito de estados;

2. Σ é o alfabeto de entrada;

3. Γ é o alfabeto da pilha, ou seja, os símbolos que podem ser escritos na pilha.

4. $\delta : Q \times (T \cup \{\epsilon\}) \times \Gamma \rightarrow \wp(Q \times G^*)$, ou seja, a função δ recebe um estado, possivelmente um símbolo de entrada (a ser lido) e um símbolo da pilha (a ser desempilhado) e devolve pares de estados e cadeias de Γ^* (a serem empilhadas);

5. $q_0 \in$ Q é o estado inicial;

6. Z $\in \Gamma$ é o símbolo inicial da pilha;

7. F $\subseteq$ Q é o conjunto de estados finais.

Podemos agora representar formalmente o AP da Figura 3.8. Onde cada etiqueta nas setas gera um valor da função δ.

Exemplificando

Para o AP da Figura 3.8 temos P = (Q, Σ, Γ, δ, q_0, Z, F), onde:

1. $Q = \{q_0, q_1\}$

2. $\Sigma = \{a, b\}$

3. $\Gamma = \{a, b, Z\}$

4. $\delta(q_0, a, a) = \{(q_0, aa)\}$
$\delta(q_0, a, Z) = \{(q_0, aZ)\}$
$\delta(q_0, a, b) = \{(q_0, \epsilon)\}$
$\delta(q_0, b, b) = \{(q_0, bb)\}$

$\delta(q_0, b, Z) = \{(q_0, bZ)\}$

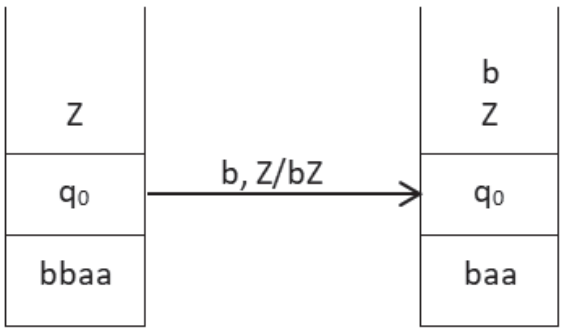
$\delta(q_0, b, a) = \{(q_0, \epsilon)\}$

$\delta(q_0, \epsilon, Z) = \{(q_1, \epsilon)\}$

5. $F = \{q_1\}$

Para entender passo a passo o funcionamento do autômato vamos indicar uma configuração do autômato como na Figura 3.9. A linha mais abaixo significa o que falta ler de uma cadeia. Na linha imediatamente superior temos o estado, acima do estado temos o conteúdo da pilha, iniciando pelo símbolo Z. A Figura 3.9 mostra, à esquerda, a configuração inicial do autômato da Figura 3.8 antes de começar a ler a cadeia $bbaa$, à direita, a configuração seguinte, após ler o primeiro b.

Figura 3.9| Primeiro passo do autômato com pilha para a linguagem L_2 lendo $bbaa$



Fonte: elaborada pelos autores.

O funcionamento passo a passo do autômato até atingir o estado final no reconhecimento da cadeia $bbaa$ é dado na Figura 3.9. Observe que quando o autômato vai para o estado final, q_1, a entrada já foi totalmente lida, situação representada pelo caractere ϵ na linha de baixo, portanto a cadeia é reconhecida.

Figura 3.10 | Passo a passo do autômato com pilha para a linguagem L_2 lendo $bbaa$

Z	b Z	b b Z	b Z	Z	ϵ
q_0	q_0	q_0	q_0	q_0	q_1
bbaa	baa	aa	a	ϵ	ϵ

Fonte: elaborada pelos autores.

Os dados representados em cada coluna da Figura 3.10 (caracteres restantes a serem lidos; estado e conteúdo da pilha) são chamados em conjuntos de configuração do autômato com pilha. Em geral são representados como uma tripla ordenada da forma: (estado, entrada, conteúdo da pilha). A configuração inicial da Figura 3.10 pode assim ser representada por: $(q_0, bbaa, Z)$, enquanto o último estado da mesma figura pode ser representado como: $(q_1, \epsilon, \epsilon)$. A relação de transição entre configurações é representada pelo símbolo: $\vdash$. Outra forma de representar a Figura 3.10 é através da sequência de transições entre configurações da Figura 3.11. Observe que o topo da pilha é o caractere mais à esquerda da terceira componente da tripla ordenada.

Figura 3.11 | Transição entre configurações do AP para a linguagem L_2 lendo $bbaa$

$$(q_0, bbaa, Z) \vdash (q_0, baa, bZ) \vdash (q_0, aa, bbZ) \vdash (q_0, a, bZ) \vdash (q_0, \epsilon, Z) \vdash (q_1, \epsilon, \epsilon)$$

Fonte: elaborada pelos autores.

Quando a transição entre configurações se dá em zero ou mais passos, representamos esta sequência de transições por $\vdash^*$. Para as transições da Figura 3.11 podemos escrever $(q_0, bbaa, Z) \vdash^* (q_1, \epsilon, \epsilon)$. Como q_1 é um estado final, dizemos que $bbaa$ é aceito pelo autômato de pilha. Em geral, dado um autômato de pilha temos P = (Q, Σ, Γ, δ, q_0, Z, F), definimos a linguagem reconhecida por P por estado final como todas as cadeias que levam P da configuração inicial até uma configuração cujo estado é o estado final.

Assimile

Dado um autômato com pilha P = (Q, Σ, Γ, δ, q_0, Z, F), definimos a linguagem aceita por P por estado final, L(P), como:

$L(P) = \{w \in \Sigma^* \mid (q_0, w, Z) \vdash^* (q, \epsilon, \alpha)$ para algum $\alpha \in \Gamma^*$ e algum $q \in F\}$.

Observe que para o autômato P_2 da Figura 3.8 $L(P_2) = L_2$. Para este autômato, as cadeias que levam ao estado final q_1 são exatamente aquelas que deixam a pilha vazia. Muitas vezes, para facilitar a escrita de um autômato com pilha P ou sua programação, deixamos de colocar um estado final e dizemos que a linguagem reconhecida pelo AP são as sentenças que levam o autômato a uma configuração onde a pilha está vazia. Neste caso, dizemos que a linguagem é reconhecida por pilha vazia e escrevemos esta linguagem como $N(P)$. No caso particular do autômato P_2 da Figura 3.8, temos $N(P_2) = L(P_2) = L_2$.

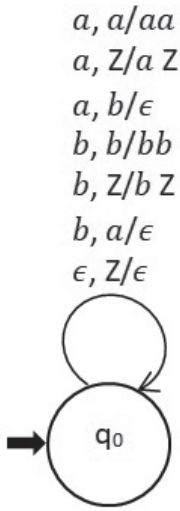
Assimile

Dado um autômato com pilha P = (Q, Σ, Γ, δ, q_0, Z, F), definimos a linguagem aceita por P por pilha vazia, N(P), como:

$N(P) = \{w \in \Sigma^* \mid (q_0, w, Z) \vdash^* (q, \epsilon, \epsilon)$ para algum $q \in Q\}$.

Dado um autômato com pilha que reconheça uma linguagem L por pilha vazia, sempre é possível obter um outro autômato com pilha que reconheça L por estado final e vice-versa (HOPCROFT; MOTWANI; ULLMAN, 2002). O reconhecimento por pilha vazia pode simplificar o desenho de um autômato. O autômato da Figura 3.7 reconhece, por pilha vazia, a mesma linguagem (L_2) que o autômato da Figura 3.3, mas tem um estado a menos.

Figura 3.12 | Autômato com pilha que reconhece a linguagem L_2 por pilha vazia



Fonte: elaborada pelos autores.

Observe que a regra $\delta(q_0,\epsilon,Z)=(q_0,\epsilon)$ pode ser aplicada a qualquer momento, tornando a pilha vazia, mas isto só caracteriza o reconhecimento de uma cadeia se a mesma foi inteiramente lida quando a pilha ficou vazia.

Os autômatos das Figuras 3.7 e 3.8 exigiram alguma criatividade em seu projeto. No caso de linguagens regulares havia um algoritmo que, dada uma gramática regular, produzia um AFND equivalente. Felizmente o mesmo se dá para autômatos com pilha. Existe um método para, dada uma gramática livre de contexto, produzir um AP que reconhece a linguagem gerada pela gramática. Vamos ilustrar esta construção para a gramática livre de contexto G_{exp}, vista na Seção 3.1, que gera as expressões aritméticas formadas com as operações soma e multiplicação e o número '1'.

$S \to S+S$,
$S \to S\times S$,
$S \to (S)$,
$S \to 1$.

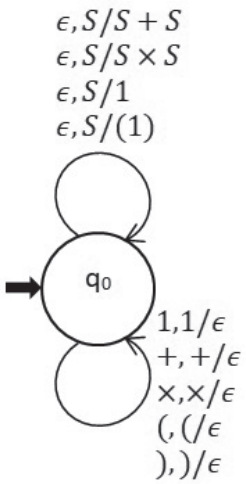
A construção do autômato se dá da seguinte forma. O AP será P = ($\{q_0\}$, $\{1,+,\times,(,)\}$, $\{S,1,+,\times,(,)\}$, δ, q_0, S, $\varnothing$), onde:

1. Para cada símbolo $a \in \{1, +, \times, (,)\}$ fazemos uma regra que ao ler a da entrada, se a está no topo da pilha, então o desempilhamos.

2. Para regra $A \to w$ da gramática, fazemos uma regra que, sem ler a entrada, desempilha A e empilha w.

Para a gramática acima o AP produzido é o da Figura 3.13, com símbolo inicial da pilha sendo S. O autômato reconhece $L(G_{exp})$ por pilha vazia. Se quisermos reconhecer a linguagem por estado final, é necessário mais um estado. Observe que o primeiro tipo de regra está agrupado na seta abaixo do estado, e o segundo tipo de regra está agrupado na seta acima.

Figura 3.13 | Autômato com pilha que reconhece a linguagem gerada por G_{exp}



Fonte: elaborada pelos autores.

Pesquise mais

Vimos anteriormente como, a partir de uma GLC, construir um AP equivalente. Também é possível fazer o processo inverso: dado um AP construir uma GLC equivalente. O leitor pode encontrar como construir esta gramática em Menezes (2000).

A versão determinística do AP é amplamente usada nos compiladores para programar a chamada análise sintática descendente. Também existem os autômatos com pilha ascendentes (APA), cuja versão determinística é usada na análise sintática ascendente (GARCIA, 2017).

Reflita

Verifique o comportamento de autômato da Figura 3.11 no reconhecimento da cadeia "1+1x1". Para cada regra de transição do segundo tipo executada, anote a regra da gramática associada. Esta sequência de regras corresponde a uma derivação da cadeia "1+1x1". Reflita:

- Qual o tipo de derivação obtido?
- Existe mais de uma sequência de regras de transição que leva ao reconhecimento desta cadeia? Por quê?

Com isso, concluímos esta unidade. Na próxima unidade veremos máquinas de Turing, linguagens sensíveis ao contexto e linguagens recursivamente enumeráveis.

Sem medo de errar

Vamos apresentar um Autômato com Pilha que reconhece os arquivos de armazenamento de temperatura por pilha vazia. Vamos supor que os números são um símbolo terminal especial "&N", que são identificados em um pré-processamento. A gramática de tais arquivos é:

F→L|M

L→<local id="X">JTJ</local>

T→<temp>&N </temp>

J→LJ | ε

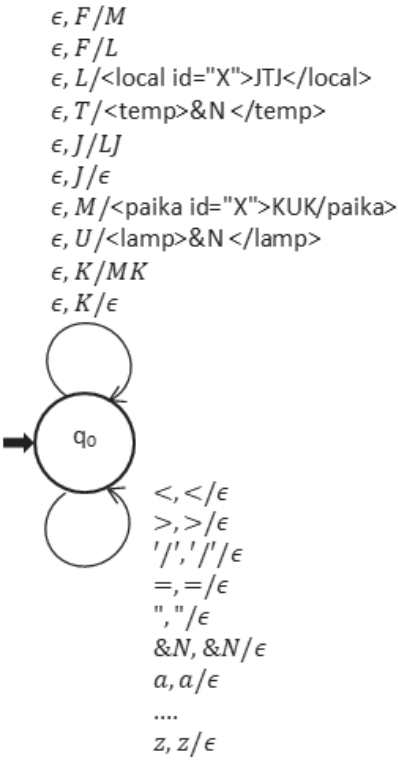
M→<paika id="X">KUK/paika>

U→<lamp>&N </lamp>

K→MK | ε

Usando o método visto nesta seção podemos obter o seguinte autômato com pilha, com símbolo inicial da pilha sendo F, que reconhece a linguagem por pilha vazia.

Figura 3.14 | Solução da SP



Fonte: elaborada pelos autores.

Avançando na prática

Linguagens determinísticas e não determinísticas

Descrição da situação-problema

Após sua implementação do analisador descendente recursivo, a pedido do DBA da sua empresa, outros desenvolvedores começaram a usar a técnica de gerar analisadores sintáticos a partir de gramáticas. Isso funcionou até um dia em que um desenvolvedor resolveu aplicar esta técnica a um exemplo bem

conhecido e simples de linguagem livre de contexto: A linguagem dos palíndromos de tamanho par, gerados pela gramática $S \rightarrow aSa \mid bSb \mid aa \mid bb$. Seu colega escreveu a primeira versão do programa, como mostrada a seguir:

```
Arv function S()=
    read(ch)
    If ch == 'a'   then  S() A()
      elseif ch == 'b' then  S() B()
            else print('Sintaxe com erro')

Arv function B()=
    read(ch)
    If ch == 'b' then return arv('b')
      else  print('Sintaxe com erro')

Arv function A()=
    read(ch)
    If ch == 'a' then return arv('a')
      else  print('Sintaxe com erro')
```

Depois de escrever a versão acima, seu colega percebeu que havia ignorado as regras $S \rightarrow aa$ e $S \rightarrow bb$. Tentou considerá-las então em uma nova versão do programa. Entretanto, teve extrema dificuldade na implementação desta segunda versão. O problema é basicamente detectar a necessidade de uso da regra $S \rightarrow aa$ e não $S \rightarrow aSa$. Após uma rápida discussão com outros colegas ele chegou à conclusão que mesmo que use $S \rightarrow \epsilon$ e $S \rightarrow aSa$ no lugar das regras acima, ele não resolve o problema, visto que o uso da regra $S \rightarrow \epsilon$ é possível em qualquer passo da análise sintática. O problema é basicamente que na análise de uma cadeia da forma $aaaaaaaa$ o programa acima `entra em loop', pois não há como descobrir onde é o meio da cadeia sem um processo de contagem, e para contar a quantidade de símbolos na cadeia inteira deve-se lê-la por completo. Isto é, deve-se primeiro ler a cadeia e

contar quantos símbolos ela tem e somente aí iniciar uma versão da função S na forma $S(m)$, onde m é inicialmente a metade do comprimento da cadeia, e a cada chamada de S é decrementado. Quando se chama $S(0)$ a função retorna a árvore vazia. Seu colega testou esta ideia, que funcionou. No entanto, alguém se lembrou da disciplina de linguagens formais e autômatos que fez na faculdade e disse que os analisadores de linguagens livres de contexto, tais como a linguagem dos palíndromos, é facilmente aceita por Autômatos de Pilha, mas se lembrava de haver duas versões destes autômatos a determinística e a não determinística.

Você ficou então encarregado de verificar essa informação e apresentar em um seminário o que existe de relevante sobre esse assunto e o problema da análise sintática de palíndromos através de suas gramáticas, já que você é quem tem mais intimidade com estes autômatos de pilha.

Resolução da situação-problema

O primeiro tópico da sua apresentação é comparar a análise descendente recursiva com autômatos de pilha. Você faz isso ao mostrar como uma linguagem de programação implementa chamadas a funções/procedimentos/módulos. O compilador dispõe de um conceito chamado de registro de ativação, que nada mais é que uma tabela de símbolos, isto é, uma tabela com espaço para armazenamento dos valores de cada variável local à função.

No caso das funções S, A e B no programa de seu colega, o registro de ativação de cada função é uma tabela que tem espaço para armazenar o valor de ch, um caractere, símbolo a ou b. A cada chamada de uma função F por uma função G, o compilador empilha o registro de ativação da função F e passa a executar o código de F, que se utiliza dos valores das suas variáveis locais armazenadas neste registro de ativação. Quando a função termina a execução, o compilador desempilha o registro de ativação. Você então monta sua apresentação para mostrar como a execução de um analisador descendente recursivo correspondente exatamente à execução de um autômato de pilha. No entanto, a versão de autômato de pilha que qualquer analisador descendente recursivo implementa é a versão determinística. Você então parte para a segunda parte do seminário, que é mostrar que o caso mais geral e

essencial de autômatos para aceitar linguagens livres de contexto é a versão não determinística. Do seu material de aula você colhe o exemplo de autômato não determinístico correspondente à aceitação da linguagem de palíndromos, que é especificada pela gramática acima. O ponto mais importante do seu seminário é a apresentação de que não é possível aceitar esta linguagem com um autômato de pilha determinístico. Claro que a intuição em que se baseia tal fato é a questão já levantada que motivou seu seminário: "Como instrumentar o código para determinar que está lendo o meio exato da cadeia que inda não se sabe qual é?". Entretanto, você vai além e descobre na literatura de linguagens formais uma demonstração mais matemática deste fato e a explica ao grupo.

A seguir segue a demonstração:

$L = \{ww^R / w \in \{a,b\}^*\}$ é a linguagem em questão. Suponha que existe um autômato de pilha determinístico (APD) $\mathcal{A}$ que aceita L. Vamos considerar a seguinte família de cadeias sobre a e b. Seja para $k \in \mathbb{N}$, $u_k = ab^{2k}a$, v_0 é a palavra vazia ϵ e $v_{k+1} = v_k u_k v_k$. Para cada k podemos verificar que $v_k \in L$. Mais que isso, cada v_k é prefixo de v_{k+1}. Desta forma, durante a leitura de uma cadeia v_{i+1}, após ler seu prefixo v_i, $\mathcal{A}$ estará em um estado final q_i e com a pilha vazia. Como existe uma quantidade infinita de cadeias na forma v_j, $j \in \mathbb{N}$, deve existir pelo menos um par de números j, i, tais que $q_i = q_j$, pois a quantidade de estados do autômato é finita. Desta forma, $\mathcal{A}$ não é capaz de distinguir $v_i u_i v_i$ de $v_j u_i v_i$, pois ambos são aceitos por $\mathcal{A}$. No entanto, $v_j u_i v_i$ não é palíndromo. Temos então uma contradição. Portanto, não aceita L.

Faça valer a pena

1. Considere a seguinte gramática livre de contexto G:

$S \to Z \mid U \mid ZU \mid UZ;$

$Z \to 0 \mid 0Z0 \mid 1Z0 \mid 0Z1 \mid 1Z1;$

$U \to 1 \mid 0U0 \mid 1U0 \mid 0U1 \mid 1U1$

Considere que você sabe que a linguagem das cadeias ww não é livre de contexto, $w \in \{0,1\}^*$.

Sobre a linguagem gerada pela gramática citada anteriormente, assinale a alternativa verdadeira.

a) $L(G) = \{w_1 w_2 / w_1, w_2 \in \{0,1\}^+ \text{ e } w_1 \neq w_2\}$.

b) $L(G)$ é aceita por autômato com pilha determinístico.

c) O complemento de $L(G)$ é aceito por autômato com pilha não determinístico.
d) $L(G) = \{0,1\}^* - \{ww \mid w \in \{0,1\}^*\}$.
e) As variáveis Z e U da gramática geram cadeias de tamanho par.

2. Considere as linguagens $L_1 = \{a^n b^n / 0 \leq n\}$ e L_2 a linguagem das cadeias com uma quantidade ímpar de ocorrência de a's.
Assinale a alternativa correta.
a) Não é possível reconhecer L_2 com um autômato de pilha com somente um estado, pois é necessário pelo menos dois estados para testar a paridade de uma cadeia.
b) $L_1 - L_2$ não é livre de contexto.
c) $L_1 \cap L_2$ é aceita por um autômato com pilha determinístico.
d) $L_2 - L_1$ não é aceita por autômato com pilha.
e) O complemento de L_2 não é aceito por autômato com pilha.

3. Considere a linguagem $L = \{wcw^R / w \in \{a,b\}^*\}$.
Assinale a alternativa correta.
a) Um AP determinístico P faz o seguinte: empilha cada símbolo lido da fita e após ler c passa a desempilhar conforme lê o mesmo símbolo que é topo da pilha. Portanto P aceita L.
b) Sem o caractere c no centro da cadeia, a linguagem ainda poderia ser aceita por APD.
c) O complemento de L não é livre de contexto.
d) $L = \{wcw^R / w \in \{a,b\}^*\}$ não é aceita por APD.
e) L estendida para incluir todos os caracteres da língua portuguesa, não é uma linguagem livre de contexto, pois o APND não lida com muitos símbolos.

Referências

GARCIA, A. **Linguagens regulares e livres de contexto**. 1. ed. Rio de Janeiro: Edição do Autor (eBook Kindle), 2017.

GOSLING, J. et al. **The Java Language Specification, Java SE 8 Edition**. 1a. ed. Redwood: Oracle, 2015.

HOPCROFT, J.; ULLMAN, J. **Formal Languages and Their Relation to Automat**a. Reading: Addison-Wesley, 1969.

HOPCROFT, J.; MOTWANI, R.; ULLMAN, J. **Introdução à teoria de autômatos, Linguagens e Computação**. 1. ed. Boston: Elsevier, 2002.

MENEZES, P. B. **Linguagens formais e autômatos**. Porto Alegre: Sagra Luzzatto, 2000.

Linguagens sensíveis ao contexto e recursivamente enumeráveis

Convite ao estudo

Nesta unidade, vamos estudar mais um tipo de autômato: a máquina de Turing. Vamos estudar mais um tipo de gramáticas as gramáticas sensíveis ao contexto, bem como as linguagens geradas por estas gramáticas, que são as linguagens sensíveis ao contexto. Diferentemente dos autômatos estudados nas unidades anteriores, a máquina de Turing foi proposta como um modelo para o que uma máquina pode fazer. Isso significa que a máquina de Turing tem o mesmo poder de computação que um computador. Esse fato vai nos levar ao estudo de linguagens recursivas, recursivamente enumeráveis e da tese de Church.

Após um ano em seu novo emprego, devido ao sucesso internacional do seu programa de reconhecimento de padrões em aquivos .csv, e outras iniciativas bem-sucedidas, você foi muito bem avaliado e ganhou uma promoção. Após sua promoção você recebeu um novo padrão de busca de dados e foi solicitado que você o especificasse com uma gramática sensível ao contexto.

Na Seção 4.1, vamos estudar gramáticas e linguagens sensíveis ao contexto e suas propriedades de fechamento. Na Seção 4.2, vamos estudar as máquinas de Turing. Finalmente, na Seção 4.3, vamos estudar linguagens recursivas, recursivamente enumeráveis e a tese de Church. Todo este conhecimento é útil para os novos desafios que você está encontrando em sua atividade profissional, portanto, continue seus estudos com afinco.

Seção 4.1

Linguagem sensível ao contexto

Diálogo aberto

Recordamos que após completar um ano em seu novo emprego, você foi muito bem avaliado pelo seu trabalho e ganhou uma promoção.

Após sua promoção, você recebeu um novo padrão de busca de dados e foi solicitado que você o especificasse com uma gramática sensível ao contexto. O padrão envolve um arquivo com sintaxe complexa, mas em um primeiro momento pode ser aproximado para a linguagem $L=\{a^n b^n c 2^n, n \geq 0\}$. Você deve escrever uma gramática sensível ao contexto que gere essa linguagem. Esta é a chance do seu trabalho continuar a ser reconhecido internacionalmente em sua empresa.

Bons estudos!

Não pode faltar

Nesta seção vamos estudar as gramáticas sensíveis ao contexto. Primeiramente, vamos entender um pouco sobre esta denominação de gramáticas livres/sensíveis ao contexto. Como vimos na Unidade 3, as gramáticas livres de contexto possuem todas as regras na forma $A \rightarrow \gamma$, onde $\gamma \in (V \cup T)^*$. Regras nessa forma definem o γ como sendo uma das possíveis instâncias do conceito associado à variável A. Por exemplo, em uma possível gramática das orações em língua portuguesa, temos que orações são constituídas de sujeito, seguido de predicado. Predicados são um verbo seguido de adjunto adverbial. Isso em termos de gramática formal pode ser escrito como $<O> \rightarrow <S><P>$; $<P> \rightarrow <Verb><AdAdv>$, onde $<O>$, $<S>$, $<P>$, $<Verb>$ e $<AdAdv>$ são variáveis que estão associadas aos conjuntos de orações, sujeitos, predicados, verbos e adjuntos adverbias, respectivamente. Os símbolos $<>$ em torno das variáveis apenas enfatizam o fato que elas são variáveis e não símbolos terminais, isto é, $<S>$ representa uma variável enquanto S é a própria letra "S"'. Se a essas regras adicionamos as regras

$Verb \rightarrow pegar \mid pego \mid pegamos \mid pegam\ estar \mid sou \mid somos$, $< S > \rightarrow Eu \mid a\ casa$ e $< AdAdv > \rightarrow o\ carro \mid em\ casa$ teremos condições de gerar orações tais como:

"Eu pegamos para casa" ou "A casa pegamos o carro"

Que são orações que precisamos ser bastante generosos para sermos capazes de processar. Apesar destas orações estarem de acordo com a gramática, elas ainda são orações inválidas na gramática da língua portuguesa. Na língua portuguesa existem fenômenos de linguagem que incluem a concordância verbal, em relação ao sujeito, e também conta com a noção de regência de verbo, isto é, se o verbo "pede" ou não objeto direto/indireto. Na primeira oração anterior, o sujeito induz que a concordância/conjugação do verbo esteja na primeira pessoa do singular, isto é, para o verbo "pegar" é "pego". Um outro ponto na mesma oração é o fato de "pegar" ser transitivo direto, portanto está faltando o objeto direto, isto é, que objeto está sendo pego. Condições similares aparecem na segunda oração. Dizemos que a gramática da língua portuguesa leva em conta o contexto onde cada elemento frasal aparece. A conjugação de um verbo depende do contexto em que este verbo aparece na oração. Como vimos neste exemplo com orações, estas não parecem ser cadeias especificadas via uma gramática livre de contexto. Antes de definirmos o tipo de gramática que dá conta deste aspecto das linguagens naturais, como a língua portuguesa, vamos ver um exemplo deste mesmo fenômeno no universo das linguagens de programação.

Considere a sintaxe da linguagem C. Um comando de atribuição em C é uma cadeia na forma $< Id >=< Exp >$, onde $< Id >$ e $< Exp >$ são variáveis da gramática, que, portanto, tem a regra $< Atr > \rightarrow < Id >=< Exp >$ como uma das componentes da gramática de C. No entanto, mesmo sendo possível derivar (hipoteticamente) a cadeia $x = y + c$ a partir da variável $< Atr >$, não há garantias gramaticais para nos assegurarmos que $x = y + c$ é um comando válido em C. Por exemplo, se x é do tipo integer (int), y do tipo char e c do tipo float, alguns compiladores de C não aceitam o comando como sendo válido. Basicamente, o contexto (cadeia) em que a variável $< Atr >$ aparece tem subcadeias da forma "int x´´, "float c´´ e "chr y´´. Uma gramática para a linguagem C deveria levar este contexto em consideração e não permitir a derivação da cadeia $x = y + c$. Equivalentemente, para se derivar $x = y + c$ o contexto

deve apresentar subcadeias que representam declaração de variáveis com tipos compatíveis com a atribuição.

O que discutimos anteriormente é retratado pelo conceito de gramática sensível ao contexto. Uma regra gramatical sensível ao contexto procura capturar o contexto em que a variável sendo derivada aparece. Mais formalmente, a motivação para as regras sensíveis ao contexto são regras da forma $\alpha A\beta \rightarrow \alpha\gamma\beta$, que indicam que A, no contexto de α à esquerda e β à direita, pode ser entendido (reescrito) como γ. Apesar de esta ser a motivação, a definição que usaremos é mais geral na forma, mas não mais expressiva que esta.

Apesar do poder de expressão de gramáticas sensíveis ao contexto possibilitarem a formalização de mecanismos tão sofisticados de linguagens quanto a conjugação e a regência, na prática, as gramáticas sensíveis ao contexto não são utilizadas, visto que seu uso na construção de um analisador sintático acarretaria um mecanismo de análise ineficiente. Outra razão é que a derivação de uma cadeia em uma gramática sensível ao contexto não possui uma estrutura que represente tão bem a cadeia do ponto de vista hierárquico, como é o caso das árvores de análise para as linguagens livres de contexto. Na prática, compiladores usam analisadores guiados por gramáticas livres de contexto e alguns mecanismos mais sofisticados que as gramáticas sensíveis ao contexto para dar conta de análise de contexto. Se o leitor estiver curioso, pode pesquisar gramáticas de atributos para ver uma das formas tradicionais de se especificar contextos em gramáticas.

Passamos então ao estudo teórico de gramáticas e linguagens sensíveis ao contexto. Definimos uma regra sensível ao contexto como aquela na qual o tamanho do lado direito é maior ou igual ao tamanho do lado esquerdo. Gramáticas sensíveis ao contexto são aquelas nas quais todas as suas regras são sensíveis ao contexto.

Assimile

Uma gramática sensível ao contexto (GSC) é uma tupla G = (V, T, P, S), onde toda regra em P é uma regra sensível ao contexto.

Uma consequência da definição dada anteriormente é o fato de que, como ela está, gramáticas sensíveis ao contexto não geram a palavra vazia. Para corrigir esta idiossincrasia, incluímos a possibilidade de P possuir uma regra da forma $S \to \epsilon$ se S não ocorrer na parte direita de nenhuma regra. Temos então a versão definitiva para a definição de regras sensíveis ao contexto.

Assimile

Dada uma gramática G = (V, T, P, S). Uma regra sensível ao contexto é uma regra de uma das formas:

1. $\alpha \to \beta$, com $\alpha, \beta \in (V \cup T)^+$, α possuindo pelo menos uma ocorrência de variável e $|\alpha| \leq |\beta|$.

2. $S \to \epsilon$, se S é o símbolo inicial e não ocorre à direita de todas as regras da gramática.

Naturalmente uma linguagem sensível ao contexto (LSC) é aquela para qual existe uma gramática sensível ao contexto que a gere.

Assimile

L é uma linguagem sensível ao contexto se, e somente se, existe uma gramática sensível ao contexto G e $L = L(G)$

Uma das propriedades mais características de linguagens sensíveis ao contexto é o fato delas poderem gerar cadeias, com um controle triplo, em comparação com as livres de contexto, caracterizadas pelo controle duplo. Isto pode ser observado pelo exemplo a seguir.

Exemplificando

A linguagem $\{a^n b^n c^n \mid 0 < n\}$ é gerada pela gramática sensível ao contexto abaixo:

$S \to ABc$.

$AB \rightarrow AABBC$

$CB \rightarrow BC$

$Cc \rightarrow cc$

$Bc \rightarrow bc$

$Bb \rightarrow bb$

$Ab \rightarrow ab$

$Aa \rightarrow aa$

A derivação de $a^3b^3c^3$ é como abaixo:

$$S \Rightarrow ABc \Rightarrow AABBCc \Rightarrow AAABBCBCc \Rightarrow AAABBBCCc \Rightarrow$$

$$AAABBBCcc \Rightarrow \quad AAABBBccc \Rightarrow \quad AAABBbccc \Rightarrow \ldots$$

$$\Rightarrow Aaabbbccc \Rightarrow aaabbbccc$$

Note que no exemplo acima, na derivação de $aaabbbccc$, as regras poderiam ter sido aplicadas em outra ordem, que não a exibida, no entanto a mesma cadeia é derivada. Por exemplo, a partir de $AAABBCBCc$, podemos derivar $AAABBCBcc$ ao invés de $AAABBBCCc$. Então a partir da cadeia $AAABBCBcc$ só poderíamos derivar $AAABBBCcc$, enquanto que a partir da cadeia $AAABBBCCc$ só é possível derivar $AAABBBCcc$. Desta forma, podemos observar que qualquer sequência de aplicações de regras da gramática gera cadeias da linguagem em questão.

Pesquise mais

A literatura na área de linguagens formais não é muito uniforme. Isto se dá em função de certas definições terem várias alternativas equivalentes. Por exemplo, a própria definição de gramática sensível ao contexto não é única na literatura. Verifique as definições alternativas a que usamos neste texto e verifique se as diferentes definições caracterizam a mesma classe de linguagens (GARCIA, 2017; LINZ, 2012; MENEZES, 2000).

Um exemplo bem representativo do poder de expressão de gramáticas sensíveis ao contexto é a geração de palavras na forma ww, com $w \in \Sigma^*$, com $1 < |\Sigma|$. O processo de geração via gramática se assemelha neste caso a um tipo de computação que é baseada em cadeias. Mais à frente no texto discutiremos a universalidade deste tipo de computação. Por enquanto vamos analisar como funciona a gramática a seguir neste exemplo.

Exemplificando

A gramática abaixo gera a linguagem $\{ww \mid w \in \{a,b\}^*\}$:

1. $S \rightarrow RT \mid aa \mid bb$
2. $R \rightarrow RaA \mid RbB \mid aaM \mid abN \mid baO \mid bbP$
3. $Aa \rightarrow aA$; $Ab \rightarrow bA$; $AT \rightarrow Ta$
4. $Ba \rightarrow aB$; $Bb \rightarrow bB$; $BT \rightarrow Tb$
5. $Ma \rightarrow aM$; $Mb \rightarrow bM$; $MT \rightarrow aa$
6. $Na \rightarrow aN$; $Nb \rightarrow bN$; $NT \rightarrow ab$
7. $Oa \rightarrow aO$; $Ob \rightarrow bO$; $OT \rightarrow ba$
8. $Pa \rightarrow aP$; $Pb \rightarrow bP$; $PT \rightarrow bb$

As regras da linha 1 geram os menores casos aa e bb e preparam a cadeia RT que será usada para a geração de ww. As duas primeiras regras da linha 2 geram cadeias na forma $(aA+bB)^*(aaM+abN+baO+bbP)$. Esta cadeia tem no seu prefixo um terminal e uma variável do mesmo (a/A ou b/B), terminando com aaM ou abN ou baO ou bbP. Veja nas regras de 5 a 8 que M, N, O e P permutam com a´s e b´s até encontrar o T. Neste ponto, o MT, por exemplo, se reescreve em aa. Lembre-se que o M foi gerado com o aa. O processo de forma geral é permutar, com a aplicação das regras 3 e 4, A e B, até que estes passam para o lado direito de T, quando são reescritos como a e b, respectivamente. Depois disso, as variáveis M, N, O e P podem ser permutadas. Como não existe regra de permute de A sobre B, nem vice-versa,

a ordem é mantida. Um exemplo de derivação é mostrado a seguir:

$$S \Rightarrow RT \Rightarrow RaAT \Rightarrow bbPaAT \Rightarrow bbaPAT \Rightarrow bbaPTa \Rightarrow bbabba$$

Reflita

Gramáticas sensíveis ao contexto, como só possuem regras na forma $\alpha \rightarrow \beta$, onde $|\alpha| \leq |\beta|$, garantem que o problema de saber se uma cadeia pertence ou não à linguagem gerada por uma gramática G é computável. Isto é, existe um programa de computador que implementa esta decisão. Isto é, basicamente, consequência do fato de que a cada uso de regra no processo de verificação a cadeia sendo gerada nunca diminui de tamanho. Em função disso e do fato de existir uma quantidade finita de aplicações de regras em que o tamanho não aumenta, podemos concluir que w, a cadeia que se quer verificar que pertence a $L(G)$, é derivada pela gramática ou que todas as possíveis derivações vão gerar cadeias de tamanho maior que w, portanto, G não derivada w. Reflita melhor sobre esse argumento. Investigue melhor o argumento quando o tamanho do lado esquerdo da regra é igual ao tamanho do lado direito da regra.

Com a exceção de regras da forma $A \rightarrow \epsilon$, toda regra livre de contexto é uma regra sensível ao contexto. Vimos na Seção 3.2 que toda a gramática livre de contexto pode ser convertida em uma Gramática Livre de Contexto Sem Regras Nulas (GSRN) equivalente. Uma GSRN é uma gramática sensível ao contexto, portanto toda a linguagem livre de contexto é também sensível ao contexto.

Existem linguagens sensíveis ao contexto que não são livres de contexto. A linguagem $L_{abc} = \{a^n b^n c^n \mid 0 < n\}$ é um exemplo. Com relação às operações de fechamento, podemos observar que o que foi feito para a união de linguagens livres de contexto pode ser feito para a união de linguagens sensíveis ao contexto. Lembre-se que se L_1 e L_2 são linguagens sensíveis ao contexto, então existem gramáticas G_1 e G_2, tais que $L_1 = L(G_1)$ e $L_2 = L(G_2)$. Sejam S_1 e S_2 os respectivos símbolos inicias das gramáticas. Tome o cuidado de renomear as variáveis de G_1 para que não haja interseção com as variáveis de G_2. Considerando a gramática que reúne todas as regras de ambas as gramáticas mais as regras $S \rightarrow S_1 \mid S_2$, que é uma regra sensível ao contexto, temos uma gramática sensível ao contexto que gera $L_1 \cup L_2$.

Considerando a gramática sensível ao contexto L_1 anterior, e as regras $S \to S_1 S \mid S_1$, temos uma gramática que gera L_1^+. A regra $S \to S_1 S \mid S_1$ é sensível ao contexto. Pelo parágrafo acima, união de L_1^+ com $\{\epsilon\}$ é sensível ao contexto, e, de fato $L_1^* = L_1^+ \cup \{\epsilon\}$. Com isso podemos destacar as propriedades de fechamento para as linguagens sensíveis ao contexto.

Assimile

Sejam L_1 e L_2 linguagens sensíveis ao contexto. As linguagens

- $L_1 \cup L_2$, e;
- L_1^*

também são sensíveis ao contexto.

Na próxima seção iremos abordar as máquinas de Turing, que possuem um poder de computação igual ao dos computadores em geral. As máquinas de Turing podem ser usadas para resolver o problema de análise sintática para as LSC usando consumo de espaço linear em função do tamanho da cadeia a ser reconhecida (HOPCROFT; MOTWANI; ULLMAN, 2002).

Sem medo de errar

Você recebeu um novo padrão de busca de dados e foi solicitado que você o especificasse com uma gramática sensível ao contexto. Você deve escrever uma gramática sensível ao contexto que gere a linguagem $L=\{a^n b^n c 2^n, n \geq 0\}$.

A seguinte gramática satisfaz a esta condição:

$S \to S_1 \mid \epsilon$

$S_1 \to aBS_1cc \mid aBcc$

$Ba \to aB$

$aB \to ab$

$bB \to bb$

Observe que aplicando as regras para S_1 podemos gerar $(aB)^n c^{2n}$, aplicando depois a regra $Ba \rightarrow aB$ obtemos $a^n B^n c^{2n}$, finalmente, aplicando as duas últimas regras obtemos as cadeias desejadas.

Avançando na prática

Por que linguagens sensíveis ao contexto são mais complexas de se analisar sintaticamente?

Descrição da situação-problema

Você entrou em uma discussão de trabalho na sua empresa sobre o papel de linguagens sensíveis ao contexto no dia a dia. A questão levantada pela sua equipe de desenvolvimento foi o fato deles acharem que podiam fazer tudo em matéria de análise de linguagens naturais com os eficientes autômatos de pilha com pequenas sofisticações como implementação eficiente de autômatos de pilha. A discussão partiu para a existência, ou melhor, o porquê da existência de linguagens sensíveis ao contexto que não são livres de contexto. Queriam saber a razão primordial para que os autômatos de pilha não sejam capazes de analisar linguagens geradas por gramáticas sensíveis ao contexto. Você então pesquisou e descobriu que um argumento de contagem mostra que a linguagem $L_{abc} = \{a^n b^n c^n \mid 0 < n\}$ não é livre de contexto. Sua tarefa é então entender esse argumento e explicá-lo para sua equipe.

Resolução da situação-problema

Para verificarmos que não há gramática livre de contexto que gere esta linguagem, suponha que G é uma GLC que gera L_{abc}. Pelo que vimos no curso sabemos que existe uma gramática $G_1 = (V, T, P, S)$ que não possui regras simples ou nulas e que gera L_{abc}. Seja m o número de variáveis em V e k o tamanho máximo do lado direito de uma regra em P. Uma árvore de derivação de altura h terá no máximo kh folhas. Seja $q = mh + 1$, vamos considerar a sentença $w = a^q b^q c^q$. Como $w \in L_{abc}$, w tem uma árvore de derivação. Como o tamanho de w é maior que mh, a altura da árvore de derivação é ao menos m+1. Portanto, a derivação de w pode ser escrita como:

$$V_0 \Rightarrow^* \alpha_1 V_1 \beta_1 \Rightarrow^* \alpha_1 \alpha_2 V_2 \beta_2 \beta_1 \Rightarrow^* \alpha_1 \alpha_2 ... \alpha_m V_m \beta_m ... \beta_2 \beta_1 \Rightarrow^* \alpha_1 \alpha_2 ... \alpha_m \delta \beta_m ... \beta_2 \beta_1$$

Na derivação existem as m+1 variáveis $V_0,\ldots,V_m$. Portanto, há 2 dessas variáveis que são iguais. Sejam V_i e V_j essas variáveis, onde 0≤i≤j≤m. Na derivação anterior temos que:

$$V_j \Rightarrow^* \alpha_j\alpha_{j+1}\ldots\alpha_m\delta\beta_m\ldots\beta_{j+1}\beta_j .$$

Observe que, se o tamanho de $\alpha_j\alpha_{j+1}\ldots\alpha_m\delta\beta_m\ldots\beta_{j+1}\beta_j$ for maior que mh, então existe uma variável repetida depois de N_j. Portanto, se escolhermos N_j como a última variável repetida, podemos garantir que o tamanho de $\alpha_j\alpha_{j+1}\ldots\alpha_m\delta\beta_m\ldots\beta_{j+1}\beta_j$ é menor que mh.

Vamos dar nomes às cadeias $v=\alpha_i\alpha_{i+1}\ldots\alpha_j$ e $y=\beta_j\ldots\beta_{i+1}\beta_i$. Como G_1 não tem regras simples, podemos garantir ao menos que uma das cadeias, v ou y, não é vazia. Finalmente, podemos dar nomes às cadeias $u=\alpha_1\alpha_2\ldots\alpha_{i-1}$ e $z=\beta_{i-1}\ldots\beta_2\beta_1$. Com esses nomes, podemos reescrever a derivação de w como:

$$V_0 \Rightarrow^* uV_iz \Rightarrow^* uvV_jyz \Rightarrow^* uvxyz$$

Recordando que $V_i=V_J$, podemos gerar, por exemplo:

$$V_0 \Rightarrow^* uV_iz \Rightarrow^* uxz$$

Mas $uvxyz = \ w=a^qb^qc^q$, vy não é a cadeia vazia e pode conter no máximo 2 símbolos entre $\{a,b,c\}$, caso contrário o tamanho de vxy seria maior que mh. Portanto $uxz \notin L_{abc}$, o que é uma contradição, pois podemos gerar a cadeia.

Faça valer a pena

1. Considere a linguagem $L=\{a^nb^mc^n \mid m\geq n\geq 1\}$.
Assinale a alternativa verdadeira.

a) L é uma linguagem livre de contexto porque pode ser gerada pela gramática:
$S\rightarrow ABC$,
$A\rightarrow aA|\epsilon$,
$B\rightarrow bB|b$,
$C\rightarrow cC|\epsilon$

b) L é uma linguagem livre de contexto porque pode ser gerada pela gramática:
$S\rightarrow aBSC|\epsilon$,
$aB\rightarrow ab$,
$Ba\rightarrow aB$,
$bB\rightarrow bb$,
$bC\rightarrow bc$,
$cC\rightarrow cc$

c) L é uma linguagem sensível de contexto porque pode ser gerada pela gramática:
$S \to aBSC \mid \epsilon$,
$aB \to ab$,
$Ba \to aB$,
$bB \to bb$,
$bC \to bc$,
$cC \to cc$
d) L é uma linguagem sensível de contexto porque pode ser gerada pela gramática:
$S \to aBSC \mid aBC$,
$B \to BB$,
$aB \to ab$,
$Ba \to aB$,
$bB \to bb$,
$bC \to bc$,
$cC \to cc$
e) L é uma linguagem livre de contexto porque pode ser gerada pela gramática:
$S \to aBSC \mid \epsilon$,
$B \to BB$,
$aB \to ab$,
$Ba \to aB$,
$bB \to bb$,
$bC \to bc$,
$cC \to cc$

2. Recordamos que gramáticas sensíveis ao contexto são aquelas em que o tamanho do lado direito das regras é maior ou igual ao tamanho do lado esquerdo. Com a possível exceção da regra $S \to \epsilon$, se o símbolo inicial S não ocorre do lado direito de uma regra.
Assinale a gramática sensível ao contexto.
a) $S \to aA$,
$A \to \epsilon$
b) $S \to aSb$,
$S \to \epsilon$
c) $S \to aSb \mid ab$,
$S \to \epsilon$
d) $S \to aSb \mid ab$
e) $S \to aSb \mid ab$,
$Sb \to b$

3. Recordamos que a união de duas linguagens sensíveis ao contexto sempre é sensível ao contexto. Considere a linguagem L_1 gerada pela gramática:

$S \to aBSc \mid aBc$

$Ba \to aB$

$aB \to ab$

$bB \to bb$

E a linguagem L_2 gerada pela gramática:

$S \to cBSa \mid cBa$

$Bc \to cB$

$cB \to cb$

$bB \to bb$

Assinale a gramática que gera a linguagem $L_1 \cup L_2$:

a) $S \to aBSc \mid aBc \mid cBSa \mid cBa$

$Ba \to aB$

$aB \to ab$

$Bc \to cB$

$cB \to cb$

$bB \to bb$

b) $S \to aBCS \mid CBaS \mid \epsilon$

$B \to b$

$C \to c$

c) $S \to aBCS \mid CBaS$

$B \to b$

$C \to c$

d) $S \to S_1 \mid S_2$

$S_1 \to aB_1S_1c \mid aB_1c$

$B_1a \to aB_1$

$aB_1 \to ab$

$bB_1 \to bb$

$S_2 \to cB_2S_2a \mid cB_2a$

$B_2c \to cB_2$

$cB_2 \to cb$

$bB_2 \to bb$

e) $S \to S_1S_2$

$S_1 \to aB_1S_1c \mid aB_1c$

$B_1a \to aB_1$

$aB_1 \to ab$

$$bB_1 \rightarrow bb$$
$$S_2 \rightarrow cB_2S_2a \mid cB_2a$$
$$B_2c \rightarrow cB_2$$
$$cB_2 \rightarrow cb$$
$$bB_2 \rightarrow bb$$

Seção 4.2

Máquinas de Turing

Diálogo aberto

Nesta unidade já vimos linguagens e gramáticas sensíveis ao contexto. Em particular, vimos as propriedades de fechamento de linguagens sensíveis ao contexto. Na área da teoria de computação há duas questões importantes em relação a nossa capacidade de resolver problemas com computadores:

• O problema pode ser resolvido por um computador? Um problema que não pode ser resolvido pelo computador é chamado "não computável". Problemas deste tipo ocorrem frequentemente na área de processamento de linguagens.

• Quão eficientemente o problema pode ser resolvido? Um problema que pode ser resolvido pelo computador, mas não de forma eficiente, é considerável "intratável". Problemas deste tipo ocorrem frequentemente na área de otimização combinatória.

Nesta seção, vamos apresentar as máquinas de Turing. A máquina de Turing foi proposta como um modelo do que é um procedimento efetivo, ou seja, do que pode ser feito por um computador, ajudando a criar um quadro de trabalho que permite abordar o primeiro problema citado anteriormente.

Recordamos que após completar um ano em seu novo emprego, você foi muito bem avaliado pelo seu trabalho e ganhou uma promoção. Após sua promoção, você recebeu um novo padrão de busca de dados e foi solicitado que você o especificasse com uma gramática sensível ao contexto. O padrão envolve um arquivo com sintaxe complexa, mas em um primeiro momento pode ser aproximado para a linguagem $L=\{a^n b^n c 2^n, n \geq 0\}$. Na seção anterior, você criou esta gramática. Sua equipe antiga, da qual você é agora o chefe, quer seguir a linha de usar autômatos para encontrar o novo padrão especificado por uma gramática sensível ao contexto. Você deve explicar que neste caso usar a gramática para obter um Autômato Linearmente Limitado não é uma boa solução para o problema. Assim, você deve redigir um e-mail explicando porque, neste caso, a solução usando um Autômato Linearmente Limitado

pode ser muito demorada e que, neste caso particular, é possível fazer soluções mais eficientes. Esta é a chance do seu trabalho continuar a ser reconhecido internacionalmente em sua empresa.

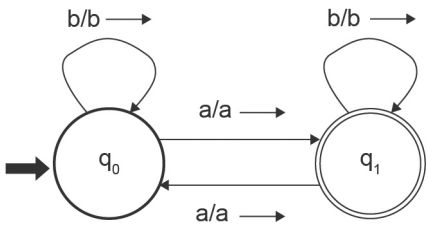
Bons estudos!

Não pode faltar

Nesta seção, vamos estudar máquinas de Turing. Assim como no caso de autômatos finitos e autômatos com pilha, veremos a máquina de Turing ser empregada para resolver o problema da análise sintática. Entretanto, ao contrário dos autômatos vistos anteriormente, a máquina de Turing foi introduzida (TURING, 1937) com o objetivo de definir o que é um procedimento, ou, nas palavras de Alan Turing, o que é uma "função computável". O leitor deve ter em mente que esta definição foi publicada antes da construção dos primeiros computadores.

Na Unidade 3, vimos que o autômato com pilha foi obtido ao se dotar um autômato finito com uma estrutura de dados auxiliar (uma pilha). Da mesma forma, podemos entender uma máquina de Turing como um autômato finito com uma estrutura de dados auxiliar, neste caso uma fita. Inicialmente, o dado de entrada é escrito na fita. A cada transição a máquina de Turing pode ler um símbolo do alfabeto da fita, pode escrever sobre o mesmo, andar na fita para direita ou para a esquerda e mudar de estado. Vamos supor que a fita possui uma primeira posição e que cada posição da fita sempre possui uma posição à sua direita, isto é, a fita é infinita para a direita. Cada posição da fita armazena um símbolo do alfabeto da fita. No início da execução da máquina, para uma entrada de tamanho n, as n primeiras posições da fita contém um símbolo da entrada. As demais posições da fita estão em branco (possuem o símbolo branco). A Figura 4.1 Ilustra uma máquina de Turing que reconhece a linguagem $L = \{w \mid w$ tem quantidade ímpar de a's $\}$. Neste caso particular a máquina de Turing anda sempre para a direita (representado pelo símbolo '$\rightarrow$'), portanto a máquina não recebe nenhuma informação da fita, a menos da cadeia de entrada, portanto a máquina de Turing, neste caso particular, tem o mesmo poder de computação que um autômato finito, o que era de se esperar, pois a linguagem reconhecida é uma linguagem regular.

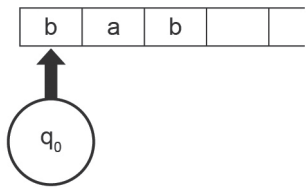
Figura 4.1 | Exemplo de máquina de Turing



Fonte: elaborada pelo autor.

O funcionamento da máquina merece uma explicação passo a passo. Suponhamos que desejamos reconhecer a cadeia 'bab'. Neste caso, a cadeia estará no início da fita. A posição da fita sendo lida pela máquina é representada pela seta vertical, e o estado no qual a máquina está é representado logo depois desta seta. A Figura 4.2 representa a configuração inicial desta máquina de Turing.

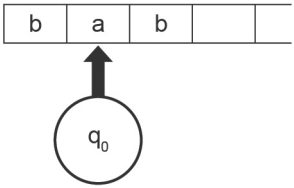
Figura 4.2 | Configuração inicial da máquina de Turing da Figura 4.1



Fonte: elaborada pelo autor.

Estando no estado q_0 e lendo um 'b', a Figura 4.1 especifica que a máquina de Turing irá escrever um 'b' por cima (portanto não vai alterar o conteúdo da fita), seguir para a direita (devido ao símbolo '$\rightarrow$') e continuar no estado q_0. Desta forma, a MT vai para a configuração descrita na Figura 4.3.

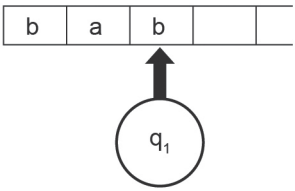
Figura 4.3 | Configuração inicial da máquina de Turing da Figura 4.1 após ler o primeiro símbolo



Fonte: elaborada pelo autor.

Neste ponto, estando no estado q_0 e lendo um 'a', a Figura 4.1 especifica que a máquina de Turing irá escrever um 'a' por cima (portanto, não vai alterar o conteúdo da fita), seguir para a direita (devido ao símbolo '$\rightarrow$') e mudar para o estado q_1. Desta forma, a MT vai para a configuração descrita na Figura 4.4.

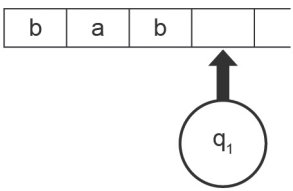
Figura 4.4 | Configuração inicial da máquina de Turing da Figura 4.1 após ler o segundo símbolo



Fonte: elaborada pelo autor.

Finalmente, estando no estado q_1 e lendo um 'b', a Figura 4.1 especifica que a máquina de Turing irá escrever um 'b' por cima (portanto não vai alterar o conteúdo da fita), seguir para a direita (devido ao símbolo '$\rightarrow$') e permanecer no estado q_1. Desta forma, a MT vai para a configuração descrita na Figura 4.5.

Figura 4.5 | Configuração inicial da máquina de Turing da Figura 4.1 após ler o terceiro símbolo



Fonte: elaborada pelo autor.

Neste caso, a Figura 4.1 não especifica o que fazer quando a MT lê um branco. Dizemos que a máquina de Turing parou. Como ela parou em um estado final (q_1), dizemos que ela reconheceu a sentença 'bab'.

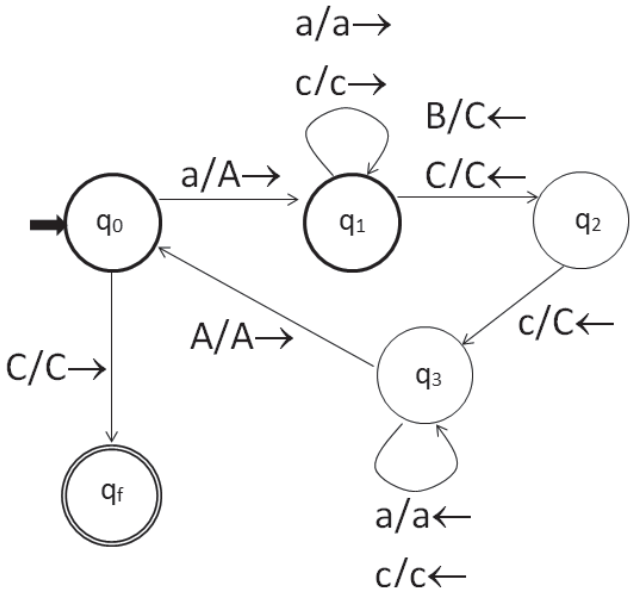
Reflita

No parágrafo anterior, dissemos que a sentença foi aceita porque a máquina de Turing parou e, quando parou, o estado era um estado final. Esta definição de aceitação está de acordo com autores como Rangel; Guedes (1997) e Linz (2012), para outros autores como Menezes (2000) e Hopcroft, Motwani e Ullman (2002) basta atingir um estado final para aceitar a cadeia. O leitor deve refletir e se convencer que estas definições não alteram o poder da máquina de Turing, bastam pequenas mudanças na máquina para converter a aceitação em um critério no outro.

Outras variações comuns são usar um marcador de início de fita, como em Menezes (2000) e Papadimitriou (1994), ou permitir que a fita seja infinita para os dois lados, como em Webber (2008), estas variações também não alteram o poder de computação da máquina de Turing.

O exemplo de reconhecimento da linguagem $L = \{w \mid w$ tem quantidade ímpar de a's $\}$ foi pouco ilustrativo porque todos os movimentos da máquina de Turing eram para a direita, fazendo com que a MT tivesse o mesmo poder de computação de um autômato finito. Apresentamos na Figura 4.6 uma máquina de Turing que reconhece a linguagem $L = \{a^n c^n \mid n \geq 1\}$ como esta linguagem não é regular a MT terá obrigatoriamente de se mover para a esquerda. A técnica usada é marcar cada caractere lido, cada 'a' lido é substituído por um 'A' e cada 'c' lido por um 'C'. O caractere branco é representado por 'B' para facilitar a visualização. Por convenção não permitimos escrever o 'B', portanto ele também é substituído por 'C'.

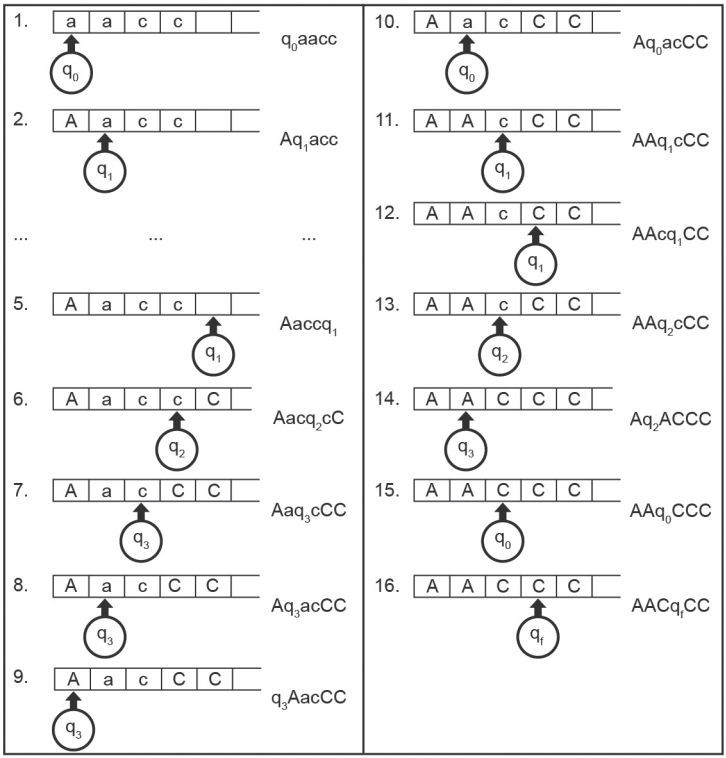
Figura 4.6 | Máquina de Turing que reconhece a linguagem $L = \{a^n c^n \mid n \geq 1\}$



Fonte: elaborada pelo autor.

Podemos representar a configuração do autômato em um dado ponto por $\alpha q \beta$, onde α é o conteúdo da fita antes da cabeça de leitura, q é o estado no qual a máquina se encontra e β é o conteúdo da fita a partir da cabeça de leitura até o primeiro caractere branco ('B'), assim a configuração inicial da máquina de Turing da Figura 4.6 ao reconhecer a cadeia 'aacc' é: '$q_0$aacc'. A Figura 4.7 ilustra o reconhecimento passo a passo desta cadeia. Do lado esquerdo está a representação vista anteriormente da configuração, enquanto que do lado direito temos a nova representação.

Figura 4.7 | Configuração passo a passo da MT da Figura 4.6 no reconhecimento de 'aacc'



Fonte: elaborada pelo autor.

Formalmente, uma máquina de Turing possui alguns elementos em comum com um Autômato Finito:

- Um conjunto finito Q de estados.
- Um alfabeto de entrada Σ, sobre o qual as cadeias a serem reconhecidas estão definidas.
- Um conjunto $F \subseteq Q$ de estados finais.
- $q_0 \in Q$, o estado inicial.

Como a máquina de Turing possui uma fita que pode ser lida e escrita, ela possui os seguintes elementos diferentes de um autômato finito:

- Um conjunto finito Γ de símbolos que podem ser escritos na fita.
- $B \in \Gamma$, o símbolo 'branco' tal que $B \notin \Sigma$ e B não pode ser escrito

na fita. A fita é infinita para a direita e é inicializada com a palavra a ser reconhecida seguida de infinitos brancos.

• δ, uma função de $Q \times \Gamma$ em $\wp(Q \times (\Gamma - \{B\}) \times \{\leftarrow, \rightarrow\})$. Ou seja, um estado e um símbolo sendo lido da fita podem estar associados a um conjunto de transições. Se este conjunto possuir sempre no máximo uma transição, dizemos que a máquina de Turing é determinística. Cada transição pertence a $Q \times (\Gamma - \{B\}) \times \{\leftarrow, \rightarrow\}$, portanto, possui um estado (estado destino), um símbolo não branco da fita (símbolo que irá sobrescrever o símbolo lido), e uma direção (esquerda ou direita) para a qual a cabeça de leitura irá se mover. Caso a cabeça se mova à esquerda do limite inicial da fita nada acontece, a máquina não vai para uma nova configuração.

Exemplificando

No caso particular da máquina de Turing da Figura 4.6 temos os seguintes elementos constituintes:

- $Q = \{q_0, q_1, q_2, q_3, q_f\}$;
- $\Sigma = \{a, c\}$;
- $F = \{q_f\}$;
- q_0 é o estado inicial;
- $\Gamma = \{a, c, a, C, B\}$;
- B é o símbolo 'branco';
- $\delta(q_0, a) = \{(q_1, A, \rightarrow)\}$;
- $\delta(q_0, C) = \{(q_f, C, \rightarrow)\}$;
- $\delta(q_1, a) = \{(q_1, a, \rightarrow)\}$;
- $\delta(q_1, c) = \{(q_1, c, \rightarrow)\}$;
- $\delta(q_1, B) = \{(q_2, C, \leftarrow)\}$;
- $\delta(q_1, C) = \{(q_2, C, \leftarrow)\}$;
- $\delta(q_2, c) = \{(q_3, C, \leftarrow)\}$;

- $\delta(q_3, a) = \{(q_3, a, \leftarrow)\}$;
- $\delta(q_3, c) = \{(q_3, c, \leftarrow)\}$;
- $\delta(q_3, A) = \{(q_0, A, \rightarrow)\}$.

Alternativamente, para simplificar a notação, representamos a função δ como quíntuplas pertencentes a $Q \times \Gamma \times Q \times \Gamma \times \{\rightarrow, \leftarrow\}$. Onde $\langle q, A, r, B, \rightarrow \rangle$ significa que $(r, B, \rightarrow) \in \delta(q, A)$. A função δ do autômato anterior pode ser representada pelas quíntuplas:

$\langle q_0, a, q_1, A, \rightarrow \rangle$; $\langle q_0, C, q_f, C, \rightarrow \rangle$; $\langle q_1, c, q_1, c, \rightarrow \rangle$; $\langle q_1, B, q_2, B, \leftarrow \rangle$; $\langle q_1, C, q_2, C, \leftarrow \rangle$; $\langle q_2, c, q_3, C, \leftarrow \rangle$; $\langle q_3, a, q_3, a, \leftarrow \rangle$; $\langle q_3, c, q_3, c, \leftarrow \rangle$; $\langle q_3, A, q_0, A, \rightarrow \rangle$

Formalmente, definimos uma máquina de Turing M como uma tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, onde os componentes foram descritos acima. Uma máquina de Turing que está em uma configuração $\alpha q a \beta$ e possui a transição $(r, x, \rightarrow) \in \delta(q, a)$, onde $q, r \in Q$, $a, x \in \Gamma, \alpha, \beta \in \Gamma^*$ se move para a configuração $\alpha x r \beta$. Neste caso, escrevemos $\alpha q a \beta \vdash \alpha x r \beta$. De forma análoga se a máquina está em uma configuração $\alpha b q a \beta$ e possui a transição $(r, x, \leftarrow) \in \delta(q, a)$, onde $q, r \in Q$, $a, x \in \Gamma$, $\alpha, \beta \in \Gamma^*$ se move para a configuração $\alpha r b x \beta$. Neste caso, escrevemos $\alpha b q a \beta \vdash \alpha r b x \beta$. Se a máquina de Turing não pode se mover em uma dada configuração, dizemos que a máquina parou nesta configuração. Chamamos $\vdash^*$ à aplicação de 0 ou mais vezes da relação $\vdash$, dizemos que uma máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ reconhece uma sentença w se $q_o w \vdash^* \alpha q_f \beta$ onde $q_f \in F$ e a máquina de Turing para na configuração $\alpha q_f \beta$. A linguagem reconhecida pela máquina de Turing é o conjunto de todas as sentenças reconhecidas pela máquina de Turing. Observe que segundo a definição, para uma máquina de Turing não determinística, basta um dos caminhos possíveis fazer a máquina de Turing parar em um estado final para que a máquina de Turing reconheça a cadeia.

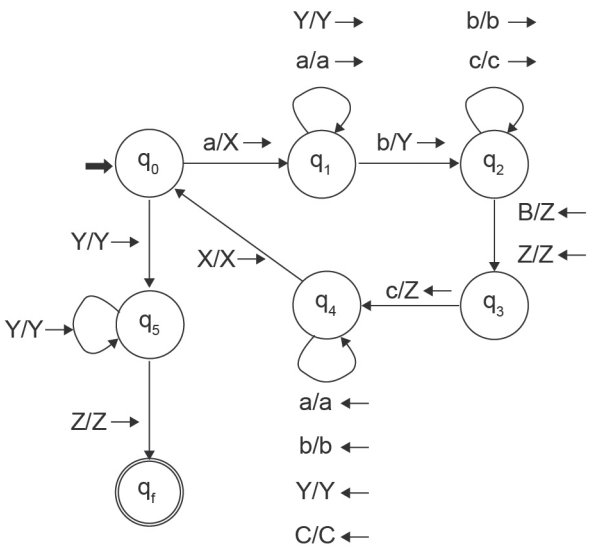
Assimile

Dada uma máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, definimos a linguagem reconhecida pela máquina de Turing como:

- $T(M) = \{w \in \Sigma^* \mid \exists q_f \in F, q_0 w \vdash^* \alpha q_f \beta$ e M para em $\alpha q_f \beta \}$

Os exemplos vistos até aqui poderiam ser reconhecidos por um autômato com pilha. A Figura 4.8 apresenta uma máquina de Turing que reconhece a linguagem $L = \{a^n b^n c^n \mid n \geq 1\}$, que não é livre de contexto.

Figura 4.8 | Máquina de Turing que reconhece a linguagem $L = \{a^n b^n c^n \mid n \geq 1\}$



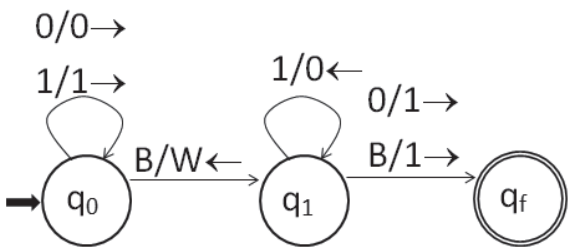
Fonte: elaborada pelo autor.

A máquina de Turing da Figura 4.8 é semelhante à máquina da Figura 4.6. Neste caso, substituímos o 'a' por 'X', o 'b' por 'Y' e o 'c' por 'Z'. O branco 'B' ao final também foi substituído por 'Z'. Ao final foi necessário criar o estado 'q_5' para verificar se sobraram símbolos entre o 'Y' e o 'Z', caso tenham sobrado símbolos, a máquina irá parar em um estado não final.

No início desta seção explicamos que a máquina de Turing foi criada como modelo do que é uma função computável. Algumas

vezes estamos interessados em usar a máquina de Turing para implementar uma função, e não para reconhecer uma linguagem. Por exemplo, podemos querer uma máquina de Turing que dada como entrada uma cadeia sobre {0,1}, representando um número na base 2, some 1 a este número. Isso significa que se a máquina começa com a representação binária do número n na fita, ela deve parar contendo na fita a representação binária de n+1. Como não permitimos escrever o símbolo branco, 'B', vamos permitir que a fita tenha um caractere não branco 'W' após o fim da representação binária de n+1. A Figura 4.9 apresenta esta máquina. Para simplificar a máquina, supomos neste caso particular a fita infinita para ambos os lados. Observe que não seria necessário usar um estado final, porque não queremos reconhecer uma linguagem, mas sim implementar uma função.

Figura 4.9 | Máquina de Turing que soma 1 ao número n



Fonte: elaborada pelo autor.

Observe que as máquinas de Turing vistas até aqui usaram apenas as casas da fita onde a entrada estava escrita, com exceção de uma casa a mais ao final da cadeia para detectarmos o final desta. Uma máquina de Turing que usa apenas as casas ocupadas pela cadeia reconhecida e uma casa à direita ou à esquerda é chamada de máquina de Turing com fita limitada, ou Autômato Linearmente Limitado. Uma linguagem pode ser reconhecida por uma máquina de Turing com fita limitada se, e somente se, é uma linguagem sensível ao contexto. A demonstração deste fato está além dos objetivos desta obra, o leitor pode encontrar uma demonstração em Hopcroft e Ullman (1969).

Pesquise mais

Vimos que uma máquina de Turing é basicamente um autômato finito que possui uma fita infinita como uma estrutura de dados auxiliar. Alternativamente, podemos definir a máquina de Turing como tendo duas (ou mais) fitas infinitas, cada qual com sua cabeça de leitura. Neste caso, o movimento da máquina de Turing é definido em função dos dados lidos em cada fita. Pesquise o motivo pelo qual essa construção não aumenta o poder da máquina de Turing, no link disponível em: <http://bit.ly/2tO72L9>. Acesso em: 13 set. 2017.

Sem medo de errar

Recordamos que na seção anterior você criou uma gramática sensível ao contexto que gera a linguagem $L=\{a^n b^n c 2^n,\ n \geq 0\}$. A gramática era:

$S \rightarrow S_1 \mid \epsilon$

$S_1 \rightarrow aBS_1cc \mid aBcc$

$Ba \rightarrow aB$

$aB \rightarrow ab$

$bB \rightarrow bb$

Sua equipe deseja usar a gramática não apenas como uma especificação, mas como subsídio para criar um autômato linearmente limitado para reconhecer L. Você deve explicar para a equipe que esta solução não é eficiente em um e-mail:

"Caros colegas,

Gostaria de chamar a atenção para o fato de que o autômato linearmente limitado, apesar de consumir uma quantidade linear de espaço, pode consumir tempo exponencialmente grande em função do tamanho da entrada. Para o caso particular do problema que enfrentamos podemos produzir um analisador sintático específico com a ajuda de um contador. Depois de contarmos o número de caracteres 'a' guardando em uma variável auxiliar inteira, por exemplo, n, fica fácil reconhecer o restante da sentença.

Abraços"

Avançando na prática

Autômato com duas pilhas

Descrição da situação-problema

Em uma reunião no seu grupo de desenvolvedores levantou-se uma questão sobre a diferença entre desempenho e poder computacional em relação à arquitetura dos autômatos (máquinas) conhecidos. O ponto-chave na discussão era se o fato de se contar com mais unidades de processamento, ou mais estruturas de dados auxiliares, pode fazer com que alguma tarefa antes não executável passasse a ser executada. Em particular, estavam interessados no poder de computação de um autômato com pilha dotado de duas pilhas.

Alguém levantou o ponto que teria lido ou estudado na universidade que uma máquina de Turing com mais de uma fita, com sua respectiva cabeça de leitura e gravação, não é capaz de executar nada a mais que sua versão com somente uma fita. A diferença em ter uma fita é apenas um desempenho melhor. Desempenho é entendido como o tempo relativo ao tamanho da entrada da máquina. Assunto de complexidade de algoritmos, que vocês concordaram que não diz respeito ao poder de expressão da máquina. Por exemplo, um autômato finito, por mais desempenho que tenha, não pode aceitar uma linguagem como $\{a^n b^n \mid 0 \leq n\}$. Isto é, o autômato finito tem um poder de expressão menor que o autômato de pilha. Outro exemplo é o caso de autômatos de pilha determinísticos que não tem o mesmo poder de expressão dos não determinísticos. Você então aceitou o desafio de pesquisar a capacidade de processamento de um autômato com duas pilhas e relatar o resultado em uma próxima reunião. Seu grupo de desenvolvimento tem um acordo que qualquer questão levantada sobre o tema de computação e construção de algoritmos deve ser pesquisada e respondida.

Resolução da situação-problema

Após algumas conversas e pesquisas na internet, você descobriu que autômatos de pilha com duas pilhas (AP2) são capazes de aceitar qualquer linguagem que uma máquina de Turing aceita.

Você preparou seu argumento para apresentar ao seu grupo.

Sua apresentação se baseia basicamente em mostrar uma configuração da máquina de Turing como a seguinte: $\alpha qa\beta$, com q o estado atual, α o que está a esquerda do símbolo a sendo lido e β o que está a direita de a. Você então combina que vai usar duas pilhas, uma com conteúdo α e outra com conteúdo $a\beta$. Usando empilhamento em uma pilha e desempilhamento em outra você é capaz de emular o funcionamento das instruções de uma máquina de Turing sobre a configuração $\alpha qa\beta$. Você lembra que na mudança de estados as máquinas de Turing e o autômato de pilha tem funcionamento idêntico. Considerando então uma instrução da máquina de Turing da forma $\langle q,a,q',b,\rightarrow\rangle$. Você associa as instruções $\langle \epsilon,q,\epsilon,a,q',a,\epsilon\rangle$, para cada símbolo x do alfabeto da máquina de Turing, no autômato de pilha com 2 pilhas esta instrução da máquina de Turing. A instrução $\langle \epsilon,q,x,a,q',ax,\epsilon\rangle$ indica que:

1. Ao ler ϵ na fita de entrada do AP2.
2. Estando no estado q.
3. Com x no topo de da fita 1.
4. Com a no topo da fita 2.
5. Muda de estado para q'.
6. Empilha a cadeia vazia na fita 2, ou seja, desempilha a da fita 2.
7. Empilha ax na pilha 1, ou seja, empilha a na fita 1.

O efeito disso é ter uma configuração do AP2 na forma $\alpha xa;q';\beta$ a partir da configuração $\alpha x;q;a\beta$, após a execução da instrução. Esta configuração $\alpha xa;q';\beta$ está relacionada à configuração $\alpha xaq'\beta$, obtida na máquina de Turing. A implementação da instrução $\langle q,a,q',b,\leftarrow\rangle$ é feita com as instruções $\langle \epsilon,q,x,a,q',\epsilon,ax\rangle$, para cada x no alfabeto da máquina de Turing.

Você também descobriu que se no autômato de pilha não se permite instruções de empilha na pilha 1 e desempilha na pilha 2, ou vice-versa, então este possui o mesmo poder computacional do autômato de pilha com somente uma pilha.

Faça valer a pena

1. Seja a máquina de Turing, escrita na forma de quíntuplas: $\langle q_1, a, q_2, A, \rightarrow\rangle$, $\langle q_2, a, q_2, a, \rightarrow\rangle$, $\langle q_2, b, q_2, b, \rightarrow\rangle$, $\langle q_2, A, q_2, A, \rightarrow\rangle$, $\langle q_2, X, q_2, X, \leftarrow\rangle$, $\langle q_2, B, q_3, a, \rightarrow\rangle$, $\langle q_1, b, q_1, b, \rightarrow\rangle$, $\langle q_3, b, q_3, b, \leftarrow\rangle$, $\langle q_3, a, q_3, a, \leftarrow\rangle$, $\langle q_3, A, q_1, A, \rightarrow\rangle$. B representa o símbolo branco. O símbolo $\square$ é um símbolo usado para demarcar o início do dado. Podemos representar as configurações da máquina como cadeias, onde os estados estarão escritos à esquerda imediata do símbolo sob o qual a cabeça de leitura e gravação da máquina está. Por exemplo, a cadeia $\square abbAaq_1aXbba$ indica uma configuração onde:

• A parte da fita que não contém símbolos brancos contém a cadeia $abbAaaXbba$.

• A máquina está no estado q_1.

• A cabeça de leitura e gravação está lendo o terceiro a da esquerda para a direita.

A máquina, quando iniciada sobre uma configuração q_0w, onde w é uma cadeia de a´s e b´s, deveria parar com uma cadeia ua^k, com k sendo a quantidade de a´s em w e u uma cadeia obtida a partir de w, que tem A onde ocorre a´s em w. A máquina está errada, entretanto. Indique a alternativa que corrige a máquina de forma a fazer o que foi dito.

a) A máquina tem muitas instruções inúteis, testes que não tem nada a ver com a computação que ela deveria fazer. Basta retirar estas instruções, por exemplo, a instrução $\langle q_2, X, q_2, X, \leftarrow\rangle$.

b) Considere a inclusão das instruções: $\langle q_0, a, q_0, a, \rightarrow\rangle$; $\langle q_0, b, q_0, b, \rightarrow\rangle$; $\langle q_0, B, q, B, \leftarrow\rangle$; $\langle q, a, q', Z, \leftarrow\rangle$; $\langle q, b, q', W, \leftarrow\rangle$; $\langle q', a, q', a, \rightarrow\rangle$; $\langle q', b, q', b, \rightarrow\rangle$; $\langle q', \square, q_1, \square, \rightarrow\rangle$; $\langle q_1, W, q_f, b, \rightarrow\rangle$.

c) Considere a inclusão das instruções: $\langle q_0, a, q_0, a, \rightarrow\rangle$; $\langle q_0, b, q_0, b, \rightarrow\rangle$; $\langle q_0, B, q, B, \leftarrow\rangle$; $\langle q, a, q', Z, \leftarrow\rangle$; $\langle q, b, q', W, \leftarrow\rangle$; $\langle q', a, q', a, \rightarrow\rangle$; $\langle q', b, q', b, \rightarrow\rangle$; $\langle q', \square, q_1, \square, \rightarrow\rangle$; $\langle q_1, W, q_f, b, \rightarrow\rangle$; $\langle q_1, Z, q_f, A, \rightarrow\rangle$.

d) Considere a inclusão das instruções: $\langle q_0, a, q_0, a, \rightarrow\rangle$; $\langle q_0, b, q_0, b, \rightarrow\rangle$; $\langle q_0, B, q, B, \leftarrow\rangle$; $\langle q, a, q', Z, \leftarrow\rangle$; $\langle q, b, q', W, \leftarrow\rangle$; $\langle q', a, q', a, \rightarrow\rangle$; $\langle q', b, q', b, \rightarrow\rangle$; $\langle q', \square, q_1, \square, \rightarrow\rangle$.

e) Deve-se refazer a máquina totalmente. Além de ter instruções inúteis algumas estão erradas, pois toda vez que um a é encontrado, depois de transformado em A, este A continua escrito até o fim da computação.

2. Seja a seguinte máquina de Turing, com estado inicial q_0 e estado final q_f, descrita como um conjunto de quíntuplas: $\langle q_0, a, q_1, a, \rightarrow\rangle$, $\langle q_1, a, q_2, a, \rightarrow\rangle$, $\langle q_0, b, q_0, b, \rightarrow\rangle$, $\langle q_1, b, q_1, b, \rightarrow\rangle$, $\langle q_2, a, q_f, a, \rightarrow\rangle$, $\langle q_2, b, q_2, b, \rightarrow\rangle$. Indique a alternativa que tem pelo menos duas cadeias que são aceitas

pela máquina do enunciado:

a) $bbababababbab$; $bbbbbbbbabbbbbabbbbbbb$; $aaabbbbbbbbbbbb$.

b) $bbbabbbbabbbbbbbbabbbbbbbbb$; $bababababababababababa$; $bbbbbbbbb$.

c) $aaabaaaabaaa$; $bbbabbbbbabbb$; $abbbbbbbabbbbbbabbbbbbbabbbbbbab$.

d) $abbbbabbbbbb$; $bbbbbbbabbbbbbbbbbbbabbbbb$; $bbbbabbbbbbabbbbbbabbb$.

e) $abababababbb$; $bababababababa$; $bbabbabbabbabba$.

3. Seja a seguinte máquina de Turing com estado inicial q_0 e estado final q_f: $\langle q_0, a, q_b, A, \rightarrow\rangle$; $\langle q_0, a, q_c, A, \rightarrow\rangle$; $\langle q_0, b, q_0, b, \rightarrow\rangle$; $\langle q_0, c, q_0, c, \rightarrow\rangle$; $\langle q_b, b, q_v, d, \leftarrow\rangle$; $\langle q_c, c, q_v, d, \leftarrow\rangle$; $\langle q_v, a, q_v, a, \leftarrow\rangle$; $\langle q_v, b, q_v, b, \leftarrow\rangle$; $\langle q_v, c, q_v, c, \leftarrow\rangle$; $\langle q_v, d, q_v, d, \leftarrow\rangle$; $\langle q_v, A, q_0, A, \rightarrow\rangle$; $\langle q_0, B, q_f, B, \rightarrow\rangle$; $\langle q_b, d, q_b, d, \leftarrow\rangle$ e $\langle q_c, d, q_c, d, \leftarrow\rangle$. O alfabeto de entrada da máquina é $\{a, b, c\}$. Observe que esta é uma máquina não determinística.

Indique a cadeia aceita pela máquina mostrada no texto da questão.

a) $abbaacbaab$.

b) $abaabaab$.

c) $abaaaacbabcb$.

d) $cbcbaaaa$.

e) $babacba$.

Seção 4.3

Máquinas de Turing e recursividade

Diálogo aberto

Nesta unidade, já vimos linguagens e gramáticas sensíveis ao contexto, bem como as máquinas de Turing. Na seção anterior, mencionamos que na área da teoria de computação há duas questões importantes em relação à nossa capacidade de resolver problemas com computadores:

• O problema pode ser resolvido por um computador?

• Quão eficientemente o problema pode ser resolvido por um computador?

Apresentamos, então, as máquinas de Turing como um modelo do que um computador pode fazer. Para terminar de responder à primeira questão, devemos, portanto, investigar o que uma máquina de Turing pode fazer e o que uma máquina de Turing não pode fazer. Nesta seção, vamos apresentar as limitações da máquina de Turing e veremos exemplos de problemas que um computador não pode resolver (linguagens não recursivas e linguagens não recursivamente enumeráveis).

Um dos membros de sua equipe programa a solução que você propôs para o novo padrão. Por algum motivo um erro não é detectado pelo setor de qualidade e o programa, ao ser enviado para o escritório em Helsinki e lá executado, entra em loop infinito. Seu chefe o pressiona devido a este problema e pede que você acrescente aos testes automatizados um teste que verifica se o programa pode entrar em loop infinito. Este problema coincide com um exemplo de linguagem não recursiva apresentada nesta seção. Você deve enviar um e-mail ao seu chefe provando que a solicitação dele é impossível.

Não pode faltar

A máquina de Turing foi definida em Turing (1937), por Alan Mathison Turing. Turing inicia seu artigo explicando que irá estudar

os números computáveis, explicando o que entende por um número Real (pertencente ao conjunto $\mathbb{R}$) computável. Turing faz isso através daquilo que chamou de "a-machines", ou "automatic machines". Uma máquina com uma fita onde se escreve e lê caracteres e uma memória finita (de estados). Apesar do nome de máquina, seu artigo define a máquina como um modelo matemático, portanto teórico.

De fato, antes de Turing, apenas Alonzo Church (1936) tinha um conceito bem definido de função computável. Turing menciona em seu artigo, provando formalmente algum tempo depois, que sua definição de computável é equivalente ao conceito de computável de Church. Alonzo Church define o -calculus, uma linguagem que possui funções como cidadãos de primeira classe e que inspirou a criação da linguagem LISP na década de 50 do século XX (MCCARTHY, 1960). A busca de um modelo do que é computável fazia parte de um desafio intelectual medonho, o programa de Hilbert. Trata-se da prova da consistência da matemática, ou da certificação que não é possível que haja provas de absurdos matemáticos, tais como a igualdade 0=1. A razão da proposição de tal desafio foge ao escopo deste texto, entretanto, parece claro dizer que uma tal prova da consistência da matemática não pode se utilizar de teorias abstratas que falam sobre objetos abstratos, tais como superfícies de dimensão superior, ou mesmo números transcendentais. Havia necessidade de uma teoria matemática que pudesse dar suporte a operações efetivas sobre objetos concretos, tais como os números naturais ($\mathbb{N}$).

Um dos objetivos do programa de Hilbert também era mostrar que todos os enunciados matemáticos deveriam ser decidíveis, no sentido que qualquer proposição na matemática deveria ser demonstrável ou refutável, não havendo espaço para dúvidas ou indeterminações. É dentro deste contexto que diversos trabalhos aparecem no início do século XX, propondo formalizações desta teoria de operações efetivas sobre objetos concretos. Além de Church e Turing, podemos citar o trabalho de Emil Post, com seus sistemas de reescrita que lembram muito o conceito de gramáticas que aprendemos aqui neste curso. Turing, assim como Church, provam que não existe um processo efetivo capaz de dada uma proposição matemática qualquer dizer se ela é verdadeira ou não. De fato, isto não foi surpresa, principalmente pelo fato de que, em 1931, Kurt Gödel ter provado que existem enunciados sobre aritmética

que são indecidíveis, ou seja, que não podem ser demonstrados nem refutados.

Após os trabalhos citados no parágrafo anterior, podemos concluir que existiam pelo menos três definições baseadas em intuições distintas para o conceito de computável, e, que todas as três provaram-se ser equivalentes. São as três possíveis respostas para a pergunta: o que é computável?

Por outro lado, os três modelos fornecem essencialmente a mesma resposta. No decorrer do século XX, outros modelos se sucederam provando-se equivalentes aos três primeiros. O conceito de computável é natural, existe na natureza. Não há como definir este conceito de forma definitiva. Existe o mesmo problema com outros conceitos, tais como tempo, espaço, movimento, força, vida. Sobre estes conceitos, podemos no máximo construir teorias que são potencialmente refutáveis. Não há como se provar que uma teoria física é verdadeira, pode-se, no máximo, oferecer evidências para a sua aceitação. Esta é a essência do que chamamos de Ciência. O conhecimento matemático, em contrapartida, não possui a mesma característica. Parece que computação é tão Ciência como Física ou Biologia (CAFEZEIRO et al., 2010).

De acordo com Davis (1982), foi Kleene que cunhou o termo tese de Church associando-o ao enunciado: uma função é efetivamente computável se, e somente se, é λ definível, o que posteriormente pôde ser enunciado como uma função é mecanicamente computável se, e somente se, é Turing-computável. Kleene, ele próprio autor do modelo de funções recursivas gerais, cunhou o mais central enunciado científico da computação, a tese de Church-Turing. A área enfim merece, sem dúvida nenhuma, o status de Ciência, Ciência natural. Vale alertar que o conceito que utilizamos de Ciência é devido a Popper e não é unânime entre os epistemólogos da Ciência.

O modelo de Turing para a computação, ou seja, a máquina de Turing tem um lugar central na história da computação (HAEUSLER, 2012). Turing é quem de fato demonstra matematicamente que é possível construir-se uma máquina programável. Cada máquina de Turing realiza uma só tarefa. Não sendo de imediato uma máquina programável. O que é uma máquina programável, senão uma máquina que lê programas e os executa. Essa intuição sobre

o programável é que fez com que Turing definisse uma máquina, chamada de máquina Universal, que executa outras máquinas de Turing, ou seja, a máquina de Turing é uma máquina que lê uma cadeia que representa uma máquina de Turing e um dado sobre o qual se deseja executar essa máquina de Turing. Essa máquina universal lê estes dados e executa a máquina lida sobre o dado. Turing definiu em detalhes esta máquina. Percebe-se imediatamente que se alguém deseja implementar uma máquina programável, deve implementar a máquina universal definida por Alan Turing.

Vamos fornecer uma ideia básica sobre como funciona a máquina universal. Escolha um alfabeto contendo os símbolos q, 1, s, $\rightarrow$, $\leftarrow$, $\#$. Com este alfabeto, e uns poucos símbolos adicionais, podemos codificar qualquer quíntupla de uma máquina de Turing. Por exemplo, se enumerarmos os estados de uma máquina como q_n, com $n \in \mathbb{N}$, lembre-se que a quantidade de estados em qualquer máquina de Turing é finita. Enumeramos também os símbolos do alfabeto da máquina de Turing que desejamos executar. Com esta convenção, uma quíntupla arbitrária é, por exemplo, $\langle q_i, s_j, q_k, s_m, \rightarrow \rangle$. Esta quíntupla é codificada como $\#q1^i s1^j q1^k s1^m \rightarrow \#$. A máquina de Turing codificada fica sendo a concatenação de suas quíntuplas, mais a informação, que pode ser concatenada ao final da cadeia de quíntuplas, de qual é o estado inicial. Pode-se convencionar que o estado inicial é simplesmente q. Na fita da máquina universal escreve-se esta máquina de Turing codificada. Ao lado desta máquina de Turing codificada, mantêm-se a configuração de execução da mesma. Isto não passa de uma cadeia na forma $\#\alpha q1^i s1^m \beta \#$, que indica que a cabeça de leitura e gravação da máquina que é simulada pela máquina universal se encontra lendo $s1^m$ no estado $q1^i$. A máquina universal realiza então ciclos em que substitui a cadeia $q1^i s1^m$ pela ação associada à mesma, buscando $q1^k s1^n$ em uma quíntupla da forma $\#q1^i s1^m q1^k s1^n \rightarrow \#$. No caso de movimento $\rightarrow$, para a direita, a cadeia que substitui $q1^i s1^m$ é $s1^n q1^k$. A execução para quando o estado final aparece na cadeia substituída. Um programador mais experiente entende bem como é a programação dos ciclos da máquina Universal.

Assimile

Existe uma máquina de Turing U, a máquina universal, escrita sobre um alfabeto capaz de codificar toda máquina de Turing com a seguinte propriedade.

• Se $c(M)$ é o código da máquina M e $c(d)$ é o código de uma cadeia d, então U, quando tem $c(M)c(d)$ na sua fita, e se M para resultando em d' então, U para tendo na fita como resultado $c(d')$.

Em outras palavras, U funciona como um interpretador e simula o que M faz ao receber d como entrada.

Vimos na definição de máquinas de Turing (MT), que estas se identificam com seus próprios conjuntos de instruções. Cada máquina de Turing aceita uma única linguagem, ou seja, existe um mapeamento entre a máquina de Turing e a linguagem que ela aceita. Este mapeamento não é injetivo, pois adicionando instruções inúteis, ou seja, que nunca serão executadas, pode-se definir uma quantidade infinita de máquinas de Turing que aceita a mesma linguagem, como é demonstrado pelo exemplo a seguir.

Exemplificando

Considere uma máquina de Turing $M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, q_f \rangle$ que aceita uma linguagem L. Seja q um estado que não é estado de M. Defina a máquina $M_q = \langle Q \cup \{q\}, \Sigma, \Gamma, \delta', q_0, B, q_f \rangle$, onde δ' estende δ incluindo $\delta(q,a) = \{(q,a,\rightarrow)\}$ para $a \in \Sigma$. Como o estado q não é alcançável a partir de q_0, pois $q \notin Q$, e temos uma infinidade de q´s que não pertencem a Q, temos, portanto, uma quantidade infinita de máquinas de Turing que aceitam L. Em se tratando de uma linguagem de programação como C, este exemplo é análogo a inserir a instrução "x=x" um número arbitrário de vezes no texto do programa. Cada inserção produz um programa distinto, mas que implementa a mesma tarefa que o programa original.

Por outro lado, toda máquina de Turing aceita um único conjunto de cadeias sobre seu alfabeto. Esta é a linguagem aceita pela máquina. Por exemplo, a máquina de Turing que não para nunca,

independentemente da cadeia que foi lida, aceita a linguagem vazia. A máquina que para em um estado final, quando lê qualquer cadeia como entrada, aceita Σ^*, onde Σ é o alfabeto de entrada da máquina.

Temos então a seguinte situação: toda máquina de Turing define uma linguagem, e existem infinitas máquinas que definem a mesma linguagem. A recíproca, ou seja, a asserção que diz que toda linguagem possui uma máquina de Turing que a aceita, não é verdadeira. Isto pode ser verificado por contagem. Cada máquina de Turing é representada por uma cadeia de símbolos, sua descrição. Você pode entender isso se observar que a cadeia formada por todas as quíntuplas de uma máquina de Turing, colocadas lado a lado, é uma representação fiel da máquina. Assim, temos que o conjunto destas cadeias que representam máquinas de Turing é infinito enumerável, e, já vimos na Unidade 1 que o total de conjuntos de cadeias sobre algum alfabeto é infinito não enumerável. Temos então mais linguagens, ou seja, conjuntos de cadeias, do que máquinas de Turing capazes de aceitá-las. De fato, existem muito mais linguagens não aceitas por máquinas de Turing do que linguagens aceitas. Vejamos algumas destas linguagens e como provar que elas não são aceitas por máquinas de Turing. Linguagens que são aceitas por máquinas de Turing são chamadas de linguagens recursivamente enumeráveis (ROGERS, 1987; HOPCROFT; MOTWANI; ULLMAN, 2002).

Diz-se que uma linguagem $L \subseteq \Sigma^*$ é recursivamente enumerável se, e somente se, existe uma máquina de Turing M, tal que, $L(M) = L$.

A denominação de linguagem recursivamente enumerável vem de uma definição equivalente que diz que uma linguagem $L \subseteq \Sigma^*$ é recursivamente enumerável se, e somente se, existe uma máquina de Turing que lista todos os elementos de L. Ou seja, existe uma máquina de Turing M, com duas fitas, uma delas sendo a fita de saída, que escreve nesta fita todos os elementos de L. Isto implica que, no caso de L ser infinita, que a máquina M não para, mas que se $\omega \in L$ então existe um número de passos $k > 0$, tal que após k passos, ω estará escrita na fita de saída. Neste caso, a separação das cadeias escritas na fita de saída é feita via um símbolo escolhido com este propósito. Com o intuito de apresentar uma versão mais precisa e mais fácil de entender desta versão para a definição de recursivamente enumerável, a literatura introduziu uma terceira definição a seguir.

Diz-se que uma linguagem $L \subseteq \Sigma^*$ é recursivamente enumerável se, e somente se, existe uma máquina de Turing M que implementa uma função sobrejetiva de $\mathbb{N}$ em L. Isto é, M aplicada a $i \in \mathbb{N}$ resulta em algum elemento de L, e, todo elemento ω de L, é tal que existe $n \in \mathbb{N}$ e M executada sobre (uma representação de) n resulta em ω. É fato que esta definição e a definição anterior são equivalentes.

Considere um alfabeto que tem condições de representar máquinas de Turing e os dados sobre os quais estas máquinas são executadas. O alfabeto binário pode ser este alfabeto. Como cada máquina de Turing é uma cadeia de quíntuplas, pode-se executar uma máquina de Turing sobre a cadeia que a representa. Em função disto, temos o seguinte exemplo de linguagem.

Exemplificando

Seja Σ_{Turing} um alfabeto escolhido para codificar máquinas de Turing. Seja $L_P = \{M\#M \in \Sigma^*_{Turing}$ e M é uma MT que não aceita $M\}$. Podemos ver que $L_P \subseteq \Sigma^*_{Turing}$. No entanto, L_P não é recursivamente enumerável.

A prova de que a linguagem L_P anterior não é recursivamente enumerável é o seguinte argumento. Suponha que existe uma máquina de Turing M_B que aceita L_P, isto é, $L_P = L(M_B)$. Como M_B é uma máquina de Turing, podemos verificar se $M_B \in L_P$, neste caso M_B tem que cumprir a especificação de L_P, isto é, M_B não aceita M_B, mas isto implica que $M_B \notin L(M_B)$. Concluímos então que $M_B \in L_P$ implica que $M_B \notin L_P$, pois $L_P = L(M_B)$. Por outro lado, a recíproca, se $M_B \notin L_P$ então $M_B \notin L(M_B)$, portanto M_B aceita M_B e daí $M_B \in L_P$. Contradição, pois nenhuma proposição pode ser equivalente a sua negação. Chamamos, neste texto, L_P de linguagem paradoxal, por ser inspirada em um paradoxo normalmente atribuído a Bertrand Russell. Veja a seguir.

Reflita

O paradoxo do barbeiro diz o seguinte: em uma cidade existe um barbeiro que barbeia todo homem que não barbeia a si mesmo, e somente estes. Este barbeiro se barbeia?

Vejamos uma analogia com a linguagem L_P citada anteriormente. Façamos a seguinte interpretação do paradoxo. Homens são interpretados como máquinas de Turing e dizer que um homem $H1$ barbeia um homem $H2$ é interpretado como M_{H1} aceita M_{H2}. Reflita em como a propriedade que define o conjunto de homens que o barbeiro barbeia está relacionada com a linguagem L das máquinas de Turing que não aceitam sua própria codificação. Este paradoxo do barbeiro, afinal, parece ser bem útil.

Um dos fatos mais curiosos sobre a computação é seu comportamento com respeito à negação. A linguagem $P = \{M \mid M \in \Sigma_{Turing}\ M$ é uma MT que aceita $M\}$ é recursivamente enumerável. Dada uma máquina de Turing M. Pode-se usar a máquina de Turing universal contendo na sua fita a cadeia formada por $M\$M$. A máquina universal então executa sobre esta entrada $M\$M$, e, caso M (a máquina q ser executada) pare quando executada sobre M (o dado), a máquina universal para também. Desta forma, temos a definição de uma máquina de Turing, que se utiliza da máquina universal, que aceita M somente quando M para ser executada sobre M. Isto prova que a linguagem P é recursivamente enumerável. Note que o conjunto das cadeias que não são máquinas de Turing é fácil de ser aceito por uma máquina de Turing. Assim, a linguagem complemento da linguagem paradoxal é recursivamente enumerável, mas a linguagem paradoxal não é. De fato, este exemplo nos diz que dado um alfabeto qualquer Σ, o conjunto das linguagens sobre Σ que são recursivamente enumeráveis não é fechado sob o complemento.

Quando uma linguagem L possui uma máquina de Turing M, tal que para toda cadeia, M para e imprime SIM ou NÃO, conforme a cadeia pertença ou não a linguagem L, dizemos que a linguagem é recursiva.

Assimile

Uma linguagem $L \subseteq \Sigma^*$ é recursiva se, e somente se, a função $F_L : \Sigma^* \to \{1,0\}$, tal que $F_L(x) = 1$, se $x \in L$, e, $F_L(x) = 0$, se $x \notin L$, é Turing-computável. Esta função F_L é conhecida na teoria de conjuntos como a função característica do conjunto L.

Se uma linguagem L é recursiva, então a máquina de Turing que implementa a função F_L aceita todos os $x \in L$, portanto L também é recursivamente enumerável. Por outro lado, se utilizarmos a máquina que implementa F_L em uma outra máquina de Turing que inverte as saídas de F_L, trocando 0 por 1 e 1 por 0, teremos uma máquina de Turing que aceita todas as cadeias que não pertencem a L. Isto prova que o complementar de L é recursivamente enumerável.

Quando temos duas máquinas de Turing, uma que aceita todas as cadeias de uma linguagem L e outra que aceita todas as cadeias que não estão em L, então, colocando as duas máquinas para rodar em paralelo teremos uma implementação em máquina de Turing da função característica F_L. Concluímos, portanto, que dada uma linguagem $L \subseteq \Sigma^*$, temos que L é recursiva se, e somente se, L e o complementar de L são recursivamente enumeráveis.

Neste caso, pela própria definição de linguagem recursiva, podemos concluir que se L é recursiva, então o seu complementar $\overline{L}$, também é, pois a implementação de $F_{\overline{L}}$ em relação à implementação de F_L é simplesmente trocar as respostas, 0 por 1 e 1 por 0.

Reflita

Na argumentação de que uma linguagem L que é recursivamente enumerável e o complemento também é recursivamente enumerável, então, L é recursiva, utilizamos uma máquina de Turing que roda as máquinas que aceitam L e $\overline{L}$, em paralelo. Reflita por que fizemos desta forma, usando execução em paralelo.

Do que foi discutido no parágrafo anterior, podemos mostrar que um problema bastante relevante em programação de máquinas de Turing, ou computadores em geral, trata-se do problema de definir

um algoritmo (máquina de Turing que sempre para) capaz de ler outro algoritmo e dizer se ele para ou não quando lê certo dado. Esse problema/linguagem é conhecido como o problema da parada.

A prova de que a linguagem paradoxal, L_P, não é recursivamente enumerável já nos diz que a linguagem das máquinas de Turing que não aceitam ao seu próprio código não é recursiva. Isso é equivalente a dizer que a linguagem das máquinas de Turing que aceitam a si mesmas não é recursiva. Vamos considerar a seguinte versão, aparentemente mais restrita, do problema da parada. Trata-se de saber se uma máquina de Turing para ou não quando lê seu próprio código. Vamos chamá-lo de problema da parada paradoxal. Suponha que exista um algoritmo (máquina de Turing, via tese de Church) que implementa uma solução ao problema da parada paradoxal. Seja P_{PP} este algoritmo. Considere uma máquina de Turing que implementa o seguinte procedimento $C(X)$, que recebe uma máquina de Turing X como entrada.

Figura 4.10 | Procedimento que para quando $X(X)$ não para

```
C (X) {
    if (P_PP(X) == 1) {
        while (true) {
            z=0
        }
        } else {
        return 1
    }
}
```

Fonte: elaborada pelo autor.

Da definição, vemos que $C(X)$ para se, e somente se, $X(X)$ não para se, e somente se X não aceita X, ou seja, a máquina de Turing que implementa $C(X)$ é um certificado de que a linguagem paradoxal é recursivamente enumerável. Contradição. O resultado é que a linguagem $L_{PP} = \{M \mid M$ é M.T. e M não para quando tem M como entrada$\}$ não é recursiva. Este é mais um feito de Turing, veja prova original em Turing (1937).

Mais um problema relevante com parada de máquinas é o problema da parada L_{Para} que é a linguagem das cadeias $M\$d$, onde M é uma máquina de Turing e d uma cadeia qualquer e M para (aceitando portanto) quando tem d por entrada. Essa linguagem não pode ser recursiva, pois se assim fosse, para saber se M para quando lê M, bastaria verificar se $M\$M$ pertence a L_{Para}.

Imagine que alguém peça que você programe um procedimento capaz de identificar qualquer programa que possua um código que imprime o número π com 10^{100} casas decimais na RAM do seu computador. Isto é um pedido feito pelo seu gerente, que precisa proteger a sua empresa deste tipo de ataque prejudicial. Vamos supor que você acha que foi capaz de escrever tal programa, chamado de S. Assim, S quando lê um programa P, é tal que $S(P)=1$ se P imprime π com 10^{100} casas decimais na RAM, e, $S(P)=0$ se P faz qualquer outra coisa que não isso. Sabemos que existe um programa que implementa a ação danosa de escrever π com 10^{100} casas decimais na RAM. Seja P_π este programa. Esta ação danosa de escrever 10^{100} caracteres na RAM não é equivalente a entrar em loop. Seja então o seguinte programa: $F(P)=P(P);P_\pi$.

Vemos que $F(P)$ ou entra em loop, se $P(P)$ entrar em loop, ou é equivalente a P_π se $P(P)$ parar. Em outras palavras, se é possível que haja um programa capaz de identificar se outro programa implementa a ação danosa de escrever π com 10^{100} casas decimais na RAM, então, é possível verificar de $P(P)$ para ou não. Este último fato já provamos ser não computável. Portanto, a identificação de programas que implementam a execução danosa em questão é não computável também.

Observe que o que P_π faz pode ser qualquer outra tarefa, desde que computável, que o argumento permanece o mesmo. Se $F(P)$ identifica este comportamento por leitura do código, então se resolve a pertinência ao problema paradoxal, que é uma versão de programas da linguagem paradoxal sobre máquinas de Turing. Este resultado é conhecido como teorema de Rice (RICE, 1953; HOPCROFT; MOTWANI; ULLMAN, 2002), e é enunciado a seguir. Acabamos de ver a demonstração do teorema.

Seja um alfabeto Σ, uma propriedade sobre Σ pode ser vista como uma linguagem sobre Σ. Por exemplo, a propriedade de ter comprimento par é a linguagem sobre Σ que só tem cadeias

de comprimento par. Outros exemplos são, dizer que a cadeia é balanceada, que tem a mesma quantidade de a´s, b´s e c´s, que começa com a e termina com x e etc. Temos duas propriedades extremas, a propriedade trivial, que toda cadeia satisfaz e a propriedade contraditória, que não é satisfeita por nenhuma cadeia. São as linguagens Σ^* e $\varnothing$, respectivamente.

Assimile

Teorema de Rice: Seja $PROP$ uma propriedade sobre cadeias sobre Σ que não é trivial nem contraditória. E seja L_{PROP} o conjunto do código das máquinas de Turing que aceitam $PROP$, $L_{PROP} = \{c(M) \mid L(M) = PROP\}$. Então L_{PROP} não é um conjunto recursivo.

Para fecharmos nosso estudo sobre linguagens recursivas e gramáticas, falta observar que toda linguagem gerada por gramática irrestrita é aceita por uma máquina de Turing e, portanto, é recursivamente enumerável. A justificativa para esse fato vem da possibilidade de usar a gramática como forma de construção de uma derivação da cadeia sob análise. Tendo a cadeia na fita de entrada de uma máquina de Turing, pode-se em outra fita se iniciar com o símbolo inicial da gramática e fazer com que a máquina de Turing aplique regra a regra nesta fita e esperar que a cadeia na fita de entrada apareça. Quando, e se, isso acontecer, tem-se a parada da máquina. Notar que esta é uma máquina que só irá parar para as cadeias da gramática.

No sentido contrário, isto é, dada uma máquina de Turing, podemos escrever uma gramática que gera uma cadeia se, e somente se, a cadeia é aceita pela máquina. A intuição por trás desta simulação de MT por gramática é escrever as configurações da MT na forma de cadeias $\alpha Q \beta$, onde Q é uma variável da gramática. Se a MT tem uma quíntupla na forma $\langle Q, \sigma, Q', \sigma', \rightarrow \rangle$ então a gramática tem uma regra na forma $Q\sigma \rightarrow \sigma' Q'$. A gramática também tem regras para gerar cadeias na forma $\alpha \$ Q_0 \alpha$, onde α é uma cadeia qualquer do alfabeto da MT e Q_0 é o estado inicial da MT. Isso fará com que $S \Rightarrow \alpha_0 \$ \alpha Q \beta$ se, e somente se, a MT quando executada com configuração inicial $Q\alpha_0$ alcança a configuração $\alpha Q \beta$. Desta forma, todas as configurações alcançáveis serão deriváveis a partir de S. Através de regras $Q_f \sigma \rightarrow Q_f$, $\sigma Q_f \rightarrow Q_f$ e finalmente

$\$Q_f \rightarrow \epsilon$, temos que α_0 pertence à linguagem aceita pela MT se, e somente se, $\alpha_0 \in L(G)$.

Concluímos, portanto, que para toda gramática irrestrita G existe uma máquina de Turing M tal que $L(M) = L(G)$, e que para toda a máquina de Turing M existe uma gramática irrestrita G tal que $L(M) = L(G)$.

A disciplina de Linguagens Formais e Autômatos se estende neste ponto para a disciplina de computabilidade, que estuda os modelos de computação, suas características e paradigmas subjacentes.

Sem medo de errar

Lembramos que o seu chefe, Sr. Hilbert, o pressiona devido a um bug em um programa entregue e pede que você acrescente aos testes automatizados um teste que verifica se o programa pode entrar em loop infinito. Você responde de acordo com o e-mail logo a seguir.

"Prezado Sr. Hilbert,

Recebi sua solicitação para executar testes automatizados para verificar se um programa entra em loop infinito. Em primeiro lugar, quero deixar claro o compromisso da equipe com a entrega de produtos com qualidade. Toda a pequena rotina implementada pelos programadores é entregue junto a um teste unitário. Antes de ser liberada uma versão de um sistema são executados testes unitários de todas as unidades do programa, mesmo as que não foram alteradas. Posteriormente, o sistema é passado para o controle de qualidade que executa manualmente um roteiro de testes. Sempre que uma mudança nos requisitos é especificada há uma mudança no roteiro de testes. E sempre que um erro é identificado o roteiro também é atualizado, para evitar a repetição do mesmo erro. Tal procedimento foi feito no problema em questão, portanto, há um teste manual no roteiro de testes para tal fim. Infelizmente, não é possível fazer o teste automatizado solicitado, pois isso significaria resolver o problema da parada, o conjunto dos programas que entram em loop para algum dado de entrada não é recursivo, portanto é impossível fazer um programa de computador que diga se um dado programa está ou não neste conjunto.

Atenciosamente,"

Avançando na prática

Linguagens recursivas

Descrição da situação-problema

Em uma discussão na sua equipe de desenvolvedores, alguém lembrou que as linguagens sensíveis ao contexto são recursivas. Afinal, se L é uma linguagem sensível ao contexto, L tem uma gramática sensível ao contexto G_L. Se ω pertence a L então, a partir do símbolo inicial de G_L, S_L, existe uma derivação de ω usando as regras de G_L. Assim, um procedimento bem escrito acabará por gerar ω, aplicando todas as alternativas de regras, gerando todas as derivações possíveis até chegar a ω. Como a cada aplicação de regra o tamanho da cadeia não diminui, e as repetições de cadeias podem ser evitadas, a quantidade de derivações a ser analisada é finita, ou seja, pode-se descartar toda a derivação que gera uma forma sentencial de tamanho maior do que ω. Se forem descartadas todas as derivações ω não pertence a L.

A partir deste comentário, vocês começaram uma discussão sobre se as linguagens recursivas não seriam justamente as sensíveis ao contexto. Linguagens recursivas, por possuírem algoritmos que retornam tanto a informação de pertinência quanto a de não pertinência, são certamente úteis. Houve, logo de início, uma desconfiança a respeito desta questão de equivalência entre sensíveis ao contexto e recursivas.

Como você era o membro mais novo e com mais lembranças do seu curso de graduação, você ficou então encarregado de fazer uma apresentação para o seu grupo com a resposta da questão sobre a identidade do conjunto de linguagens sensíveis ao contexto e recursivas.

Resolução da situação-problema

Após sua pesquisa, você concluiu que existem linguagens recursivas que não são sensíveis ao contexto. Para a sua apresentação você escolheu o seguinte planejamento.

1. Apresentar os aceitadores de linguagens sensíveis ao contexto, os ALLs (Autômatos Linearmente Limitados), que nada mais são do que máquinas de Turing que não possuem instruções que escrevam

sobre brancos. Um ALL, só se utiliza da porção da fita em que já há algo escrito, ou seja, o dado de entrada, que é uma cadeia com um certo comprimento. A denominação "linearmente", deve-se ao fato da máquina poder ter mais de uma fita. Assim, se ela tem k fitas, ela utiliza-se no máximo de kn células das fitas, onde n é o tamanho da cadeia de entrada.

2. Observar que qualquer gramática sensível ao contexto pode ser simulada em uma ALL não determinístico. Seja G uma gramática sensível ao contexto. Se G tem k regras, então define-se um ALL com $k+2$ fita, as fitas de entrada, de trabalho, e, uma fita para cada regra da gramática, para armazenar cada regra da gramática. A partir de uma cadeia ω escrita na fita de entrada, escreve-se o símbolo inicial da gramática na fita de entrada e procede-se a execução, não deterministicamente da aplicação das regras possíveis de serem aplicadas, substituindo lado esquerdo por lado direito na fita de trabalho. Testa-se então a igualdade do conteúdo da fita de trabalho com a fita de entrada. Este ciclo é repetido até que ou se produz uma cadeia igual ao conteúdo da fita de entrada na fita de trabalho, ou o conteúdo da fita de trabalho já ocupa todas as n células, onde n é comprimento de ω, e não produziu ω na fita de trabalho. O ALL não aceita ω caso todas as suas computações não aceitam ω, como é usual em autômatos não determinísticos.

3. Mostra-se que a máquina de Turing universal, aquela que executa máquinas de Turing, pode ser modificada para implementar a execução de qualquer ALL, de forma a retornar dois resultados, 1 quando o ALL sendo executado aceita a entrada ω, ou, escreve 0 na fita indicando que todas as computações da máquina universal não aceitam ω. Isto é realizado usando-se o mecanismo de implementação de máquinas de Turing não determinísticas por máquina de Turing determinísticas, já comentada no curso. Apesar de ineficiente, este mecanismo consegue sempre decidir se um ALL dado como entrada aceita ou não uma cadeia qualquer que também é fornecida como entrada.

4. Finalmente, chega-se ao ponto alto da apresentação. Você então mostra a linguagem $L_{ALLp} = \{\mathcal{A} \mid \mathcal{A}$ é um ALL que não aceita seu próprio código $cod(\mathcal{A})\}$. Por um argumento semelhante ao do exemplo da linguagem paradoxal, L_{ALLp} não pode ser aceita por ALLs, pois se fosse, existiria um ALL, o aceitador de L_{ALLp} que aceita seu

próprio código se, e somente se, não aceita seu próprio código. L_{ALLp} é também uma linguagem paradoxal.

Figura 4.11 | Hierarquia de capacidade de diferentes autômatos

Fonte: elaborada pelo autor.

5. Conclua comentando que L_{ALLp} não é aceita por ALL, mas é recursiva, de acordo com o argumento mostrado no item 3. Sendo assim, existem linguagens recursivas que não são sensíveis ao contexto.

6. A classe das linguagens sensíveis ao contexto é do ponto de complexidade o conjunto das linguagens que são aceitas com o uso de espaço linear. Esta classe é denotada por $NLIN$, ou $NSPACE(n)$. Sabe-se da existência de muitos problemas que necessitam de espaço maior que linear para serem resolvidos em uma máquina de Turing não determinística. Por exemplo, $NPSPACE(= PSPACE)$, $EXPTIME$ etc. No entanto, é um problema em aberto saber se $NSPACE(n)$ é igual ou não a $DSPACE(n)$.

Faça valer a pena

1. Se $c(M)$ é o código da máquina de Turing M, definimos que as linguagens $L_1 = \{c(M) \mid L(M)$ é regular $\}$ e $L_2 = \{c(M) \mid L(M)$ é livre de contexto $\}$.

Assinale a alternativa verdadeira.

a) L_1 e L_2 são recursivas.
b) L_1 é recursiva e L_2 não é recursiva.
c) $L_1 \subseteq L_2$.
d) $L_2 \subseteq L_1$.
e) $L_1 \cap L_2$ e $L_1 \cup L_2$ são recursivas.

2. Seja $c(M)$ o código de uma máquina de Turing M.

Assinale a linguagem recursivamente enumerável.

a) $L = \{c(M)w \mid$ M não para quando começa com a entrada w $\}$.
b) $L = \{c(M)w \mid$ M para quando começa com a entrada w $\}$.
c) $L = \{c(M_1) \mid L(M_1) = \varnothing\}$.
d) $L = \{\langle c(M_1), c(M_2), c(M_3)\rangle \mid L(M_1) = L(M_2)L(M_3)\}$.
e) $L = \{\langle c(M_1), c(M_2)\rangle \mid L(M_1) \cap L(M_2) = \varnothing\}$.

3. Seja L_1 uma linguagem definida sobre o alfabeto $\Sigma = \{a, b\}$ tal que L_1 é recursivamente enumerável, mas L_1 não é recursiva.

Considere a linguagem $L_2 = \{aw \mid w \in L\} \cup \{bw \mid w \notin L\}$.

Assinale a alternativa verdadeira.

a) L_2 é necessariamente recursiva.
b) L_2 pode ser recursiva, dependendo da linguagem L_1.
c) L_2 é necessariamente recursivamente enumerável.
d) L_2 pode ser recursivamente enumerável, dependendo da linguagem L_1.
e) L_2 não pode ser recursivamente enumerável.

Referências

CAFEZEIRO, I. et al. Recontando a computabilidade. **Revista Brasileira de História da Ciência**, Rio de Janeiro, v. 3, n. 2, p. 231-251, dez. 2010.

CHURCH, A. A Note on the Entscheidungsproblem. **The Journal of Symbolic Logic**, v. 1, p. 40-41, mar. 1936.

DAVIS, M. Why Gödel didn't have church's thesis. **Information and Control**, v. 54, n. 1, p. 3-24, 1982.

GARCIA, A. **Linguagens regulares e livres de contexto**. 1. ed. Rio de Janeiro: Edição do Autor (eBook Kindle), 2017.

GÖDEL, K. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. **Monatshefte für Mathematik und Physik**, Viena, v. 38, p. 173–98, 1931.

HAEUSLER, E. H. A celebration of Alan Turing's achievements in the year of his centenary. **International Transactions in Operational**, v. 19, n. 3, p. 489-491, maio 2012.

HOPCROFT, J.; MOTWANI, R.; ULLMAN, J. **Introdução à teoria de autômatos, linguagens e computação**. 1. ed. Boston: Elsevier, 2002.

HOPCROFT, J.; ULLMAN, J. **Formal languages and their Relation to automata**. Reading: Addison-Wesley, 1969.

LINZ, P. **An introduction to formal languages and automata**. Burllington: Jones & Bartlett Learning, 2012.

MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. **Communications of the ACM**, v. 3, n. 4, p. 184-195, 1960.

MENEZES, P. B. **Linguagens Formais e Autômatos**. Porto Alegre: Sagra Luzzatto, 2000.

PAPADIMITRIOU, C. H. **Computational complexity**. 1. ed. Reading: Addison Wesley Longman, 1994.

RANGEL, J.; GUEDES, L. Linguagens Formais e Autômatos - INF1626. **Apostila de Linguagens Formais de Autômatos**, 1997. Disponivel em: <http://www.tecmf.inf.puc-rio.br/LFA>. Acesso em: 21 jul. 2017.

RICE, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. **Transactions of the American Mathematical Society**, v. 74, p. 358-366, 1953.

ROGERS, H. **Theory of recursive functions and effective computability**. Cambridge: MIT Press, 1987.

TURING, A. M. On Computable Numbers, with an Application to the Entscheidungs problem. **Proceedings of the London Mathematical Society**, Londres, p. 2230-265, 1937.

WEBBER,. **Formal language, a practical Introduction**. 1. ed. Wilsonville: Franklin, Beedle & Associates, 2008.

Anotações

Anotações

ISBN 978-85-522-0189-2