FICHA CATALOGRÁFICA
Dados Internacionais de Catalogação na Publicação (CIP)
Tuxped Serviços Editoriais (São Paulo - SP)

F491o Finbow, Thomas; Lopes, Marcos; Viotti, Evani (org.).

Objetos Linguísticos: análises em perspectiva /
Organizadores: Thomas Finbow, Marcos Lopes e Evani Viotti. –
1. ed. – Campinas, SP : Pontes Editores, 2024; figs.; tabs.; quadros.
E-book: 7 Mb; PDF.

Inclui bibliografia.
ISBN 978-85-217-0351-8.

1. Gramática. 2. Língua Portuguesa. 3. Linguística. I. Título. II. Assunto. III. Organizadores.

Bibliotecário Pedro Anizio Gomes CRB-8/8846

Índices para catálogo sistemático:

1. Linguística. 410

2. Linguagem / Línguas – Estudo e ensino. 418.007

3. Língua portuguesa. 469

(Organizadores) Thomas Finbow
Marcos Lopes
Evani Viotti

OBJETOS LINGUÍSTICOS

Análises em Perspectiva

CAPÍTULO 6

Modelos de Gramática Formais: Sintaxe, Semântica e Implementação Computacional

Marcelo Ferreira
Universidade de São Paulo

Marcos Lopes
Universidade de São Paulo

1. Introdução

Existe razoável consenso sobre a capacidade de todo falante de reconhecer as expressões bem formadas e dotadas de sentido de sua língua. Os métodos formais de análise linguística têm como parte de sua missão explicitar essa capacidade. Este capítulo se propõe a estabelecer uma sequência heurístico-interpretativa que se inicia com a análise sintática, passa pela semântica e chega a uma implementação computacional de expressões linguísticas de um fragmento restrito do português, todas simples e de fácil entendimento, mas incorporando algumas das propriedades cruciais das línguas naturais, como a composicionalidade e a recursividade. A análise do fragmento escolhido como exemplo está centrada em conectivos sentenciais e em sentenças complexas formadas a partir deles.

A fim de compreender o que são e como funcionam as gramáticas formais, é necessário revisar um conjunto de noções fundamentais ligadas à sintaxe e à semântica das línguas naturais. A primeira parte do texto é voltada à geração das estruturas sintáticas recursivas a partir de um vocabulário e de regras explícitas. Mostraremos como, a partir de um formalismo bastante simples, com uma gramática de constituintes

e um léxico propositadamente limitado, é possível gerar um número potencialmente infinito de sentenças.

A segunda parte trata da interpretação dessas estruturas de maneira composicional, relacionando significado e verdade, bem como de relações semânticas entre as sentenças, como o acarretamento, a consistência e a inconsistência. Nessa parte, veremos como uma semântica de condições de verdades baseada em valorações pode ser facilmente formalizada com ferramentas tradicionais da lógica proposicional.

Na terceira parte, será apresentada uma implementação computacional na linguagem de programação Python, que é conhecida por sua simplicidade, mesmo para quem não tem qualquer experiência prévia com programação. Através da implementação, visamos a emular aspectos de nossa competência sintática e semântica e a testar algumas intuições sobre as interpretações geradas.

Ao final, proporemos uma avaliação crítica das possibilidades geradas por essa sequência heurística formal e indicaremos algumas possíveis extensões e reformulações para o prosseguimento das atividades de aprendizado e de pesquisa relacionadas.

2. Sintaxe

A sintaxe estuda como as palavras se organizam para formar sentenças, sendo parte fundamental da gramática de uma língua. Uma sintaxe formal especifica explicitamente, com recursos lógico-simbólicos, a estrutura interna das sentenças de uma língua, discriminando entre estruturas bem formadas e, portanto, pertencentes à língua, e estruturas mal formadas, não pertencentes à língua. Falantes competentes são capazes de, rápida e inconscientemente, discriminar entre sequências bem formadas e sequências mal formadas de sua língua materna. Falantes de português, por exemplo, não hesitariam em diferenciar (1) de (2) nesses termos:

(1) O cachorro latiu.

(2) *Cachorro latiu o.

Nesta seção, olharemos para dois aspectos fundamentais do conhecimento gramatical subjacente a essa competência, para, em seguida, apresentar uma gramática formal que incorpora, por assim dizer, um fragmento do português.

2.1. Estruturas sintagmáticas

À primeira vista, sentenças são sequências de palavras. Reconhecemos (3) como uma sentença do português formada por três palavras, sendo *Pedro* a primeira, *viu* a segunda e *Maria* a terceira:

(3) Pedro viu Maria.

Trocando ao menos uma palavra ou alterando a ordem entre pelo menos duas delas, obtemos sentenças diferentes:

(4) Pedro viu Joana.

(5) Maria viu Pedro.

Mas seriam sentenças meras sequências, caracterizáveis unicamente por suas palavras e a ordem em que estão dispostas? Vejamos um argumento, ao mesmo tempo simples e poderoso, de que esse não é o caso, ou seja, de que sentenças são mais do que uma lista ordenada de palavras. Para tanto, voltemos aos nossos três exemplos iniciais. Sentenças têm *significado*. As sentenças (3), (4) e (5) têm significados diferentes, e isso não é nada surpreendente. Afinal, ao passarmos de (3) para (4), alteramos uma palavra, e, ao passarmos de (4) para (5), alteramos sua ordenação. Sendo palavras e ordem linear os ingredientes que caracterizam uma sentença, seria de se esperar que alterar ao menos um deles possa automaticamente alterar o significado da sentença em questão. Considere-se, agora, (6):

(6) Pesquisadores testam cobaias com vírus.

Essa sentença é ambígua, ou seja, possui mais de um significado. Ela pode significar que pesquisadores testaram cobaias portadoras de vírus, sem nada dizer sobre a natureza do teste, ou que pesquisadores usaram vírus para testar cobaias, sem dizer nada sobre o estado das cobaias antes do teste. Como explicar essa ambiguidade? Note que estamos lidando com uma mesma sequência. Não fizemos, como nos exemplos anteriores, nenhuma substituição de palavras, nem alteração em sua ordem. Note, ainda, que não se trata de um caso de ambiguidade lexical, ou seja, de uma palavra com mais de um significado, como se vê em (7), a seguir:

(7) Pedro está na frente de um banco.

Nesse caso, sendo *banco* uma palavra lexicalmente ambígua, por causa dos diferentes tipos de "banco" que ela pode designar (um assento, um prédio que abriga uma instituição financeira, uma extensão geográfica constituída de pedra, areia ou gelo etc.), é de se esperar que sentenças que a contenham herdem essa ambiguidade. O léxico (vocabulário) do português contém dois itens homônimos, ou seja, dois itens com a mesma pronúncia e grafia, mas com significados diferentes. Nada muito surpreendente, portanto. Mas, como já dissemos, com (6) a situação é diferente, uma vez que não há ocorrência de homonímia na sentença.

Um breve momento de reflexão sobre os significados de (6) nos revela que, em uma das interpretações, a sequência *com vírus* está relacionada ao substantivo *cobaias*, ao passo que, na outra interpretação, a mesma sequência está relacionada ao predicado verbal *testam cobaias*. Vamos representar essa intuição delimitando com colchetes as sequências internas à sentença:

(8) Pesquisadores testam cobaias com vírus.
   a. [Pesquisadores [testam [cobaias com vírus]]]
   b. [Pesquisadores [[testam cobaias] com vírus]]

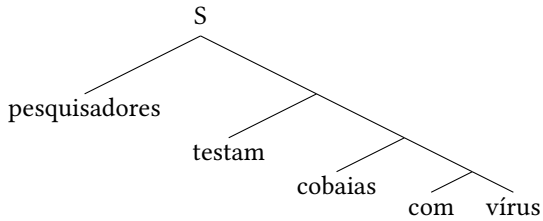
Como indicado pelos colchetes, em (8a), *com vírus* se combina com *cobaias* formando uma unidade, à qual o verbo *testam* se junta posteriormente, formando o predicado *testam cobaias com vírus*. Igualmente como indicado pelos colchetes, em (8b), primeiro

cobaias se junta com *testam*, formando uma unidade, à qual a sequência *com vírus* se junta posteriormente. Em ambos os casos, o resultado é a mesma sequência, *testam cobaias com vírus*, que se combina com o substantivo *pesquisadores*, formando uma sentença.

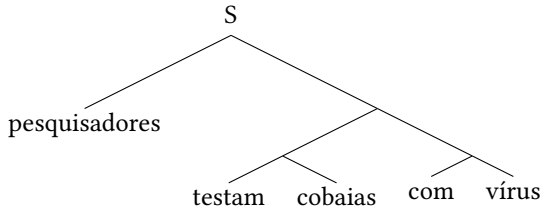
Essas estruturas indicadas pelos colchetes podem ser também vistas em forma de árvores, em que os galhos correspondem aos constituintes hierarquizados:

(9) Pesquisadores testam cobaias com vírus.

a.



b.



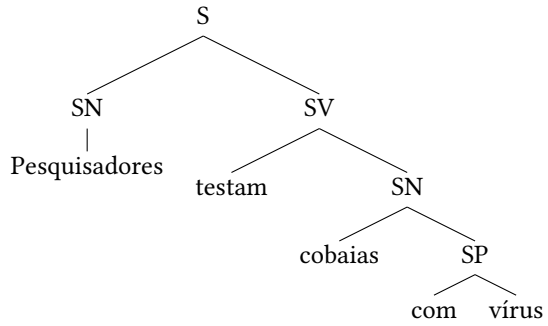
A ideia de fundo, portanto, é que sentenças não são meras sequências de palavras mas, sim, estruturas hierárquicas que contemplam, além dessas palavras, constituintes intermediários. Como se vê em (8), uma mesma sequência de palavras pode estar atrelada a mais de uma estrutura sintática, fenômeno conhecido por **ambiguidade estrutural**, em oposição à *ambiguidade lexical* vista em (7). Em suma, para caracterizar uma sentença, não basta conhecer suas palavras e a ordem em que são pronunciadas (ou escritas). É preciso conhecer também a maneira como essas palavras se estruturam internamente à sentença.

Constituintes sentenciais são conhecidos como **sintagmas** e podem ser de vários tipos: **nominais** (como *pesquisadores*, *cobaias com vírus*), **verbais** (*testam cobaias*) e **preposicionais** (*com vírus*), entre outros. Podemos, inclusive, dar rótulos a esses diferentes tipos: SN para sintagma nominal; SV para sintagma verbal etc. Isso nos leva a representações como aquelas em (10) e suas versões arbóreas em (11):

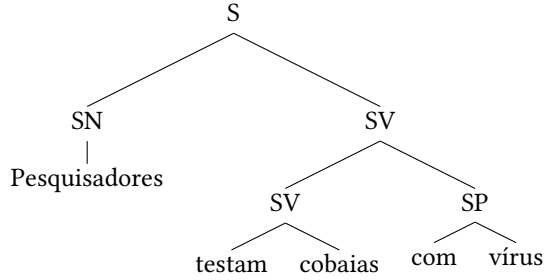
(10) Pesquisadores testam cobaias com vírus.

a. [$_{S}$ [$_{SN}$ Pesquisadores] [$_{SV}$ testam [$_{SN}$ cobaias [$_{SP}$ com vírus]]]]
b. [$_{S}$ [$_{SN}$ Pesquisadores] [$_{SV}$ [$_{SV}$ testam cobaias] [$_{SP}$ com vírus]]]

(11) a.

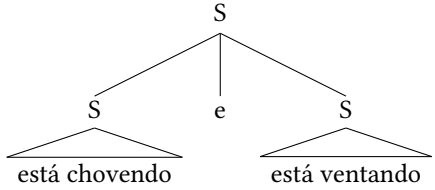


b.



Por fim, observe que mesmo sentenças podem desempenhar o papel de constituintes intermediários, quando aparecem como parte de outra sentença maior:

(12) Está chovendo e está ventando.
[$_S$ [$_S$ está chovendo] e [$_S$ está ventando]]



A existência de constituintes sentencias intermediários conduzirá a um ponto importante da sintaxe das línguas naturais: a *recursividade*.

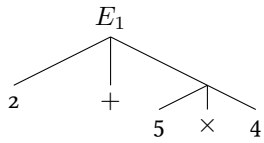
2.2. Recursividade

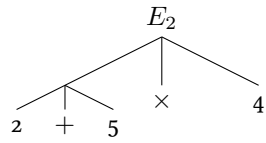
O que vimos na seção anterior sobre as sentenças do português guarda uma notável semelhança com expressões aritméticas, que expressam operações de soma ($+$), subtração ($-$), multiplicação ($\times$) e divisão ($\div$) de números naturais (0, 1, 2, ...). Considere, por exemplo, as duas expressões a seguir:

(13) a. $(2 + (5 \times 4))$
b. $((2 + 5) \times 4)$

Note que se trata de expressões diferentes, apesar de ambas serem formadas pelo mesmo "vocabulário" (os números e as operações) dispostos na mesma ordem: 2, $+$,

5, ×, 4. A diferença, claro, está nos parênteses. No primeiro caso, efetuamos primeiro a multiplicação e, depois, a adição. No segundo caso, efetuamos primeiro a adição e, depois, a multiplicação. Os resultados ('interpretações') são distintos: 22 e 28, respectivamente. Assim, expressões aritméticas são, como as sentenças do português, estruturas hierárquicas. Podem até ser representadas de forma arbórea, se quisermos:

(14) a. 

b. 

Mas há um segundo aspecto que aproxima as sentenças do português das expressões aritméticas. Se nos perguntarmos quantas expressões aritméticas existem, a resposta é, naturalmente: infinitas. E isso não se deve apenas ao fato de os números naturais serem infinitos. Mesmo que nos limitássemos aos primeiros dez números naturais (0 a 9, portanto), as expressões possíveis ainda continuariam infinitas. Isso porque qualquer expressão, por mais longa que seja, pode ser parte de uma expressão maior. Em outras palavras, apesar de termos apenas quatro operações fundamentais, essas operações podem subordinar-se a outras, inclusive do mesmo tipo. Podemos ter adições dentro de adições, multiplicações dentro de multiplicações etc. Em (15), por exemplo, construímos uma nova expressão a partir das expressões do exemplo anterior:

(15) $((2 + (5 \times 4)) + ((2 + 5) \times 4))$

O fato a se notar é que algo semelhante ocorre com o português. Sintagmas nominais, por exemplo, podem subordinar-se a sintagmas nominais:

(16) filho de professor
filho de filho de professor
filho de filho de filho de professor
...

Veja que, mesmo com um vocabulário de apenas três palavras, temos uma sequência infinita de sintagmas nominais, todos eles passíveis de ser integrados em uma sentença da língua:

(17) Um filho de filho de filho de professor acabou de nascer.

Obviamente, à medida que as sentenças vão ficando mais longas, elas vão se tornando mais e mais difíceis de pronunciar e interpretar. Isso, porém, é uma limitação da capacidade humana de processamento e memória. Do ponto de vista estritamente gramatical, são sequências impecáveis. Aliás, também nesse quesito, o paralelo com a aritmética se mantém. As operações requerem mais lápis e papel à medida que vão se tornando muito longas.

Como já antecipamos ao final da seção anterior, sentenças, elas mesmas, podem ser partes de outras sentenças, o que nos leva a um paradigma semelhante ao que acabamos de ver com os sintagmas nominais:

(18) Está chovendo.
Ele disse que está chovendo.
Ele disse que ele disse que está chovendo.
...

Note o encadeamento de subordinações sentenciais:

(19) [$_S$ Ele disse que [$_S$ ele disse que [$_S$ está chovendo]]]

Como consequência, assim como no caso das expressões aritméticas, o número de sentenças do português é infinito, de modo que não faria sentido perguntar, por exemplo, qual a sentença mais longa da língua. Para qualquer sentença imaginada, é sempre possível construir outra mais longa tendo a anterior como parte.

A essa propriedade de gerar infinitas expressões a partir de um número finito de elementos, dá-se o nome de **recursividade**. Sempre se coloca a possibilidade sintática de 'encaixar' uma expressão de um certo tipo X em outra do mesmo tipo, como esquematizado a seguir:

(20) [$_X$ ... [... [$_X$ ...]]]

Estamos prontos, agora, para entender o que é uma gramática formal.

2.3. Uma gramática formal

Formalizar uma gramática significa representar explicitamente, através de ferramentas oriundas da lógica, todas as regras de formação de constituintes, ou seja, especificar todas as estruturas possíveis para todos os tipos de sentenças de uma língua.[1]

A gramática de uma língua natural, como o português, pode ser assustadoramente complexa. Formalizá-la é uma tarefa extremamente laboriosa. Mesmo no caso de sentenças simples, o sintaticista precisa estar atento a uma grande diversidade de fatores, incluindo tipos sentenciais, ordem de palavras, concordância, todas as classes gramaticais (nomes, adjetivos, artigos, verbos, preposições, advérbios...), incluindo diversos subtipos (verbos transitivos e intransitivos, nomes contáveis e não-contáveis, artigos definidos e indefinidos etc.).

Como nosso objetivo neste texto não é, nem de longe, formalizar a gramática do português em sua totalidade, iremos, a partir de agora, voltar nossa atenção a um pequeníssimo fragmento dessa gramática, ao qual vamos ainda impor algumas simplificações. Entretanto, nossa escolha é criteriosa, já que o fragmento ilustra com clareza os dois aspectos sintáticos já apresentados nesta seção e que são essenciais a todas as línguas naturais: a estrutura sintagmática e a recursividade.

---

[1] A figura mais influente em sintaxe formal é o linguista americano Noam Chomsky. Uma de suas obras mais conhecidas é Chomsky (1957). Para uma introdução acessível, ver Larson (2010).

O fragmento que escolhemos engloba sentenças declarativas complexas formadas pela expressão negativa *não é verdade que* e pelos conectivos *e* e *ou*. Seguem alguns exemplos, já com os colchetes rotulando os constituintes sentenciais:

(21) a. [$_S$ não é verdade que [$_S$ está chovendo]]
b. [$_S$ [$_S$ está chovendo] e [$_S$ está ventando]]
c. [$_S$ [$_S$ está frio] ou [$_S$ não é verdade que [$_S$ está ventando]]]
d. [$_S$ [$_S$ está chovendo] ou [$_S$ [$_S$ está frio] e [$_S$ está ventando]]]

Como se pode notar, todas essas sentenças são formadas a partir de três sentenças simples:

(22) a. [$_S$ Está chovendo]
b. [$_S$ Está ventando]
c. [$_S$ Está frio]

Como não vamos lidar com a estrutura interna dessas sentenças simples, isto é, não vamos analisar separadamente os sintagmas componentes da sentença, podemos tratar as sentenças completas como objetos atômicos, sem partes menores. Isso, claro, é uma simplificação. No mínimo, notamos nessas sentenças verbos auxiliares flexionados em terceira pessoa do singular e desinências de gerúndio nos verbos principais que os acompanham. Aqui, vamos deliberadamente passar por cima dessas questões.

Olhando para as estruturas em (21), todos os constituintes sentenciais complexos são como em (23)-(25):

(23) [$_S$ S e S]

(24) [$_S$ S ou S]

(25) [$_S$ não é verdade que S]

A cada uma dessas estruturas, corresponde uma regra gramatical. O formalismo mais usado para representar essas regras é o seguinte:

(26) S → S e S

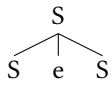
(27) S → S ou S

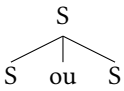
(28) S → não é verdade que S

A regra em (26) deve ser lida da seguinte forma: uma sentença pode ser formada por uma sentença, seguida pela conjunção *e*, seguida por uma sentença. Ou, se você preferir: uma sentença pode ser formada pela conjunção *e* ladeada por duas outras sentenças. O mesmo vale para a regra (27), trocando-se o *e* pelo *ou*. Já (28) nos diz que uma sentença pode ser formada pela sequência *não é verdade que* seguida por uma sentença. Para simplificar, vamos representar essa sequência por *neg*, evitando analisar sua estrutura interna. Sendo assim, reescreveremos (28) como (29):

(29) S → neg

Note que (26), (27) e (29) contêm mais de um símbolo após a seta. Isso indica que são regras que introduzem ramificação na estrutura. As regras (26) e (27) são regras ternárias, enquanto (29) é uma regra binária. Podemos visualizar suas contribuições para a estrutura sentencial da seguinte forma:

(30) a. 

b. 

c. 

Perceba o caráter recursivo dessas três regras: são sentenças sendo formadas por outras sentenças.

Por fim, precisamos de regras para nossas sentenças simples em (22). Como trataremos essas sentenças como se fossem formadas por um único item indivisível, vamos representá-las com regras unárias, que não introduzem ramificação na estrutura. Usaremos, para simplificar, as letras minúsculas c, para *está chovendo*, v, para *está ventando* e f para *está fazendo frio*. Estas são as três novas regras:

(31) $\text{S} \rightarrow \text{c}$

(32) $\text{S} \rightarrow \text{v}$

(33) $\text{S} \rightarrow \text{f}$

Como se pode ver, diferentemente das regras anteriores, essas três regras contêm apenas um elemento após a seta.

Temos, então, seis regras, que vamos agrupar em um conjunto G. Essa será a representação formal da gramática do nosso fragmento. Para facilitar a referência posterior a essa regras, vamos nomeá-las R_1-R_6:

(34) ***Gramática G*:**

R_1: $\text{S} \rightarrow \text{S e S}$
R_2: $\text{S} \rightarrow \text{S ou S}$
R_3: $\text{S} \rightarrow \text{neg S}$
R_4: $\text{S} \rightarrow \text{c}$
R_5: $\text{S} \rightarrow \text{v}$
R_6: $\text{S} \rightarrow \text{f}$

O símbolo S constitui o **vocabulário não-terminal** da gramática. É o único símbolo que aparece do lado esquerdo das regras. Já os símbolos *e*, *ou*, *neg*, *c*, *v* e *f* constituem o **vocabulário terminal** da gramática, só aparecendo do lado direito das regras. Serão sempre os elementos mais baixos da estrutura sentencial.

(35) Vocabulário não-terminal de G: {S}

(36) Vocabulário terminal de G: {e, ou, neg, c, v, f}

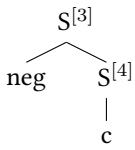
Uma gramática como G pode ser entendida como um *dispositivo gerador de sentenças*, sendo, por isso mesmo, chamada de **gramática gerativa**. Que sentenças ela gera? A ideia é começar com o símbolo S, que representa as sentenças, e ir percorrendo as

regras, da esquerda para a direita, substituindo os símbolos que forem aparecendo do lado direto da seta, até que só restem elementos do vocabulário terminal. Esse percurso é chamado de **derivação sintática**. Para exemplificar, retomemos as seguintes sentenças:

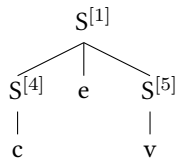
(37) a. Não é verdade que está chovendo
b. Está chovendo e está ventando.
c. Está fazendo frio ou não é verdade que está ventando.

Representaremos a seguir suas estruturas arbóreas, já com as sentenças simples substituídas pelas respectivas letras. Para facilitar, vamos rotular cada galho da árvore com o número da regra correspondente àquele momento da derivação:

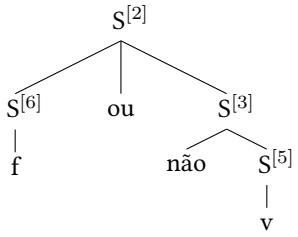
(38) a. Não é verdade que está chovendo



b. Está chovendo e está ventando



c. Está frio ou não está ventando



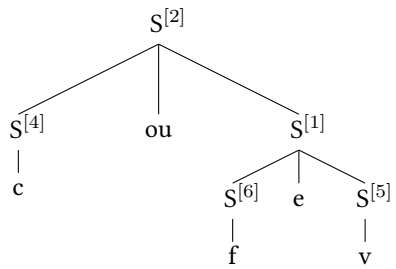
No caso de (a), usamos a regra R_3 da gramática para passar de S a [não S]. Em seguida, usamos R_4 para inserir c, que é um item do vocabulário terminal. No caso de (b), usamos R_1 para passar de S para [S e S], e em seguida usamos R_4 e R_5 para inserção dos itens terminais. Por fim, no caso de (c), começamos com R_2 para obter [S ou S]. Para o S à esquerda do conectivo, usamos R_6 para a inserção de f. Já para o S à direita do conectivo, usamos R_3 para obter a sequência [neg S] e, em seguida, R_5 para o item v do vocabulário terminal. Como se pode ver, nossa gramática gera corretamente as estruturas para as três sentenças em (37).

Vejamos agora um exemplo que fará retomar o conceito de ambiguidade estrutural que vimos anteriormente:

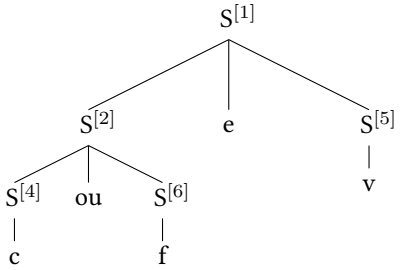
(39) Está chovendo ou está frio e está ventando.

Se você analisar essa sentença exatamente como acabamos de fazer com as sentenças em (37), vai perceber que há duas derivações possíveis, ambas começando com S e terminando com os itens do vocabulário terminal na ordem em que aparecem em (39). Eis as derivações, novamente com as respectivas regras gramaticais em cada galho:

(40) Primeira derivação para (39)



(41) Segunda derivação para (39)



Talvez você não tenha percebido esta ambiguidade já de saída, pois, de fato, ela é sutil. Porém, ela tende a ficar mais nítida se pronunciarmos a sentença com duas prosódias distintas. Para a primeira derivação, faça uma pausa logo antes do *ou* e passe mais rapidamente pelo *e*. Isso deve tornar mais saliente a interpretação correspondente a uma asserção alternativa: ou está chovendo, ou está fazendo frio e ventando. Para a segunda derivação, faça o oposto, passando mais rapidamente pelo *ou* e dando uma breve pausa entes do *e*. Isso deve tornar mais saliente a interpretação correspondente a uma asserção aditiva: está ventando e, ou está chovendo ou fazendo frio.

Por fim, uma questão importante. Queremos que nossa gramática G gere apenas sentenças que, de fato, pertençam ao fragmento do português que tínhamos em mente. As sentenças (37) e (39) são exemplos assim. Há, porém, muitas sequências formadas pelos mesmos itens de nosso vocabulário terminal e que não queremos que sejam geradas pela gramática. Eis alguns exemplos:

(42) a. *E está chovendo está ventando.
b. *Está chovendo não é verdade que está ventando.

c. *Ou está frio e não é verdade que está ventando.

Como o leitor poderá verificar, não há nenhum derivação que siga as regras de G e que termine com essas sequências. Nossa gramática, corretamente, não as gera.

Encerramos, assim, nossa breve apresentação do que é uma gramática formal, capaz de gerar recursivamente um número potencialmente infinito de estruturas sentencias, mesmo tendo por base um vocabulário finito. Vimos como tal gramática especifica um conjunto de sentenças, discriminando entre estruturas gramaticais e agramaticais. Primariamente, gramáticas formais são objetos simbólicos, matemáticos. Cognitivamente, podemos pensá-las como modelos abstratos de nossa competência sintática, manifestada em nossa capacidade de julgar certas sequências como linguisticamente bem formadas e outras como linguisticamente mal formadas.

Na próxima seção, faremos um percurso semelhante ao que acabamos de fazer, mas com a atenção voltada à interpretação das sentenças geradas pela gramática. Estaremos em busca de um objeto lógico que modele aspectos de nossa competência interpretativa. Entraremos, assim, no domínio da semântica formal.

3. Semântica

A semântica é o estudo do significado. Neste texto, ela será o estudo do significado sentencial. As sentenças, como definidas na seção anterior, são estruturas hierárquicas geradas por uma gramática. Nossa semântica, portanto, deverá ser capaz de atribuir significado a toda sentença que nossa gramática for capaz de gerar. Isso quer dizer que a recursividade de nossa sintaxe deverá ser espelhada em nossa semântica, a fim de que ela seja capaz de interpretar um número infinito de sentenças. Continuaremos com nossa perspectiva formal, ou seja, buscaremos um modelo explícito, lógico-matemático, que receba como entrada uma estrutura sintática e que retorne seu significado.

O "significado", porém, é uma noção um tanto vaga e cheia de nuances. Precisamos, logo de saída, ser explícitos em relação ao que estamos nos referindo quando falamos do significado sentencial. No âmbito deste trabalho, nosso foco estará no que se costuma chamar de função **referencial** da linguagem. Trata-se de um aspecto central da linguagem humana, que diz respeito à relação entre expressões linguísticas (sentenças declarativas, no nosso caso) e o mundo. Se perguntamos a alguém como está o tempo lá fora e essa pessoa responder com a sentença *está chovendo*, interpretamos a resposta como dizendo respeito às condições meteorológicas no local e momento em que nos situamos. Tal resposta, portanto, informou algo sobre o mundo. Dominando a semântica do português, somos capazes de passar da linguagem para o mundo.

Imagine, agora, uma outra situação: você está dormindo em seu quarto com a janela e as cortinas fechadas quando seu despertador toca. Tendo planejado uma caminhada pelo parque pela manhã, você se pergunta se está ou não chovendo. Abre então as cortinas, olha pela janela, vê água caindo das nuvens e, imediatamente, diz a si mesmo: *está chovendo*. Nesse exemplo, você foi capaz de passar dos fatos (do

mundo, portanto) para a linguagem. Novamente, isso só foi possível graças ao seu domínio da semântica do português.

A conclusão a que chegamos é que o domínio da semântica de uma língua nos permite transitar do mundo para a linguagem e da linguagem para o mundo. É esse aspecto do significado que buscaremos captar e modelar formalmente nesta seção.

3.1. Condições de verdade

No núcleo de nossa semântica, está a noção de *verdade*. Vejamos o porquê, voltando ao nosso cenário anterior, no momento em que o despertador toca. Naquele momento, você ainda não sabia se a sentença *está chovendo* era verdadeira ou não. Para descobrir, foi preciso abrir a cortina e observar através da janela se estava caindo água das nuvens ou não. Por que você agiu assim? Porque estar caindo água das nuvens no momento e no local em que você se encontra são as condições necessárias e suficientes para que a sentença *está chovendo* seja verdadeira.

A lição que podemos tirar dessa breve narrativa é que saber o significado de uma sentença é saber as suas **condições de verdade**, ou seja, as condições necessárias e suficientes para que a sentença seja verdadeira. Se você sabe o significado da sentença *está chovendo*, sabe como o mundo deve ser para que ela seja verdadeira. Obviamente, por si só, isso não informa se a sentença é verdadeira ou não. Mas, se tiver acesso aos fatos relacionados, você saberá o que observar.

Nossa semântica será, portanto, uma semântica baseada em condições de verdade. Para toda e qualquer sentença declarativa S de uma língua, o significado de S será suas condições de verdade:[2]

(43) S é verdadeira se, e somente se, ...

S, como já sabemos, é uma estrutura sintática gerada pela gramática da língua em questão. A locução *se, e somente se* expressa a noção de condições ao mesmo tempo suficientes (*se*) e necessárias (*somente se*). As reticências marcam o local em que se especifica tais condições.

No caso de sentenças simples, como *está chovendo*, *está ventando* e *está frio*, não teremos nada muito interessante a dizer, apenas que se trata de condições de verdade distintas, já que as sentenças não são sinônimas. Não será nosso objetivo especificar exatamente o que está por trás de estados meteorológicos como chuva, vento e frio. Por isso, vamos nos contentar com especificações como as seguintes (abreviaremos a locução *se, e somente se* por *sse*):

(44) a. *está chovendo* é verdadeira sse estiver chovendo.
b. *está ventando* é verdadeira sse estiver ventando.
c. *está frio* é verdadeira sse estiver frio.

---

[2] A ideia de uma semântica baseada em condições de verdade se encontra nos trabalhos pioneiros do lógico polonês Alfred Tarski. Sua aplicação às línguas naturais se deve, sobretudo, ao filósofo norte-americano Donald Davidson. Ver, por exemplo, Tarski (1944) e Davidson (1984). Para uma introdução acessível, consultar Larson (2022).

Atente-se apenas para o seguinte: as expressões grafadas em itálico são objetos sintáticos, sentenças da língua que estamos interpretando. Essa língua é chamada de **linguagem objeto**. No nosso caso, a linguagem objeto é o português. Já o restante, grafado em tipo comum, está escrito na língua que estamos usando para teorizar, ou seja, para especificar a semântica da língua-objeto. É chamada de **metalinguagem**. Estamos usando o próprio português, o que dá às afirmações em (44) um ar um tanto circular ou banal. Se, no entanto, fosse o inglês nossa linguagem objeto, as coisas pareceriam menos banais, sobretudo para um leitor brasileiro monolíngue que estivesse tendo um primeiro contato com a língua inglesa:

(45) a. *it is raining* é verdadeira sse estiver chovendo.
b. *it is windy* é verdadeira sse estiver ventando.
c. *it is cold* é verdadeira sse estiver frio.

Será, entretanto, na interpretação de sentenças complexas, como as que vimos na seção anterior, que nossa semântica e o aparato formal que desenvolveremos a partir de agora mostrará mais claramente sua utilidade. Antes de analisá-las, porém, introduziremos uma primeira dose de formalismo, ainda com foco no que acabamos de ver em (44). Sentenças declarativas podem ser verdadeiras ou falsas. Quando uma sentença é verdadeira, diremos que seu **valor de verdade** é 1. Quando uma sentença é falsa, diremos que seu valor de verdade é 0. Representaremos o valor de verdade de uma sentença S por colchetes duplos: $[\![S]\!]$. Sendo assim, as afirmações em (44) podem ser reescritas como em (46):

(46) a. $[\![$*está chovendo*$]\!] = 1$ sse estiver chovendo.
b. $[\![$*está ventando*$]\!] = 1$ sse estiver ventando.
c. $[\![$*está frio*$]\!] = 1$ sse estiver fazendo frio.

Isso posto, passemos agora às sentenças complexas.

3.2. Composicionalidade

Lembremos que a gramática G da seção anterior gerava três tipos de sentenças complexas: sentenças negativas, sentenças coordenadas com o conectivo *e* e sentenças coordenadas com o conectivo *ou*. Retomemos alguns exemplos e as regras gramaticais em questão:

(47) a. Não é verdade que está chovendo
b. Está chovendo e está ventando.
c. Está chovendo ou está ventando.

(48) a. S $\rightarrow$ neg S
b. S $\rightarrow$ S e S
c. S $\rightarrow$ S ou S

Comecemos com (47a). Comparando-a com sua contraparte afirmativa *está chovendo*, intuímos que ela será verdadeira se sua contraparte for falsa, e falsa se sua contraparte for verdadeira. Em outras palavras, o papel semântico da negação é inverter o valor de

verdade da sentença com a qual combina. Isso pode ser expresso formalmente através da seguinte regra semântica, que pode ser aplicada a qualquer sentença negativa formada por (48a):

(49) $$[\![\text{neg S}]\!] = \begin{cases} 1 & se\ [\![\text{S}]\!] = 0 \\ 0 & se\ [\![\text{S}]\!] = 1 \end{cases}$$

De forma mais sucinta:

(50) $[\![\text{neg S}]\!] = 1$ sse $[\![\text{S}]\!] = 0$

Passando a (47b), intuímos facilmente que essa sentença será verdadeira se ambas as sentenças coordenadas forem verdadeiras, e falsa se ao menos uma delas for falsa. Formalmente, para duas sentenças quaisquer S_1 e S_2:

(51) $$[\![\text{S}_1 \text{ e S}_2]\!] = \begin{cases} 0 & se\ [\![\text{S}_1]\!] = 0 \text{ e } [\![\text{S}_2]\!] = 0 \\ 0 & se\ [\![\text{S}_1]\!] = 1 \text{ e } [\![\text{S}_2]\!] = 0 \\ 0 & se\ [\![\text{S}_1]\!] = 0 \text{ e } [\![\text{S}_2]\!] = 1 \\ 1 & se\ [\![\text{S}_1]\!] = 1 \text{ e } [\![\text{S}_2]\!] = 1 \end{cases}$$

De forma mais sucinta:

(52) $[\![\text{S}_1 \text{ e S}_2]\!] = 1$ sse $[\![\text{S}_1]\!] = 1$ e $[\![\text{S}_2]\!] = 1$

Essa regra semântica se aplica a qualquer sentença coordenada formada pela conjunção aditiva *e* através da regra (48b).

Já em relação a (47c), as intuições são um pouco menos óbvias. Quando apenas uma das sentenças coordenadas for verdadeira, está claro que (47c) será verdadeira. Se uma pessoa diz que está chovendo ou ventando e então se verifica que estava chovendo, mas não ventando (ou vice-versa), a pessoa estava correta. Igualmente, quando ambas as sentenças forem falsas, (47c) será falsa. Se a pessoa me diz que está chovendo ou ventando, e se verifica que não estava chovendo nem ventando, ela estava errada. O fator complicador em relação a (47c) e às sentenças formadas pelo *ou* em geral é a possibilidade de ambas as alternativas serem verdadeiras. Se uma pessoa diz que está chovendo ou ventando e se verifica que estava chovendo e também ventando, a pessoa estava certa ou errada? Seria o *ou* um conectivo *inclusivo*, que admite a possibilidade de ambas as sentenças coordenadas serem verdadeiras, ou um conectivo *exclusivo*, que não admite tal possibilidade? Não há uma resposta única. É possível formalizar as duas opções. Para um *ou* inclusivo, teríamos:

(53) $$[\![\text{S}_1 \text{ ou S}_2]\!] = \begin{cases} 0 & se\ [\![\text{S}_1]\!] = 0 \text{ e } [\![\text{S}_2]\!] = 0 \\ 1 & se\ [\![\text{S}_1]\!] = 1 \text{ e } [\![\text{S}_2]\!] = 0 \\ 1 & se\ [\![\text{S}_1]\!] = 0 \text{ e } [\![\text{S}_2]\!] = 1 \\ 1 & se\ [\![\text{S}_1]\!] = 1 \text{ e } [\![\text{S}_2]\!] = 1 \end{cases}$$

Já para um *ou* exclusivo, teríamos:

(54) $$[\![S_1 \text{ ou } S_2]\!] = \begin{cases} 0 & se\ [\![S_1]\!] = 0 \text{ e } [\![S_2]\!] = 0 \\ 1 & se\ [\![S_1]\!] = 1 \text{ e } [\![S_2]\!] = 0 \\ 1 & se\ [\![S_1]\!] = 0 \text{ e } [\![S_2]\!] = 1 \\ 0 & se\ [\![S_1]\!] = 1 \text{ e } [\![S_2]\!] = 1 \end{cases}$$

Vamos admitir, de maneira assumidamente precipitada, que o *ou* do português tem a semântica inclusiva em (53). Dessa maneira, todas as sentenças geradas pela regra (48c) serão interpretadas de acordo com (53). Representaremos sucintamente essa versão inclusiva da seguinte forma:

(55) $[\![S_1 \text{ ou } S_2]\!] = 1$ sse $[\![S_1]\!] = 1$ ou_{inc} $[\![S_2]\!] = 1$

Com as três regras semânticas que acabamos de formular, estamos aptos a interpretar composicionalmente, ou seja, a partir de uma estrutura sintática, qualquer sentença formada pelas três regras recursivas de nossa gramática.

3.3. Uma semântica formal

Comecemos listando nossas regras semânticas:

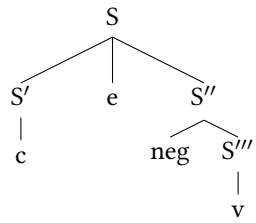
(56) a. $[\![\text{neg } S]\!] = 1$ sse $[\![S]\!] = 0$
b. $[\![S_1 \text{ e } S_2]\!] = 1$ sse $[\![S_1]\!] = 1$ e $[\![S_2]\!] = 1$
c. $[\![S_1 \text{ ou } S_2]\!] = 1$ sse $[\![S_1]\!] = 1$ ou_{inc} $[\![S_2]\!] = 1$

Com essas regras em mente, vejamos um exemplo de sentença um pouco mais complexa:

(57) Está chovendo e não é verdade que está ventando.

Sua estrutura sintática é representada em (58), com as letras c e v representado as sentenças simples, como fizemos na seção anterior, e com o uso de ′ para facilitar a referência aos diferentes constituintes sentenciais:

(58)



Derivemos, então, o significado de S, ou seja, suas condições de verdade, procedendo de cima para baixo, de acordo com sua estrutura sintática. Como S é formada pela conjunção *e*, usaremos a regra (56b):

(59) $[\![S]\!] = 1$ sse $[\![S']\!] = 1$ e $[\![S'']\!] = 1$

Como S″ é formada pela negação, usaremos para ela a regra (56a):

(60) $[\![S'']\!] = 1$ sse $[\![S''']\!] = 0$

De (59) e (60), temos que:

(61) $[\![S]\!] = 1$ sse $[\![S']\!] = 1$ e $[\![S''']\!] = 0$

Para completar a derivação, resta apenas especificar regras semânticas para S' e S''', que são estruturas não-ramificadas geradas pelas regras unárias de nossa gramática, que repetimos a seguir:

(62) a. $S \rightarrow c$
b. $S \rightarrow v$
c. $S \rightarrow f$

Como não há ramificação, podemos entender que o constituinte sentencial simplesmente herda o valor de verdade de seu único sub-constituinte. Em termos formais:

(63) a. $[\![[_S\ c]]\!] = [\![c]\!]$
b. $[\![[_S\ v]]\!] = [\![v]\!]$
c. $[\![[_S\ f]]\!] = [\![f]\!]$

Ou, se quisermos formulá-las em termos de condições de verdade:

(64) a. $[\![[_S\ c]]\!] = 1$ sse $[\![c]\!] = 1$
b. $[\![[_S\ v]]\!] = 1$ sse $[\![v]\!] = 1$
c. $[\![[_S\ f]]\!] = 1$ sse $[\![f]\!] = 1$

Com essas regras a nosso dispor, e de volta ao ponto em que paramos em (61), teremos:

(65) $[\![S']\!] = [\![c]\!]$

(66) $[\![S''']\!] = [\![v]\!]$

Após as devidas substituições em (61):

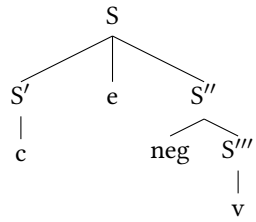
(67) $[\![S]\!] = 1$ sse $[\![c]\!] = 1$ e $[\![v]\!] = 0$

Como já sabemos as condições de verdade de c e v, podemos finalizar:

(68) $[\![S]\!] = 1$ sse estiver chovendo e não estiver ventando

Isso é exatamente o que queríamos derivar. Resumindo, o que fizemos foi derivar passo a passo as condições de verdade da árvore (58), que reproduzimos a seguir, desde o topo da estrutura até os itens terminais em sua parte mais baixa. Isso pode ser visualizado compactamente na **tabela de verdade** em (70):

(69)



(70)

		S'''	S''	S'	S
c	v	[S v]	[S neg [S v]]	[S c]	[S [S c] e [S neg [S v]]]
0	0	0	1	0	0
1	0	0	1	1	1
0	1	1	0	0	0
1	1	1	0	1	0

Tabelas de Verdade como (70) mostram, para cada combinação possível de valores de verdade atribuídos às sentenças simples de uma sentença S qualquer (as duas primeiras colunas em (70)), os valores de verdade dos constituintes sentenciais de S, incluindo os da própria sentença S (última coluna em (70)), Assim, vemos que a sentença completa só é interpretada como verdadeira quando simultaneamente *está chovendo* for verdadeira e *está ventando* for falsa.

Essas tabelas de verdade refletem o pareamento das regras sintáticas que compõem a gramática de nosso fragmento e as regras semânticas que as interpretam, como ilustrado a seguir:

(71)

Sintaxe	**Semântica**
$S \rightarrow S$ e S	$[\![S_1 \text{ e } S_2]\!] = 1$ sse $[\![S_1]\!] = 1$ e $[\![S_2]\!] = 1$
$S \rightarrow S$ ou S	$[\![S_1 \text{ ou } S_2]\!] = 1$ sse $[\![S_1]\!] = 1$ ou$_{\text{inc}}$ $[\![S_2]\!] = 1$
$S \rightarrow$ neg S	$[\![\text{não } S]\!] = 1$ sse $[\![S]\!] = 0$
$S \rightarrow$ c	$[\![\ [_S \text{ c}]\ ]\!] = 1$ sse $[\![\text{c}]\!] = 1$
$S \rightarrow$ v	$[\![\ [_S \text{ v}]\ ]\!] = 1$ sse $[\![\text{v}]\!] = 1$
$S \rightarrow$ f	$[\![\ [_S \text{ f}]\ ]\!] = 1$ sse $[\![\text{f}]\!] = 1$

Esse isomorfismo entre os dois sistemas de regras transfere para a semântica o caráter recursivo que havíamos dado à sintaxe na seção anterior. Formalizamos, assim, um fragmento de língua que permite gerar e interpretar composicionalmente, com recursos finitos, um número infinito de sentenças (estruturas sintagmáticas). Mas ainda há mais a extrair dessa formalização, como veremos a seguir.

3.4. Relações semânticas

Considere novamente a sentença (72):

(72) Está chovendo e não é verdade que está ventando.

Considere, agora, a sentença em (73):

(73) Está chovendo.

A princípio, (73) pode ser verdadeira ou falsa. Seu valor de verdade depende dos fatos. Imagine, porém, que alguém lhe diga que (72) é verdadeira. Imediatamente (isto é, sem precisar verificar os fatos), você concluirá que (73) também é verdadeira. Sua competência semântica lhe permite deduzir a verdade de (73) a partir da verdade de (72). Dizemos, nesse caso, que (72) **acarreta** (73). Acarretamento é uma relação semântica entre sentenças que equivale à noção de consequência lógica. Quando uma sentença S_1 acarreta uma sentença S_2, sempre que S_1 for verdadeira, S_2 também será verdadeira. Em outras palavras, não é possível que S_1 seja verdadeira e S_2 falsa em uma mesma situação.

O acarretamento não é uma relação simétrica. Não se pode dizer que (73) acarreta (72). Basta imaginar uma situação em que esteja chovendo e ventando. Nessa situação, (73) será verdadeira, mas (72), falsa.[3]

Considere, agora, a sentença (74):

(74) Está ventando.

A princípio, (74) pode ser verdadeira ou falsa. Mas e se alguém lhe garantir que (72) é verdadeira? Imediatamente você concluirá que (74) é falsa. Na direção inversa, imagine, agora, que alguém lhe diga que (74) é verdadeira. Imediatamente, você concluirá que (72) é falsa. A conclusão final é que é impossível que (72) e (74) sejam ambas verdadeiras em uma mesma situação. Dizemos, nesse caso, que as sentenças *não são* **consistentes** entre si, ou que elas são *inconsistentes*. Assim, como o acarretamento, (in)consistência é uma relação semântica entre sentenças.

Por fim, considere (75):

(75) Está frio.

Novamente, temos uma sentença que, a princípio, pode ser verdadeira ou falsa. E se alguém lhe informar que (72) é verdadeira? Ainda assim, não há inferência lógica a se fazer. Você continuará sem saber se (75) é verdadeira ou falsa. (72) nem acarreta nem é inconsistente com (75). Em casos como esse, dizemos que as sentenças são **consistentes**. É possível que ambas sejam verdadeiras em uma mesma situação, ainda que isso não seja necessário.

A semântica que formulamos na seção anterior nos permite definir formalmente essas relações que acabamos de caracterizar. Como preliminar, vamos chamar de **valoração** uma atribuição de valores de verdade (Verdadeiro / Falso ou 0 / 1) a todas as sentenças simples de uma língua. Para o fragmento do português correspondente à gramática G formalizada na Seção 2.3., por exemplo, no qual há três sentenças simples (representadas por c, f e v), haverá oito valorações possíveis. Observe que, quando a semântica da seção anterior nos entregava as condições de verdade de uma sentença complexa S, ela estava, de fato, apresentando o valor de verdade de S relativo a cada valoração possível. Isso, aliás, é o que as linhas de tabelas de verdade como a

[3] Há também situações especiais em que S_1 acarreta S_2 e, reciprocamente, S_2 acarreta S_1. Nesses casos, dizemos que as duas sentenças são *paráfrases* uma da outra.

que já vimos em (70) nos mostrava com clareza. Naquele exemplo, como havia duas sentenças simples contidas em S, a tabela continha apenas quatro linhas. Já sentenças complexas contendo três sentenças simples produzirão tabelas com oito linhas, e seriam dezesseis linhas para quatro sentenças, e assim por diante.

Vamos representar o valor de verdade de uma sentença S relativo a uma valoração v por $[\![\mathrm{S}]\!]^{val}$. A partir disso, podemos formalizar as relações de acarretamento e (in)consistência entre duas sentenças S_1 e S_2 quaisquer:

(76) *Acarretamento*
S_1 acarreta S_2 se, e somente se,
para toda valoração val, se $[\![\mathrm{S}_1]\!]^{val} = 1$, então $[\![\mathrm{S}_2]\!]^{val} = 1$.

(77) *Consistência*
S_1 e S_2 são consistentes entre si se, e somente se,
existe ao menos uma valoração val tal que $[\![\mathrm{S}_1]\!]^{val} = 1$ e $[\![\mathrm{S}_2]\!]^{val} = 1$.

(78) *Inconsistência*
S_1 e S_2 são inconsistentes entre si se, e somente se,
não existe valoração val tal que $[\![\mathrm{S}_1]\!]^{val} = 1$ e $[\![\mathrm{S}_2]\!]^{val} = 1$.

As relações semânticas entre (72) e (73)–(75) com que começamos esta seção podem ser inferidas da tabela de verdade a seguir (com as linhas numeradas):

(79)

	c	f	v	(73) [S c]	(75) [S f]	(74) [S v]	[S neg [S v]]	(72) [S [S c] e [S neg [S v]]]
1	0	0	0	0	0	0	1	0
2	1	0	0	1	0	0	1	1
3	0	0	1	0	0	1	0	0
4	1	0	1	1	0	1	0	0
5	0	1	0	0	1	0	1	0
6	1	1	0	1	1	0	1	1
7	0	1	1	0	1	1	0	0
8	1	1	1	1	1	1	0	0

Que (72) acarreta (73) pode ser visto comparando-se as colunas correspondentes às duas sentenças. Em todas as linhas em que o valor de verdade de (72) é 1 (linhas 2 e 6), o valor de verdade de (73) também é 1. Que (73) não acarreta (72) pode ser visto, por exemplo, na linha 4: de acordo com essa valoração, o valor de verdade de (73) é 1, mas o de (72) é 0. Que (72) e (75) são consistentes pode ser visto em suas respectivas colunas: como se vê na segunda linha, a valoração correspondente leva ambas as sentenças ao valor de verdade 1. Por fim, que (72) e (74) são inconsistentes pode ser visto comparando-as linha por linha: não existem dois valores iguais a 1 simultaneamente associados a elas em linha alguma.

4. Implementação computacional

Vamos, a partir de agora, implementar em uma linguagem de programação de computadores as representações formais que estudamos. Para tanto, utilizaremos a linguagem Python[4]. A escolha dessa linguagem, entre tantas outras boas opções possíveis, justifica-se principalmente por se tratar da linguagem mais usada no Processamento de Linguagem Natural (PLN) e nas tarefas envolvendo Inteligência Artificial de um modo geral.

Antes de seguir adiante, cabem algumas breves observações. Os códigos propostos têm em vista primordialmente atender a propósitos didáticos e esses, por sua vez, são dirigidos a pessoas sem qualquer experiência prévia com programação de computadores. Assim, as soluções criadas por nós não são necessariamente as mais elegantes ou computacionalmente mais otimizadas. Sua ambição é que sejam executáveis e inteligíveis passo a passo, pressupondo o menor esforço possível da parte de quem aprende.

A seção seguinte trata do acesso à plataforma de programação Google Colab. Se você já está familiarizado com ela, ou se já costuma desenvolver seus próprios programas usando outros recursos, pode pular direto para a Seção 4.2..

4.1. Preparação para execução dos programas

Embora os códigos que produziremos juntos possam ser executados em virtualmente qualquer máquina com um interpretador Python versão 3.0 ou superior instalado, sugerimos o uso de uma plataforma de programação online oferecida gratuitamente pelo Google, chamada Colab (forma abreviada de *Colaboratory*), a fim de tornar o trabalho mais simples para aqueles que não têm o Python instalado em seus computadores.

Para acessar o Colab, digite o endereço `https://colab.research.google.com` em seu navegador Web. Você deve chegar a uma página como a que aparece parcialmente representada na Figura 1.

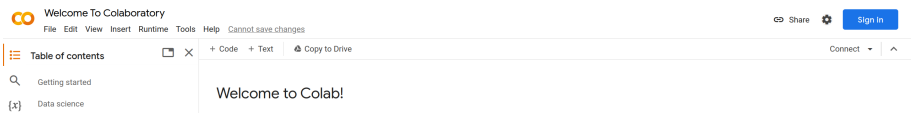


Figura 1: Vista parcial da tela inicial do Google Colab.

É importante chamar a atenção para alguns dos elementos visuais presentes já nessa página inicial. No canto superior esquerdo, logo abaixo da mensagem de boas-vindas (*Welcome to Colaboratory*), está a Barra de Menus (com File, Edit, View...). Abaixo da Barra de Menus, estendendo-se até o canto direito, está a Barra de Ferramentas (*toolbar*), que se inicia à esquerda com os botões + Code e + Text e se estende até um botão cujo texto atualiza-se em função da conexão ao servidor do Google. O

[4] `https://www.python.org`

texto inicialmente exibido deve ser Connect, indicando que você pode se conectar ao servidor ao clicar sobre ele. Isso não é estritamente necessário porque, ao se executar qualquer célula de código, como faremos logo mais, a conexão é estabelecida automaticamente.

Para começar a escrever seus programas, clique no item de menu File › New notebook.

Feito isso, o Colab cria um novo *notebook* e fica à espera de suas instruções. Um *notebook* é um ambiente interativo (passo a passo) de execução de instruções de uma linguagem de programação. Possui dois componentes básicos, ambos chamados de *células*:

Células de texto que servem basicamente para organizar o programa em blocos de execução, gerar explicações, comentários e anotações de todo tipo. Esses conteúdos textuais não fazem parte dos programas, mas ajudam a organizar o código e torná-lo mais inteligível para o próprio programador.

Células de código que são aquelas a executar. Nessas células, todos os conteúdos serão considerados expressões em linguagem Python. Se houver algum equívoco na escrita, como digitar uma palavra desconhecida pelo Python, esquecer de fechar parênteses ou outros, um erro será anunciado e a execução do programa, interrompida.

Você pode criar novas células dos dois tipos à vontade clicando sobre os botões + Code e + Text, na Barra de Ferramentas. Em nossos exercícios, vamos usar exclusivamente as células de código.

Ao abrir um novo *notebook*, a primeira célula de código é automaticamente criada. Nela, vamos escrever nosso primeiro programa. Se você nunca programou na linguagem Python, reza a tradição que seu primeiro programa deve, simplesmente, escrever uma mensagem para o usuário — e, indo além, que essa mensagem deve ser especificamente "Hello, World!". Isso pode parecer simplório (e, na maioria das vezes, é mesmo!), mas, em algumas linguagens de programação especialmente intrincadas, são necessárias várias linhas de código para escrever essa mensagem simples. Em Python, basta uma linha. Digite em sua célula de código:

```
1 print("Hello, World!")
```

Uma pequena observação: o número 1 que se vê à esquerda da célula corresponde à numeração das linhas, que é colocada automaticamente pelo Colab. Você pode escolher se quer ver os números de linha ou não através do menu Tools › Settings › Editor › Show line numbers.

Ao terminar de digitar, para executar o código, tecle Ctrl + Enter. Ao fazer isso, você deve perceber que leva algum tempo para aparecer o resultado da execução. Isso acontece só na primeira execução, pois o Colab não conecta automaticamente o seu *notebook* ao servidor do Google, porque isso consome recursos de hardware. Iniciado o processamento do código, o símbolo ▶ à esquerda da célula sendo processada recebe uma animação simples na forma de um contorno pontilhado. Ao final do

processamento, o resultado é exibido imediatamente abaixo do código que você digitou.

```
1 print("Hello, World!")
```

```
Hello, World!
```

Vamos resumir o que fizemos até este momento, aproveitando para esclarecer algumas das convenções da linguagem Python e da interface dos *notebooks* do Google Colab. Para exibir uma mensagem de texto em Python, você usou uma **função** definida nessa linguagem, que é a função `print()`. Uma função de uma linguagem de programação é uma série de procedimentos criados para realizar determinada tarefa — como, no caso de `print()`, exibir mensagens na tela. Para usar uma função do Python, você precisa conhecer o nome que ela tem e, além do nome, sempre digitar os parênteses logo em seguida, mesmo que sejam parênteses vazios, isto é, sem qualquer informação entre eles. Quando os parênteses não estão vazios, a informação entre eles é o conjunto de **parâmetros** passados à função. Continuando com nosso exemplo, `"Hello, World!"` foi o parâmetro (único, nesse caso) passado à função `print()`.

Você deve ter reparado que a mensagem exibida foi escrita dentro de aspas duplas. As aspas servem para delimitar as sequências de caracteres (letras, números ou símbolos como a pontuação), ou seja, indicar onde começam e terminam tais sequências. No Python, você pode escolher se prefere usar aspas simples ou duplas. Aqui, usamos as duplas, para que as simples possam ser usadas na função de apóstrofos.

Por último, algumas breves observações sobre a interface do Google Colab. Quando uma célula de código é executada sem produzir erros, ela pode gerar uma exibição logo abaixo do código inserido. Essa região abaixo da tela é chamada em inglês de *Output*. Vamos nos referir a ela por *Célula de Resultados*. Diferentemente das células de texto e de código, você não pode escrever nada diretamente na célula de resultados. Ela só aparece após a execução de códigos que geram algum tipo de mensagem para o usuário. Se você executar a mesma célula de código novamente, a célula de resultados será momentaneamente apagada e depois será novamente preenchida com informações.

Se tudo caminhou bem até aqui, podemos iniciar a implementação de nossa Semântica Formal na linguagem Python.

4.2. Implementação da Semântica Formal

Você deve ter percebido que as representações ligadas à Semântica Formal costumam operar sobre conjuntos de valores de verdade. Por exemplo, quando temos uma única sentença, o conjunto de valores de verdade associados à sua interpretação é $\{0, 1\}$, ou, ainda, $\{Falso, Verdadeiro\}$. Além disso, quando combinamos diferentes sentenças seja por conectivos (*e*, *ou* e outros), seja por relações tais como o acarretamento ou a inconsistência, esses dois valores de verdade são colocados em série de forma a expressar todas as combinações possíveis entre eles. Assim, para duas sentenças, teríamos o conjunto de combinações dado por $\{(0, 0), (1, 0), (0, 1), (1, 1)\}$.

No Python, existe um **módulo** criado para se trabalhar com *matrizes* e *vetores*, chamado `NumPy`. Um módulo é um conjunto de códigos pré-escritos que podem ser importados para os nossos programas a fim de poupar o trabalho de reescrever e testar o código ou, ainda, para estender os recursos já disponíveis. Alguns módulos desempenham papéis bastante específicos, como produzir gráficos estatísticos ou realizar cálculos de reagentes químicos. Para o programador, seria custoso desenvolver sozinho programas assim. Outros módulos são ainda mais difíceis de se produzir, em razão da complexidade da tarefa. Exemplos ligados ao processamento linguístico são os módulos de reconhecimento automático de voz, que permitem converter a fala humana em texto.

O `NumPy` é um módulo que oferece um único **objeto**, chamado `array`, que serve para representar tanto matrizes quanto vetores. Assim como o módulo, o *objeto* compreende um conjunto de funções e de informações pré-definidas relacionadas a elas (como parâmetros internos das funções). Um mesmo módulo pode abarcar diversos objetos.

As matrizes podem ser entendidas como *tabelas*, ou seja, arranjos estruturados de informações de um determinado **tipo** (números inteiros, números reais, sequências de caracteres...) com duas ou mais dimensões. Quando se tem uma matriz de duas dimensões, por exemplo, seus elementos se organizam em linhas e colunas. Já os vetores são séries unidimensionais de dados, como se fossem tabelas com uma única coluna.

Para usar o módulo `NumPy`, é preciso importá-lo para o seu notebook. Em um notebook novo, logo na primeira célula de código, digite:

```
1 import numpy as np
```

Ao executar essa célula, nenhuma informação deve aparecer na célula de resultados. Isso ocorre porque esse comando não produz qualquer tipo de *feedback* ao programador.

Observe, ao final da linha digitada, a expressão `as np`. Ela informa ao Python que `np` será o *apelido* do módulo `NumPy`. Esse apelido não é estritamente necessário, mas servirá para economizarmos um pouco de digitação ao usar o módulo nos códigos que escreveremos. Uma pequena ajuda, mas bem-vinda.

Antes de prosseguir, será útil compreender o uso básico de **variáveis** no Python. As variáveis são nomes arbitrários, escolhidos pelo programador, para representar objetos[5]. A atribuição do objeto à variável é feita através de uma expressão simples, em que a variável é declarada à esquerda do sinal de igual e, o objeto atribuído, à direita. Para efeito de ilustração, abra uma nova célula de código (clicando em `+ Code` na Barra de Ferramentas) e digite:

[5] Cabem duas observações sobre os nomes das variáveis. Primeira, eles devem sempre iniciar por uma letra e não podem conter símbolos especiais, como +, *, % e outros que têm significado determinado em Python (como operadores matemáticos, nesses exemplos). Na prática, é melhor pensar que só valem as letras, os números e o sinal de sublinhado, que é usado para separar palavras do nome (pois os nomes de variáveis não podem ter espaços em branco no meio). Segunda observação, é sempre melhor criar nomes associados ao que as variáveis representam, isto é, bons mnemônicos.

```
x = 2 + 2
saudação = "Hello, World!"
```

Executada a célula, a função `print()` pode ser usada em uma nova célula para exibir as informações associadas às variáveis.

```
print(x)
print(saudação)
```

```
4
Hello, World!
```

Note que as variáveis registram ali tipos de dados diferentes: a variável `x` está associada a um *número inteiro*, ao passo que `saudação` registra uma cadeia de caracteres, chamada de *string* no jargão da programação.

Agora podemos gerar representações da interpretação semântica das sentenças que vimos nas seções anteriores. Vamos iniciar com o caso mais simples, que é o conjunto das possibilidades interpretativas para uma única sentença. Como você deve se lembrar, há somente duas possibilidades que vamos considerar: a de que uma sentença seja verdadeira ou que seja falsa (1 ou 0). Tomemos a sentença (22a) (*Está chovendo*). Vamos representá-la com a variável `c`, que já foi usada para expressar essa sentença nas seções anteriores. A essa variável serão atribuídos seus dois valores de verdade possíveis. Tais valores, por sua vez, serão representados como itens de um vetor do `NumPy`.

```
c = np.array([0, 1])
```

Vamos compreender o que digitamos: `np` designa o módulo `NumPy` que importamos; `array` representa o método que cria o vetor de valores de verdade. Em seguida, encontramos parêntese e colchetes, que são delimitadores: os parênteses delimitam o parâmetro passado ao método de criação do vetor (`array`). Já os colchetes são delimitadores do próprio vetor. Assim, se abríssemos novos colchetes dentro dos mesmos parênteses, estaríamos criando novos vetores. Por fim, internamente ao vetor que estamos criando (isto é, dentro dos colchetes), a vírgula serve para separar entre eles os itens que, nesse caso, são os valores de verdade.

Nesse momento, se exibirmos os valores associados à variável `c`, veremos:

```
print(c)
```

```
[0 1]
```

Repare que a exibição gerada pela função `print()` não mostra a vírgula entre os itens do vetor.

4.3. Conectivos

Tendo produzido um vetor com os valores de verdade de uma sentença, podemos agora passar aos conectivos lógicos através dos quais as sentenças se combinam entre si, formando expressões compostas. Vimos que a interpretação de tais compostos depende do valor de verdade das sentenças e dos conectivos entre elas.

Negação

O conectivo mais simples entre os que já estudamos é a *negação*, pois trata-se de um conectivo *unário*, isto é, que afeta uma só sentença. Semanticamente, a negação inverte o valor de verdade de uma sentença.

O Python tem seu próprio operador de negação lógica, `not`, que é muito prático na construção de expressões. O problema é que esse operador incide sobre uma expressão de cada vez, o que não facilita as coisas quando se deseja avaliar uma *série* de valores de verdade. Essa é uma das razões pelas quais escolhemos usar o `NumPy`, que realiza essas operações em todos os itens de um vetor com uma única instrução.

```
1 print(np.logical_not(c))
```

```
[ True False]
```

Como se percebe, os valores de verdade de `c` aparecem, agora, na ordem invertida: primeiro o Verdadeiro, correspondente a `1`, depois o Falso, representado por `0`. Um detalhe incômodo nessa forma de apresentação é que, ao invés da expressão por números, o `NumPy` nos entregou palavras (`True` e `False`) — e, ainda por cima, em inglês. Tecnicamente, não há nada errado, mas não é a forma de exibição que preferiríamos. Felizmente, a adaptação é simples. Basta mudar o tipo (`astype`) do vetor de resultados para números inteiros (`int`):

```
1 print(np.logical_not(c).astype(int))
```

```
[1 0]
```

Essa solução funciona bem, mas ainda é inconveniente, porque é muito longa para se digitar várias vezes. Não seria melhor se tivéssemos um nome fácil e curto no lugar de tudo isso?

A maneira mais fácil de se resolver o problema é através da criação de uma *função* desenvolvida por nós mesmos. O Python oferece um recurso prático para a criação de funções simples, cuja declaração não ocupe mais de uma linha de código: são as funções `lambda`. Sua sintaxe é bastante sucinta: `lambda x: y`, em que `x` corresponde à variável recebida pela função e `y` corresponde ao processamento desejado do valor associado à variável. Vamos usá-la desta forma:

```
neg = lambda x: np.logical_not(x).astype(int)
```

Você certamente reconheceu a expressão à direita dos dois pontos. Ela é idêntica à que já digitamos antes. A única diferença é que ali aparece uma variável `x` no lugar em que antes digitamos `c`. Essa nova variável representa, na realidade, não um valor de vetor já determinado, mas qualquer vetor a ser passado para a função `lambda`. É por isso que essa mesma variável é declarada logo antes dos dois pontos. Ela indica uma informação que a função espera receber. Por fim, repare que toda a função definida por `lambda` é associada à variável `neg`. Assim, `neg` representa toda essa expressão cuja digitação queríamos poupar. Daqui por diante, para calcular a negação de uma sentença qualquer, bastará escrever:

```
print(neg(c))
```

```
[1 0]
```

Muito mais fácil, não? E tem mais: você deve se recordar de que uma das propriedades mais importantes da sintaxe e da semântica que vimos construindo é a *recursividade*, já que esta é uma característica essencial das línguas humanas. Pois bem: quando os conectivos são definidos como funções, eles se tornam recursivos, porque as funções podem receber outras funções (ou a si mesmas) como argumentos:

```
print(neg(neg(c)))
```

```
[0 1]
```

Observamos que os valores de verdade resultantes correspondem exatamente àqueles que originalmente formavam o vetor `c`. De quebra, aí temos um dos teoremas da lógica clássica: a Lei da Dupla Negação, segundo a qual a negação da negação de uma proposição p qualquer é a própria proposição p [6].

Antes de seguir para a formalização dos outros conectivos, convém criar mais um recurso para facilitar a visualização dos dados. Embora nossos vetores funcionem bem, eles até aqui mostram unicamente uma linha de valores de verdade, o que contrasta com as Tabelas de Verdade como aquelas que vimos nas seções precedentes. Como fazer para que nossos vetores também fiquem na forma de tabelas?

O `NumPy` traz um recurso que pode ser usado exatamente para isso. Trata-se do método `column_stack()`, que permite juntar vetores e, também, transpor as linhas em colunas. Aqui está ele já integrado em uma nova função `lambda`, para simplificar sua aplicação daqui por diante:

---

[6] Ou, na formulação de Whitehead e Russell (1910, p. 121): "uma proposição é equivalente à falsidade de sua negação". Tradução livre.

```
tab = lambda x: print(np.column_stack(x))
```

E aqui vai nossa primeira tabela, ainda bem pequena, contendo os vetores de `c` e da negação de `c`:

```
tab((c, neg(c)))
```

```
[[0 1]
 [1 0]]
```

A primeira coluna da tabela corresponde, como mostra o argumento da função `tab()`, ao vetor dos valores de verdade de `c`, e, a segunda, a negação desses valores. Muita atenção à chamada da função `tab()`: você vai observar que a expressão `c, neg(c)` está contida dentro de parênteses. Por quê? Porque, apesar de a forma `c, neg(c)` sem parênteses ser uma expressão válida para o Python, seriam dois argumentos, e não um, e a função `tab()` foi criada de forma a esperar por um único argumento, correspondente à variável `x` na declaração da função. A solução é, então, fazer com que `c` e `neg(c)` formem um só argumento composto. No Python, esse tipo de dado composto recebe o nome de **tupla**. Trata-se, no caso, de uma tupla com duas posições internas, mas que funciona como um objeto único quando usada como argumento de uma função.

Na célula de resultados onde aparece a tabela gerada, há três pares de colchetes. Dois desses pares delimitam os vetores de `c` e de `neg(c)`, naturalmente. Quanto ao par externo a esses vetores, ou seja, o primeiro colchete aberto na primeira linha, à esquerda, e o último fechado, na segunda linha, à direita, ele aparece por conta da aplicação da função `column_stack()`, que junta os vetores numa matriz bidimensional. Os colchetes externos marcam os limites dessa matriz, portanto.

Conectivos binários

Passemos aos conectivos binários, isto é, aqueles que se colocam entre duas sentenças. Será necessário representar cada uma das sentenças com os valores de verdade possíveis para sua interpretação (0 e 1) e, mais que isso, será preciso representar todas as possíveis relações entre elas, como o que se vê na definição da conjunção lógica em (51). Como se viu, há quatro linhas componentes da definição, cada uma delas correspondente a uma das combinações de valores de verdade possíveis. Como são dois valores de verdade para cada uma das duas sentenças, isso resulta em quatro combinações no total (2^2). Assim, vamos reescrever nosso vetor de valores de verdade para `c` e, ao mesmo tempo, criar um vetor de valores correspondentes à sentença (22b), *Está ventando*, a ser representada pela nova variável `v`.

```
c = np.array([0, 1, 0, 1])
v = np.array([0, 0, 1, 1])
```

Os conectivos binários serão definidos, assim como a negação, por funções do módulo `NumPy`, todas já convenientemente inseridas em suas respectivas funções `lambda`. Note que as funções, agora, recebem sempre dois argumentos (`x` e `y`) a fim de acomodar duas sentenças.

```
e = lambda x, y: np.logical_and(x, y).astype(int)
ou = lambda x, y: np.logical_or(x, y).astype(int)
ou_exc = lambda x, y: np.logical_xor(x, y).astype(int)
```

O uso dos novos conectivos é tão simples quanto o que vínhamos praticando com a negação. Para calcular a conjunção lógica das duas sentenças `c` e `v`, por exemplo, basta escrever:

```
print(e(c, v))
```

```
[0 0 0 1]
```

É importantíssimo observar, neste momento, que nossas funções são expressas por uma sintaxe diferente daquela da língua natural. Na nossa notação, os operadores (conectivos, no caso) precedem a declaração dos argumentos sobre os quais incidem, ou seja, escrevemos algo que leríamos como *E está chovendo está ventando*, o que seria totalmente aberrante na língua natural, em que os conectivos binários aparecem entre as sentenças, não antes delas. Essa notação em que os operadores precedem os argumentos é conhecida pelo nome de *notação polonesa*. Ela simplifica a interpretação das expressões, particularmente nas situações em que é necessário trabalhar com vários conectivos numa mesma expressão composta.[7]

Como se vê, a conjunção resulta verdadeira somente quando as duas sentenças conjugadas também são interpretadas como verdadeiras.

Para gerar uma tabela completa semelhante à que se vê em (51), usaremos a função `tab()` que criamos anteriormente.

---

[7] Isso, entretanto, não altera em nada o aspecto hierárquico das representações sintáticas introduzidas na Seção 2.. Note, por exemplo, que tanto (i) quanto (ii) representam conjunções de uma sentença simples c e a negação de uma outra sentença simples v:

(i) (c e (neg(v)))

(ii) (e(c, neg(v)))

Em ambos os casos, portanto, a negação se subordina à conjunção. Sendo assim, o caráter composicional da semântica formal apresentada na Seção 3.2. se preserva integralmente na implementação computacional que estamos desenvolvendo nesta seção.

```
tab((c, v, e(c, v)))
```

```
[[0 0 0]
 [1 0 0]
 [0 1 0]
 [1 1 1]]
```

As tabelas restantes, relativas ao *ou* inclusivo e ao *ou* exclusivo, respectivamente, são geradas de forma semelhante.

```
tab((c, v, ou(c, v)))
```

```
[[0 0 0]
 [1 0 1]
 [0 1 1]
 [1 1 1]]
```

```
tab((c, v, ou_exc(c, v)))
```

```
[[0 0 0]
 [1 0 1]
 [0 1 1]
 [1 1 0]]
```

4.4. Interpretação de sentenças compostas

Conhecendo as regras de formação de sentenças compostas com conectivos lógicos e suas respectivas interpretações em termos de valores de verdade, podemos criar composições mais complexas, envolvendo diversos conectivos a cada vez.

Nosso primeiro exercício será realizado com a sentença (57), *Está chovendo e não está ventando.* Já temos todos os elementos de que necessitamos: os vetores relativos às sentenças isoladas (`c` para *Está chovendo* e `v` para *Está ventando*), além das regras de interpretação dos conectivos de negação e de conjunção. Tudo o que precisamos fazer é escrever a expressão na sintaxe própria para o formalismo computacional que criamos. Assim, construiríamos uma expressão nestes termos:

```
print(e(c, neg(v)))
```

```
[0 1 0 0]
```

Simples, não? Indo um pouco além, você pode querer gerar uma tabela completa, que explicita todos os passos para a interpretação a que acabamos de chegar, como aquela que se vê em (70):

```
tab((v, neg(v), c, e(c, neg(v))))
```

```
[[0 1 0 0]
 [0 1 1 1]
 [1 0 0 0]
 [1 0 1 0]]
```

Nossa próxima tarefa é representar a Tabela (79), que é a maior deste capítulo. Nela, são usadas três sentenças. Será preciso, portanto, gerar vetores de valores de verdade para expressar as oito (2^3) combinações possíveis entre elas. Aqui estão os novos vetores:

```
c = np.array([0, 1, 0, 1, 0, 1, 0, 1])
v = np.array([0, 0, 1, 1, 0, 0, 1, 1])
f = np.array([0, 0, 0, 0, 1, 1, 1, 1])
```

Agora podemos gerar a tabela:

```
tab((c, f, v, neg(v), e(c, neg(v))))
```

```
[[0 0 0 1 0]
 [1 0 0 1 1]
 [0 0 1 0 0]
 [1 0 1 0 0]
 [0 1 0 1 0]
 [1 1 0 1 1]
 [0 1 1 0 0]
 [1 1 1 0 0]]
```

4.5. Cálculo das Relações Semânticas

Na Seção 3.4., estudamos três relações semânticas entre sentenças: consistência, inconsistência e acarretamento. Vamos tratar de implementá-las computacionalmente.

Consistência

Comecemos pela relação de consistência. Para que duas sentenças sejam consideradas consistentes entre si, tem de haver pelo menos um caso em que ambas sejam avaliadas como verdadeiras simultaneamente. Na representação vetorial que temos utilizado, a

tarefa seria verificar se há pelo menos um caso de valores 1 nas mesmas posições dos vetores relativos às duas sentenças em causa.

Uma forma prática de se mostrar as posições vetoriais em que estão os valores 1 é através do método `flatnonzero()` aplicável aos vetores do `NumPy`, que recebe como argumento um vetor (de zeros e uns, no nosso caso) e produz como resultado um novo vetor contendo os índices em que os valores do vetor analisado são diferentes de 0. Vejamos, como ilustração, o método aplicado ao vetor `v`.

```
1 print(v)
2 print(np.flatnonzero(v))
```

```
[0 0 1 1 0 0 1 1]
[2 3 6 7]
```

A primeira linha exibe o vetor `v`, só para refrescar nossa memória e facilitar a visualização dos dados que vão passar pelo método que no interessa. Nessa primeira linha, vemos que `v` tem quatro valores diferentes de 0 (ou seja, 1): são a terceira, quarta, sétima e oitava posições. Na segunda linha aparecem os índices correspondentes às posições de 1 em `v`. Os índices de um vetor são numerados a partir de 0 (correspondente à primeira posição). Assim, o vetor de resultado da aplicação do método `flatnonzero()`, que é `[2 3 6 7]`, indica que foram encontrados números 1 nas posições 3 (índice 2), 4 (índice 3) e assim por diante.

Se agora aplicarmos o mesmo método a um vetor de uma sentença consistente com `v`, como a sentença `c`, digamos, teríamos:

```
1 print(c)
2 print(np.flatnonzero(c))
```

```
[0 1 0 1 0 1 0 1]
[1 3 5 7]
```

Novamente, a linha (1), que exibe os valores de verdade do vetor `c`, está ali somente para tornar claro o funcionamento do método `flatnonzero()`. O que nos interessa mesmo é o segundo vetor, que mostra os índices em que o valor 1 aparece em `c`. Com ele, percebe-se que as sentenças `v` e `c` são ambas verdadeiras em dois casos, representados pelos índices coincidentes 3 e 7. Com isso, já podemos ter certeza de que elas são consistentes, pois bastaria um só caso em que ambas são verdadeiras para garantir sua consistência.

Falta, agora, implementar um meio de verificar se há índices em comum entre os dois vetores de índices de posições que vamos utilizar. Uma das maneiras de se fazer isso é tratar os vetores de posições como *conjuntos* e calcular a *intersecção* entre eles. O `NumPy` tem o método `intersect1d()` para isso. Vejamos:

```
print(np.intersect1d(np.flatnonzero(v),
↪  np.flatnonzero(c)))
```

```
[3 7]
```

O último passo é transformar esses elementos da intersecção entre os dois vetores numa informação *lógica*, isto é, avaliada como verdadeira ou falsa. É simples: o que queremos saber, afinal, é se a intersecção dos vetores de posição relativos a duas sentenças tem elementos ou não. Se tem, ou seja, se há intersecção, a relação entre as sentenças é consistente; caso contrário, não é consistente. O `NumPy`, mais uma vez, traz um método para verificar se um vetor qualquer é não-vazio e se possui itens diferentes de 0. Trata-se do método `any()`. No uso que nos interessa, ele resultará Verdadeiro (`True`) sempre que a intersecção que vamos testar entre as sentenças tiver um ou mais elementos.

```
print(np.any(np.intersect1d(np.flatnonzero(v),
↪  np.flatnonzero(c))))
```

```
True
```

Pronto! Aí está a função que buscávamos. Para tornar sua aplicação mais fácil, vamos associá-la a uma nova variável através de uma função `lambda`.

```
consistência = lambda x, y:
↪  np.any(np.intersect1d(np.flatnonzero(x),
↪  np.flatnonzero(y)))
```

Um pequeno detalhe sobre a exibição do código acima: você deve ter percebido que o texto na linha (1) é interrompido na posição de uma vírgula e continua na linha de baixo, que é marcada pelo sinal ↪ próximo à margem esquerda. Na sua digitação do código, você não precisa se preocupar em quebrar a linha. O Colab vai rolar automaticamente para a direita a célula onde você estiver digitando.

Voltando à nossa função recém-definida, agora podemos brincar à vontade:

```
print(consistência(v, c))
```

```
True
```

Que tal experimentar com sentenças que sejam contraditórias entre si?

```
1 print(consistência(v, neg(v)))
```

```
False
```

O resultado, como seria de se esperar, é o Falso. Isso nos conduz à formalização de outra relação semântica, a inconsistência.

Inconsistência

Uma vez definida a consistência, fazer o mesmo para a relação de inconsistência é muito fácil. A rigor, nem seria preciso defini-la, pois ela pode ser inferida a partir da aplicação da função `consistência()`, quando esta resulta no Falso, como acabamos de ver. De todo modo, se quisermos definir uma função particular para a inconsistência, basta inverter o resultado da função `consistência()`. Como afirmamos anteriormente, o Python dispõe de um operador lógico de negação, `not`, que podemos empregar agora.

```
1 inconsistência = lambda x, y: not(consistência(x, y))
```

Vamos usar nosso exemplo anterior de sentenças contraditórias como teste inicial.

```
1 print(inconsistência(v, neg(v)))
```

```
True
```

Na Tabela (79), o exemplo de inconsistência que vimos foi o da relação entre a sentença composta *Está chovendo e não está ventando* e a sentença simples *Está ventando*. É muito claro que elas são inconsistentes entre si. Vamos testar com a função que acabamos de definir:

```
1 print(inconsistência(v, e(c, neg(v))))
```

```
True
```

Acarretamento

Através da Regra (76), compreendemos que uma sentença S_1 acarreta outra sentença S_2 se, e somente se, em todos os casos em que S_1 for avaliada como verdadeira, S_2 também será. Cumpre lembrar que podem existir casos em que S_2 seja verdadeira e S_1, não, mas isso não atinge a aplicação da regra. Isso só nos lembra que a relação de acarretamento não é simétrica, ou seja, aplica-se de S_1 a S_2, mas não necessariamente o inverso. Além disso, postularemos que, para que haja acarretamento, é preciso que

exista ao menos um caso em que S_1 seja avaliada como verdadeira, pois, do contrário, não há o que acarretar[8].

Com essas definições em mente, passemos à implementação. Uma atitude importante para o programador é procurar conceber a solução do problema *antes* de começar a programar. Devemos nos perguntar quais os passos para a solução e, feito isso, quais os métodos de que dispomos para a implementação em código.

Como já nos acostumamos a gerar funções para aplicar nossas propostas de solução, podemos imaginar a criação de uma função `acarretamento()`, que buscaria resolver o problema desta maneira:

1. A função receberia como dados de entrada os vetores de valores de verdade que representam duas sentenças S_1 e S_2, a fim de que se verifique se a primeira acarreta a segunda.

2. Para garantir que S_1 seja verdadeira em pelo menos uma situação, deve-se verificar se existe algum valor 1 em seu vetor de valores de verdade.

3. Em seguida, passamos a verificar se, para todos os valores 1 no vetor S_1, existem valores também 1 ocupando a mesma posição no vetor S_2.

4. Se as duas condições acima estiverem satisfeitas, a função teria como resultado o Verdadeiro. Do contrário, o resultado será o Falso.

Os quatro passos que acabamos de descrever correspondem ao que se convencionou chamar de um **algoritmo**, isto é, uma descrição finita para a solução de um dado problema, que inclui a caracterização dos dados de entrada e de saída. Agora, com o algoritmo em mente, passemos à sua implementação em Python.

Recebidos os vetores das sentenças, a primeira condição a checar é se a primeira sentença tem ao menos um valor 1. Para isso, usaremos o método `any()`, que já conhecemos.

Tendo em vista a segunda condição, vamos precisar do método `flatnonzero()` para encontrar os índices das posições onde os vetores de sentenças têm valores diferentes de 0 (ou seja, 1). Com isso, vamos gerar dois vetores com esses índices das posições. Experimentemos com o acarretamento presente na Tabela (79). Nela, vimos que a sentença composta *Está chovendo e não está ventando* acarreta *Está chovendo*. Aqui estão os vetores resultados da aplicação do método `flatnonzero()` a essas sentenças:

```
print(np.flatnonzero(e(c, neg(v))))
print(np.flatnonzero(c))
```

```
[1 5]
[1 3 5 7]
```

[8] Convém observar que essa restrição não se aplica à lógica proposicional, em que a implicação resulta verdadeira mesmo quando o antecedente é falso.

De posse desses vetores, devemos agora verificar se todos os índices do primeiro vetor ($[1, 5]$) estão espelhados no segundo. A resposta, claro, é afirmativa, pois o segundo vetor exibe esses dois índices e mais alguns outros, que não nos interessam agora.

Formalmente, o que estamos verificando é se os números do primeiro vetor formam um *subconjunto* dos números do segundo. Em outras palavras, queremos saber se há elementos no primeiro conjunto que poderiam não aparecer no segundo. O `NumPy` traz o método `setdiff1d()` para verificar exatamente isso:

```
1 print(np.setdiff1d(np.flatnonzero(e(c, neg(v))),
  ↪  np.flatnonzero(c)))
```

```
[]
```

Como se vê, a resposta é negativa, isto é, um vetor vazio, o que significa que não há casos em que a primeira sentença é verdadeira e a segunda, não. Justamente o que buscávamos.

Para concluir, haverá mais uma etapa simples, que servirá para verificar se o primeiro vetor corresponde, de fato, a um subconjunto do segundo. Conforme acabamos de observar, quando isso ocorre, fica vazio o vetor resultante destinado a conter os elementos do primeiro conjunto que não estão no segundo é vazio. Desta vez, o `NumPy` não oferece um método para testar se um vetor corresponde a um conjunto vazio, mas essa é uma dificuldade simples de se contornar. Basta avaliar o tamanho do vetor, ou seja, aquilo que corresponderia à cardinalidade do conjunto. Por ser vazio, esse número deve ser o, naturalmente. Se houver qualquer resultado maior que zero, é porque existe algum elemento no vetor de valores de verdade da primeira sentença que não aparece no vetor da segunda – e não há acarretamento, portanto. Testemos:

```
1 print(np.setdiff1d(np.flatnonzero(e(c, neg(v))),
  ↪  np.flatnonzero(c)).size == 0)
```

```
True
```

Note que há dois sinais de igual `==` entre `size` e o. Esse é o símbolo do **comparador de igualdade**, um operador usado para verificar se dois valores são iguais. Quando um só sinal de igual é usado no Python, ele serve para atribuir valores a uma variável, conforme visto no início desta seção.

O próximo passo é juntar as duas condições que queríamos satisfazer numa só função. Relembrando, a primeira condição, que implementamos com `any()`, era que o vetor da primeira sentença não fosse vazio. A segunda, de que nos ocupávamos até agora, exige que todos os valores verdadeiros em S_1 também constem em S_2. Para a junção das duas condições, usaremos o operador lógico `and` do Python. Como você imaginaria, ele só resulta em Verdadeiro quando as duas condições estão individu-

almente também resultam em Verdadeiro. Finalmente, através da função `lambda`, vamos atribuir a função completa a uma variável `acarretamento` para facilitar sua aplicação.

```
acarretamento = lambda x, y: np.any(x) and
↪  np.setdiff1d(np.flatnonzero(x),
↪  np.flatnonzero(y)).size == 0
```

Experimentemos nossa função com o acarretamento da Tabela (79):

```
print(acarretamento(e(c, neg(v)), c))
```

```
True
```

Podemos agora inverter a ordem das sentenças a fim de mostrar que o acarretamento não é uma relação simétrica.

```
print(acarretamento(c, e(c, neg(v))))
```

```
False
```

5. Considerações finais

Neste capítulo, expusemos modelos que buscam representar formalmente algumas expressões das línguas naturais. Embora tais modelos possam ser generalizados para abarcar um número muito maior de expressões do que aquelas que exemplificamos, existem também limites e ressalvas que devemos estabelecer claramente.

Comecemos ressaltando que a semântica apresentada possui certos limites intrínsecos. Um deles é o Princípio da Bivalência, o que significa que toda sentença declarativa deve ser avaliada ou como verdadeira ou como falsa. Essa é a fonte de muitos questionamentos históricos à teoria. Em um de seus textos mais célebres, Bertrand Russell (1905) reflete sobre a interpretação da oração *O atual rei da França é calvo* nas situações em que não existe um rei da França, como era o caso quando o texto foi escrito (e, claro, ainda é). Se não existe um rei da França, tampouco existe o referente da expressão. Dessa forma, sobre o que se predica nessa oração, ou seja, a quem se aplica *ser calvo*? Ocorre que, por obediência ao Princípio da Bivalência, para qualquer sujeito A e qualquer predicado B, ou bem *A é B* ou bem *A não é B*, de modo que o rei da França tem de estar ou na extensão do predicado *ser calvo* (*A é B*) ou em seu complementar, ou seja, na extensão de *não ser calvo* (*A não é B*), sem qualquer outra possibilidade. O desenvolvimento completo da argumentação, que não vamos reproduzir aqui, levaria Russell a propor uma distinção entre ocorrências primárias e secundárias das expressões denotativas, numa tentativa de solucionar o problema com recursos estritamente lógico-semânticos. Com efeito, a consideração a tipos

distintos de ocorrência da expressão denotativa pode elucidar alguns dos problemas propostos, mas há também razões para entender que ela não esgota a questão, o que serviu de base para o surgimento de conceitos que, hoje, constituem os domínios compartilhados da pragmática, entre a linguística e a filosofia.

Para citar apenas algumas das contribuições mais célebres nessa direção, uma delas encontra-se já no artigo seminal *Sobre o Sentido e a Referência*, de Gottlob Frege (1892). Nessa obra, Frege apresenta uma questão ligada à pressuposição de existência de referentes que têm como sentido nomes próprios ou expressões que podem ser substituídas por nomes próprios: no uso da língua natural, acontece às vezes de usarmos nomes próprios ou expressões definidas no lugar de nomes próprios que, por alguma razão, podem não ter um referente associado. Retomando os exemplos de Frege, quando se diz *Kepler morreu na miséria* ou, ainda, *Quem quer que tenha criado a teoria orbital dos planetas morreu na miséria*, a garantia de um referente é uma *pré-condição*[9] necessária à valoração da proposição completa. Por sua vez, Peter Strawson (1950) toma explicitamente o texto de Russell como ponto crítico de partida para suas próprias reflexões que levaram ao estabelecimento de uma análise pragmática das expressões denotativas, incluindo o estudo das pressuposições.

Podemos ilustrar algumas das diferenças entre as interpretações semântica e pragmática das expressões linguísticas apoiados no estudo dos conectivos, como aqueles que estudamos ao longo deste capítulo. Considere-se a conjunção *e*, por exemplo. Vimos representações sintáticas e semânticas para ela. Na semântica, em particular, ela tem o papel de um conectivo com a função de combinar valores de verdade de duas sentenças de acordo com uma regra pré-definida, como aquela que expusemos em (56b). Formalmente, podemos expressá-la como uma intersecção dos conjuntos de valores de verdade da interpretação de duas sentenças S_1 e S_2, quaisquer sejam S_1 e S_2. Evidentemente, essa intersecção só resulta verdadeira quando ambos os valores de verdade conjugados também forem verdadeiros. Daí decorre, entre outras coisas, que $S_1\ e\ S_2$ é o mesmo que $S_2\ e\ S_1$, porque a ordem das sentenças não altera a intersecção de seus valores de verdade. Muito bem: embora essa seja uma interpretação cabível e até mesmo frequente do papel da conjunção, está longe de ser seu único uso possível nas expressões das línguas naturais. Sejam, como exemplos, as sentenças (80a) e (80b) em português:

(80) a. João faliu *e* foi vender picolé na praia.
b. João foi vender picolé na praia *e* faliu.

As breves biografias profissionais de João retratadas por (80a) e por (80b) diferem bastante. Ao ler ou ouvir (80a), temos a impressão de que os eventos aconteceram nessa ordem determinada, ou seja, primeiro João faliu e, *depois*, foi vender picolé. Mais que isso, pode-se entender também que ele foi vender picolé *porque* estava falido. Já em (80b), primeiro viria o empreendimento de vender picolés e esse teria sido o empreendimento a falir. Nada se diz sobre a causa da falência do negócio, nem o que fez João depois de falir, nem sobre sua motivação para começar a vender picolés. Se você também percebe essa leitura dos fatos, é porque atribui à conjunção *e* um valor

[9] Daí o termo fregeano para a pressuposição, *Voraussetzung*, literalmente "pré-condição" ou "pré-requisito".

não somente coordenativo, mas também *temporal* e *causal*. Observe ademais que, além das leituras temporal e causal, a conjunção no seu sentido de intersecção de valores continua valendo: se é verdade que João faliu e depois, por causa disso, foi vender picolé (exemplo (80a)), ou o inverso (80b), é automaticamente verdade que os dois eventos isolados aconteceram.

Questões semelhantes a essa, que partem do reconhecimento dos limites estritos que a análise semântica formal impõe às interpretações, motivaram Paul Grice a propor sua teoria das implicaturas, uma das bases da pragmática contemporânea. Nos termos dessa teoria, a ordenação das sentenças em torno da conjunção, que acabamos de expor, está prevista como uma das submáximas da **Máxima de Modo**: *Seja ordenado* (GRICE, 1975, p. 46). Uma de suas aplicações é esclarecer, justamente, que a expectativa dos falantes durante as situações de conversa é que a ordem de apresentação das sentenças corresponda à ordem de sucessão dos fatos, se não houver nada mais indicando o contrário. Não é preciso insistir sobre o fato, já bastante evidente, de que a semântica dos conectivos entre sentenças não capta por si só essa ordenação. Isso mostra que parte do significado dos enunciados das línguas naturais demanda análises que complementem a semântica. Inserido nesse panorama, o estudo do significado das relações entre expressões das línguas naturais e suas contrapartes nas teorias formais oferece um dos mais instigantes domínios de pesquisa linguística e filosófica.

Por fim, cabem algumas poucas palavras sobre o papel da implementação computacional das representações linguísticas. Criar e implementar algoritmos computacionais voltados ao processamento linguístico permite mostrar com clareza o quanto os modelos formais estão preparados para análises empíricas, inclusive pela exposição a conjuntos de dados de grande escala. Pensando nas questões que ilustramos neste capítulo, vimos que não é difícil implementar conectivos e relações semânticas entre sentenças, justamente porque é elevado o grau de rigor das definições e das interpretações nelas baseadas. O mesmo não se dá de forma tão automática com a interpretação dos *usos* das sentenças. Basta dizer que qualquer inferência feita nesses contextos pode ser **cancelada** em favor de outras interpretações. Desse modo, na fala espontânea dos falantes, surgem desafios à formalização de todo tipo. Em contraste com a antiguidade da gramática enquanto disciplina, só muito recentemente surgiram recursos para estudo da fala e da escrita aplicados a materiais de fontes espontâneas, geradas em situações de interação natural. Com a popularização de equipamentos e do acesso a informações nas redes, esses recursos estão hoje ao alcance de todos, o que vem impactando fortemente as pesquisas linguísticas no mundo todo.

Referências

CHOMSKY, N. *Syntactic Structures*. The Hague: Mouton, 1957.

DAVIDSON, D. *Inquiries into Truth and Interpretation*. Oxford: Clarendon Press, 1984.

FREGE, G. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, v. 100, p. 25–50, 1892.

GRICE, P. Logic and Conversation. In: COLE, P.; MORGAN, J. (Ed.). *Syntax and Semantics 3: Speech Acts*. New York: Academic Press, 1975. p. 41–58.

LARSON, R. *Grammar as Science*. Cambridge, Mass: MIT Press, 2010.

LARSON, R. *Semantics as Science*. Cambridge, Mass: MIT Press, 2022.

RUSSELL, B. On denoting. *Mind*, JSTOR, v. 14, n. 56, p. 479–493, 1905.

STRAWSON, P. F. On referring. *Mind*, JSTOR, v. 59, n. 235, p. 320–344, 1950.

TARSKI, A. The Semantic Conception of Truth and the Foundations of Semantics. *Philosophy and Phenomenological Research*, v. 4, p. 341–375, 1944.

WHITEHEAD, A. N.; RUSSELL, B. *Principia mathematica*. Cambridge University Press, 1910. v. 1. (University of Michigan Historical Math Collection). Disponível em: <https://name.umdl.umich.edu/u/umhistmath/aat3201.0001.001>.