UNIVERSIDADE DO ALGARVE

FACULDADE DE CIÊNCIAS E TECNOLOGIA

Lógica e Computação

(Sebenta)

Licenciatura em Engenharia Informática

(2º ano)

Daniel Graça

Ano letivo 2022/23

Conteúdo

Capítulo 1

Introdução

1.1 Motivação

Num curso universitário, para além das disciplinas mais aplicadas, um aluno terá também disciplinas ditas de base. Estas disciplinas nem sempre fornecem conhecimentos que possam ser imediatamente aplicados na prática, e por essa razão são às vezes desdenhadas por alguns alunos como sendo "inúteis".

As disciplinas aplicadas são certamente importantes, mas as disciplinas de base também têm um papel fundamental, pela maturidade intelectual que desenvolvem.

O conhecimento aplicado é, regra geral, mais imediato e de mais rápida aplicação. Mas também tem tendência a ser útil apenas numa janela temporal de tamanho relativamente limitado. E isto é um problema, se nós não nos preocuparmos em evoluir já que, utilizando as palavras de Isaac Asimov: "The only constant is change". Você pode saber neste momento o estado da arte de um certo domínio, mas de uma coisa pode estar certo: isso tudo vai mudar. O que era a "última novidade" em Informática há 10 ou 20 anos, estará provavelmente ultrapassado hoje em dia.

Por isso é importante ter disciplinas com conteúdos que fujam um pouco às necessidades imediatas, e que nos deem uma perspetiva mais global do domínio em que trabalhamos e de domínios relacionados. Estas disciplinas não se focam na implementação concreta de uma tecnologia particular (que pode ser hoje "extremamente moderna e atual", mas que daqui a um par de anos já estará completamente desatualizada), mas sim nas ideias e na forma de pensar que estiveram na génese dessa tecnologia. Por esta razão, estas disciplinas serão mais teóricas e menos práticas. Não terão provavelmente uma utilização tão imediata. Mas as ferramentas e formas de pensar que desenvolvem manterão a sua validade por um período de tempo bastante mais alargado.

A disciplina de Lógica e Computação é uma destas disciplina de base que referimos acima. Mais especificamente, pretende melhorar a vossa capacidade de raciocínio lógico, e dar-vos um conhecimento mais aprofundado do que significa o termo "computação".

A capacidade de raciocínio lógico é importante para conseguir dizer logicamente a um computador como chegar de um ponto A a um ponto B. Para nós, muitas vezes é

"evidente" que se pode ir de A para B (embora muitas vezes nos enganemos!). Mas um computador não tem a capacidade de chegar de A a B de forma intuitiva: todos os passos lógicos para lá chegar têm de ser especificados. Por outras palavras, vocês têm de ser capazes de se meter no papel de um computador para lhe dar as instruções que ele necessita. Daí a utilidade de melhorar as vossas capacidades de raciocínio lógico.

O outro grande objetivo desta disciplina é que percebam melhor o que o termo "computação" significa. Muitas pessoas têm uma noção intuitiva do que é uma computação. Mas se definirmos este termo de uma forma mais rigorosa, conseguimos chegar a conclusões que nunca poderiam ser obtidas se apenas nos fiássemos na nossa intuição. Por exemplo, iremos ver que há problemas não-computáveis, isto é, que não podem ser resolvidos através de *nenhum programa informático*, por mais sofisticado que seja esse programa, ou por mais rápido que seja o computador que o executa. Iremos ver que, mesmo dentro dos problemas resolúveis por programas informáticos, há uns que são muito mais difíceis de resolver do que outros (alguns problemas até são resolúveis teoricamente, mas na prática exigem tantos recursos computacionais, que quase podem ser considerados como sendo impossíveis de resolver). Desta forma é possível apresentar toda uma taxonomia sobre o grau de dificuldade computacional de problemas.

Em resumo, o objetivo desta segunda parte da disciplina é dar-vos uma melhor noção das potencialidades e dos limites da Informática.

Temos algumas sugestões para esta disciplina. Em particular recomendamos que tentem resolver os exercícios, em vez de simplesmente esperar pela resolução do exercício do quadro. O olhar para um exercício, tentando dar-lhe a volta sem sucesso pode parecer uma pura perda de tempo. Mas a verdade é que o seu cérebro está a tentar usar várias estratégias para resolver o exercício, mesmo que aparentemente sem sucesso. E, às vezes, quando menos se espera, vem aquele momento do "Eureka! Já sei como resolver o exercício!" — mas ele só aparece como o culminar de um esforço anterior. Com o tempo, tal como o exercício físico, será capaz de lidar com problemas mais complexos. Mas para isso é fundamental tentar resolver problemas.

Ao resolver exercícios, não há problema em não conseguir o resultado certo à primeira vez. O erro, longe de insucesso, significa que estamos a tentar sair da nossa zona de conforto, e a tentar adquirir novas competências. Afinal, e utilizando uma analogia com a programação, ninguém espera ter um programa funcional logo à primeira tentativa.

1.2 Breve descrição do conteúdo da disciplina

Nesta disciplina iremos abordar vários tópicos. Os primeiros dois tópicos constituem a primeira parte da disciplina (lógica) e são o núcleo duro de qualquer curso de lógica. A segunda parte (teoria da computação) é constituída pelos dois últimos tópicos.

1. **Cálculo Proposicional.** Neste tópico iremos estudar expressões (fórmulas) lógicas relativamente simples. Iremos começar por utilizar uma abordagem *semântica*,

em que estas fórmulas são interpretadas como podendo tomar dois valores distintos: verdadeiro ou falso. Iremos depois analisar estas fórmulas utilizando uma outra abordagem (sistemas dedutivos), que não utiliza a noção de "verdadeiro" ou "falso", mas sim regras de inferência para deduzir novas fórmulas lógicas a partir de um conjunto de fórmulas iniciais. No final desta secção, iremos ver que a abordagem semântica e a que utiliza sistemas dedutivos são equivalentes.

2. **Lógica de primeira ordem.** Embora o cálculo proposicional seja suficiente para muitas aplicações, há casos em que é necessário considerar uma teoria mais abrangente. Por essa razão introduz-se a lógica de primeira ordem, que nos permite trabalhar com quantificadores, etc. A lógica de primeira ordem é suficiente para a maioria das aplicações da Lógica à Matemática e à Informática. Mais uma vez iremos utilizar duas abordagens distintas para estudar a lógica de primeira ordem: iremos utilizar primeiro uma abordagem semântica (isto é, onde as fórmulas são interpretadas como podendo tomar os valores lógicos verdadeiro e falso), e depois iremos introduzir uma abordagem sintática, com recurso a sistemas dedutivos.

3. **Introdução à teoria da computabilidade.** Neste tópico iremos estudar as chamadas *máquinas de Turing*, que são o modelo teórico dos computadores digitais. Iremos estudar as capacidades computacionais deste modelo, relacionando-o com o computador digital (tese de Church-Turing), e iremos verificar que existem problemas *não-computáveis*, que não podem ser resolvidos por nenhuma máquina de Turing (isto é, que não podem ser resolvidos por nenhum programa informático).

4. **Complexidade computacional.** Finalmente, no último tópico desta disciplina, iremos efetuar uma análise mais fina dos resultados da secção anterior. Nessa secção vimos que há problemas que podem ser resolvidos por máquinas de Turing, enquanto que há outros que não o são. Mas, mesmo dentro da classe dos problemas que são resolúveis através de máquinas de Turing, pode haver problemas que exijam tantos recursos computacionais (tempo, memória, etc.), que na prática é também impossível resolvê-los. Por esta razão, há necessidade de fazer uma classificação mais fina dos problemas resolúveis através de máquinas de Turing, que seja feita de acordo com a quantidade de recursos necessários para os resolver. Isto é efetuado nesta secção, onde são também introduzidas as classes P e NP, que aparecem em muitos contextos importantes.

1.3 Notação Matemática

Nestes apontamentos iremos utilizar diversa notação matemática, que relembramos de forma muito sucinta. Se A é um conjunto, então $a \in A$ indica que o elemento a pertence ao conjunto A, e $B \subseteq A$ indica que B é um subconjunto de A, i.e. que B é um conjunto com a propriedade que todo o elemento de B é também um elemento de A. Escrevemos também $B \subsetneq A$ para indicar que B está estritamente contido em A, i.e. $B \subseteq A$ e $A \neq B$.

O conjunto vazio, designado de $\varnothing$, não contém nenhum elemento e está contido em qualquer conjunto. Utilizamos também a seguinte notação para conjuntos de números:

- $\mathbb{N} = \{1, 2, 3, \ldots\}$ designa o conjunto dos números naturais, e $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$;
- $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ designa o conjunto dos números inteiros;
- $\mathbb{Q}$ designa o conjunto dos números racionais;
- $\mathbb{R}$ designa o conjunto dos números reais, e $\mathbb{R}_0^+ = [0, +\infty[$.

Dados conjuntos A e B:

- $A \cap B$ designa a intersecção dos conjuntos A e B, i.e. o conjunto formado pelos elementos que pertencem simultaneamente a A e a B;
- $A \cup B$ designa a união dos conjuntos A e B, i.e. o conjunto formado pelos elementos que pertencem a pelo menos um dos conjuntos;
- $A \setminus B$ designa o conjunto formado pelos elementos de A que não pertencem a B;
- $A \times B = \{(a, b) : a \in A, b \in B\}$ designa o produto cartesiano de A com B;
- $A^k = \underbrace{A \times A \times \ldots \times A}_{k \text{ vezes}} = \{(a_1, a_2, \ldots, a_k) : a_1, a_2, \ldots, a_k \in A\}$ para $k \in \mathbb{N}$, onde obviamente $A^1 = A$;
- $\mathcal{P}(A) = \{X : X \subseteq A\}$ designa o conjunto das partes de A.

Se $(x_1, x_2, \ldots, x_k) \in A_1 \times A_2 \times \ldots \times A_k$ pertence a um produto cartesiano de k conjuntos (alguns destes conjuntos, ou mesmo todos, poderão ser iguais), então diz-se que $(x_1, x_2, \ldots, x_k)$ é um k-tuplo.

Uma relação é um subconjunto R de $A \times B$ ou, mais geralmente, de um produto cartesiano de k conjuntos. Neste último caso diz-se que estamos na presença de uma relação k-ária, ou de aridade k. Se $k = 2$, também é usual dizer-se que estamos na presença de uma relação binária.

Uma função f é uma relação R de $A \times B$ que a cada elemento de A associa um e um só elemento de B. O conjunto A diz-se o conjunto de partida da função e o conjunto B diz-se o conjunto de chegada. Dado um elemento a de A, designamos por $f(a)$ a imagem por f de a, que é o único elemento de B que está em relação com a através de R, i.e. se $(a, b) \in R$, então $b = f(a)$. É usual relaxar-se a noção de função para permitir que alguns elementos de A não tenham imagem, dizendo-se neste caso que f é uma função parcial. Nesta disciplina iremos assumir que qualquer função utilizada pode ser parcial, mesmo que isso não seja explicitamente mencionado.

Capítulo 2

Cálculo Proposicional

2.1 Linguagem natural *vs*. linguagens formais

Inicialmente a lógica fazia parte da Filosofia, e pretendia estudar a capacidade que os seres humanos possuem de efetuarem raciocínios, já que um ser humano consegue raciocinar sobre um conjunto de informações, para concluir novos factos. Um exemplo de raciocínio é o Silogismo, que certamente já terão aprendido nas aulas de Filosofia:

> **Premissa:** *Todos os homens são mortais.*
> **Premissa:** *Sócrates é um homem.*
> **Conclusão:** *Logo Sócrates é mortal.*

Se assumirmos que as premissas são verdadeiras, então a regra do Silogismo afirma que a conclusão também será verdadeira.

No entanto, a linguagem utilizada por todos nós (*linguagem natural*), não é normalmente muito precisa, pelo que muitas afirmações são ambíguas e não é possível de forma mecânica (i.e. através de um algoritmo) derivar sempre uma conclusão válida. Por exemplo considere o seguinte "silogismo":

> **Premissa:** *Todo o banco é um móvel.*
> **Premissa:** *A Caixa Geral de Depósitos é um banco.*
> **Conclusão:** *A Caixa Geral de Depósitos é um móvel.*

Como se pode ver acima o problema advém da ambiguidade da palavra "banco", que tanto pode significar um móvel, ou uma instituição financeira. Nós, humanos que sabemos português, sabemos que aquele raciocínio é falso, pois utilizamos uma "lógica informal" (ou *lógica natural*).

No entanto, este tipo de raciocínio é inútil se efetuado por um computador, pois ele não sabe distinguir os diferentes significados de uma palavra (e é uma das razões pelas quais é difícil a um computador compreender a linguagem humana, não obstante alguns progressos recentes, mas ainda limitados).

Mas a ambiguidade não é o único problema. Outro problema que pode acontecer quando utilizamos a linguagem natural é a existência de paradoxos. Por exemplo,

considere o seguinte exemplo (paradoxo do mentiroso):

> *Esta afirmação é falsa.*

Será a afirmação acima verdadeira ou falsa? Se é verdadeira, então terá de ser falsa. Por outro lado, se é falsa, então terá de ser verdadeira... Outra variante do paradoxo do mentiroso é a seguinte afirmação:

> *A afirmação seguinte é verdadeira. A afirmação anterior é falsa.*

Poderíamos ser levados a crer que os paradoxos com linguagem natural apenas ocorrem quando tentamos determinar se uma afirmação é verdadeira ou falsa. Mas tal não acontece, como ilustra o seguinte exemplo (paradoxo de Berry). Considere-se o número natural (isto é um número do conjunto $\mathbb{N}$) definido da seguinte forma:

> *O menor número natural que não pode ser definido com menos de vinte palavras.*

Como só existe um número finito de palavras, só se pode definir um número finito de números naturais com menos de vinte palavras, pelo que terá de haver um menor número que não pode ser definido com menos de vinte palavras. Mas em cima acabamos de o definir com 14 palavras...

Poderíamos ainda julgar que estes paradoxos são devidos ao uso da linguagem natural, e que a utilização de linguagem "matemática" resolveria todos estes problemas. Vamos ver se assim é no seguinte exemplo.

A noção de conjunto é uma das noções basilares da Matemática, e parece que a nossa noção intuitiva de conjunto é tão simples e natural, que nunca irá dar problemas. Mas isso não é verdade, tal como exemplificado no *paradoxo de Russel*: Considere o conjunto X definido da seguinte forma:

$$X = \{A | A \notin A\}.$$

Por exemplo $\{1\} \in X$ já que $\{1\} \notin \{1\}$. Será que $X \in X$? Por um lado, se $X \in X$, então por definição de X terá de ser $X \notin X$, o que é um absurdo. Por outro lado, se $X \notin X$, então por definição de X, $X \in X$, o que também é um absurdo... Este paradoxo pode ser eliminado se definirmos com rigor o que é um conjunto, em vez de nos basearmos na nossa intuição do que é um conjunto.

Estes paradoxos ilustram bem os problemas que podemos encontrar se nos basearmos apenas na nossa intuição. Em lógica procura-se eliminar estes problemas, de forma a que se possa raciocinar de forma logicamente correta.

2.2 Fórmulas proposicionais

Para eliminar a ambiguidade da linguagem natural, é usual utilizar em lógica as chamadas *linguagens formais*, que descrevem precisamente os objetos e conceitos sobre os quais iremos efetuar raciocínios, de forma a eliminar qualquer tipo de ambiguidade.

Isto é fundamental quando trabalhamos com computadores, pois os computadores não têm intuição e trabalham apenas com instruções rigorosas e precisas. Antes de definirmos o que é uma linguagem formal, necessitamos de alguns conceitos que serão agora apresentados.

Definição 2.2.1. Um *alfabeto* é um conjunto não-vazio.

Os elementos de um alfabeto são normalmente chamados de *símbolos*. Nesta disciplina iremos apenas considerar alfabetos finitos, pelo que haverá sempre apenas um número finito de símbolos. Em geral iremos utilizar letras gregas maiúsculas para designar alfabetos. Por exemplo, podemos considerar $\Sigma = \{0, 1\}$ como sendo o alfabeto binário, ou ainda o alfabeto $\Pi = \{\$, \ddagger, a\}$.

Definição 2.2.2. Uma *palavra* sobre um alfabeto Σ é uma sequência finita de símbolos de Σ. O conjunto de todas as palavras sobre o alfabeto Σ é designado por Σ^*.

Por exemplo, $12 + 3$ ou $23+$ são palavras sobre o alfabeto $\Sigma = \{1, 2, 3, +\}$ e da mesma forma $pq\wedge \in \{p, q, \wedge\}^*$.

Definição 2.2.3. Uma linguagem formal sobre um alfabeto Σ é um subconjunto L de Σ^*.

Por exemplo $\{p \vee q, \vee, \vee q\}$ é uma linguagem formal sobre o alfabeto $\{p, q, \vee\}$. É usual omitir-se o adjetivo "formal" quando estamos a falar de uma linguagem formal. Iremos agora definir a linguagem formal que define as *fórmulas do cálculo proposicional*. Para definir essa linguagem, temos de indicar o alfabeto utilizado. No caso da linguagem formada pelas fórmulas do cálculo proposicional, o alfabeto utilizado é constituído pelos seguintes elementos (símbolos):

1. Váriaveis proposicionais $p_1, p_2, p_3, \ldots$ (por conveniência, estes símbolos são frequentemente representados de $p, q, r, \ldots$);
2. Os símbolos $\neg, \wedge, \vee, \Rightarrow$, chamados de conetivos lógicos;
3. Os símbolos (,), (parêntesis) utilizados para agrupar expressões.

Uma vez definido o alfabeto da linguagem, não nos interessa considerar todas as possíveis palavras que podem ser geradas por este alfabeto, como $pq\wedge$, mas apenas palavras que tenham uma determinada estrutura, como $p \wedge q$.

Definição 2.2.4. Considere o alfabeto definido acima. Uma fórmula do cálculo proposicional é toda a expressão que pode ser obtida aplicando as seguintes regras:

(i) Toda a variável proposicional é uma fórmula;

(ii) Se ϕ é uma fórmula, então $(\neg\phi)$ é uma fórmula;

(iii) Se ϕ e ψ são fórmulas, então $(\phi \wedge \psi)$, $(\phi \vee \psi)$, e $(\phi \Rightarrow \psi)$ são fórmulas.

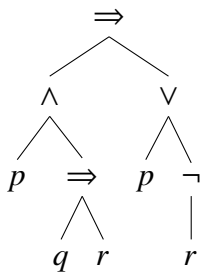


Figura 2.1: Árvore associada à fórmula $(p \wedge (q \Rightarrow r)) \Rightarrow (p \vee \neg r)$.

A linguagem formal definida acima, que consiste no *conjunto de todas fórmulas do cálculo proposicional* é normalmente designada de $\mathcal{F}_V$, onde V é o conjunto formado por todos os símbolos de variáveis, i.e. $V = \{p_1, p_2, p_3, \ldots\}$. Em geral, letras minúsculas (com ou sem índice) irão representar variáveis proposicionais. Letras gregas minúsculas (por exemplo ϕ, ψ, φ) irão representar fórmulas, e letras gregas maiúsculas (por exemplo Γ) irão representar conjuntos finitos de fórmulas.

Na definição dada acima, construímos fórmulas através da aplicação sucessiva das regras (i)–(iii). No entanto, em muitas situações, é mais adequado utilizar *árvores* para construir fórmulas. A cada fórmula podemos corresponder uma árvore em que a cada vértice está associado um símbolo de operador ou uma variável proposicional, com as seguintes propriedades:

1. Cada folha (também conhecida como nó terminal) está associada a uma variável proposicional.
2. Variáveis proposicionais apenas estão associadas a folhas.
3. Cada vértice associado a $\neg$ tem exatamente um sucessor.
4. Cada vértice associado a $\wedge$ ou $\vee$ tem exatamente dois sucessores.

Exemplo 2.2.5. Considere a fórmula $(p \wedge (q \Rightarrow r)) \Rightarrow (p \vee \neg r)$. A árvore que está associada a esta fórmula está indicada na Fig. 2.1.

De acordo com a Definição 2.2.4, cada vez que introduzimos um operador, temos de introduzir um par de parêntesis. Isto faz com que as fórmulas tenham muitos parêntesis e sejam pouco legíveis. Por esta razão, introduzimos a seguinte convenção: o símbolo de negação tem precedência sobre todos os outros símbolos. Também omitimos parêntesis, desde que isso não torne a fórmula ambígua (i.e. desde que a fórmula não possa ser interpretada de duas maneiras distintas). Por exemplo, não iremos colocar os parêntesis "mais de fora", porque o conteúdo da fórmula é claro sem estes parêntesis, como é o caso da seguinte expressão

$$\neg(\phi \Rightarrow \psi) \wedge \neg(\psi \Rightarrow \phi)$$

que designa a fórmula

$$((\neg(\phi \Rightarrow \psi)) \wedge (\neg(\psi \Rightarrow \phi)))$$

Note-se que, quando definimos as fórmulas do cálculo proposicional, não introduzimos o símbolo $\Leftrightarrow$. Podemos considerar que o símbolo $\Leftrightarrow$ é uma abreviatura:

$$\phi \Leftrightarrow \psi = (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$$

i.e. a fórmula $\phi \Leftrightarrow \psi$ é uma abreviatura da fórmula $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, onde ϕ, ψ são quaisquer fórmulas.

2.3 Semântica no cálculo proposicional

Uma fórmula, por si só, é uma expressão sem qualquer significado intrínseco. Na semântica pretende-se atribuir valores lógicos a fórmulas. Nomeadamente, no caso concreto do cálculo proposicional, pretende-se atribuir o valor lógico 0 (falso) ou 1 (verdadeiro) a uma fórmula. Isso pode ser feito através de valorações.

Definição 2.3.1. Seja V um conjunto de variáveis proposicionais. Uma *valoração* de V é uma aplicação $v : V \to \{0, 1\}$.

Obviamente não nos interessa dar valores lógicos apenas às variáveis proposicionais em V, mas sim a qualquer fórmula do cálculo proposicional sobre V.

Para isso vamos ter de saber como interpretamos fórmulas que contenham os símbolos $\neg$, $\wedge$, $\vee$, $\Rightarrow$ e $\Leftrightarrow$. Isso pode ser feito através de operadores booleanos.

Definição 2.3.2. Um *operador booleano* é uma função $f : \{0, 1\}^n \to \{0, 1\}$, onde $n \in \mathbb{N}$.

Na tabela seguinte, indicam-se os principais operadores que iremos utilizar:

nome	símbolo	nome	símbolo
conjunção	$\wedge$	*implicação*	$\Rightarrow$
disjunção	$\vee$	*equivalência*	$\Leftrightarrow$
negação	$\neg$		

O operador de negação é um operador unário (i.e. $n = 1$ na Definição 2.3.2) e os restantes operadores são binários ($n = 2$). Eles são definidos pelas seguintes tabelas:

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$	$x \vee y$	$x \Rightarrow y$	$x \Leftrightarrow y$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Note-se que a implicação não deve ser interpretada de acordo com a nossa intuição do dia a dia, em que uma expressão como "se a Terra é feita de queijo, então a Lua é feita de bolacha" não tem qualquer sentido, e será muito menos encarada como verdadeira pela larga maioria das pessoas. Ao invés, deve-se assumir que o operador de implicação não tem qualquer conhecimento sobre a validade do antecedente (por mais absurda que esta nos pareça), apenas se preocupando em garantir que se o antecedente é verdadeiro, então o mesmo terá de acontecer ao consequente. Essa utilização é mais natural, por exemplo, quando expressamos propriedades matemáticas. Por exemplo, se n é um inteiro, a afirmação:

$$\text{Se } n \geq 2 \text{ então } n^2 \geq 4$$

é vista como sendo verdadeira, e não nos preocupamos se para um dado inteiro n o antecedente $n \geq 2$ é ou não verdadeiro.

Agora já estamos em condições de estender a noção de valoração para fórmulas arbitrárias, obtendo as chamadas valorações totais.

Definição 2.3.3. Uma *valoração total* $\bar{v}$ (também chamada de *interpretação*) é uma aplicação $\bar{v} : \mathcal{F}_V \to \{0, 1\}$ que satisfaz as seguintes propriedades:

(a) Existe uma valoração $v : V \to \{0, 1\}$ tal que $\bar{v}(p_i) = v(p_i)$ para todo o $p_i \in V$;

(b) $\bar{v}(\neg\phi) = \neg\bar{v}(\phi)$ para todo o $\phi \in \mathcal{F}_V$;

(c) $\bar{v}(\phi \wedge \psi) = \bar{v}(\phi) \wedge \bar{v}(\psi)$; $\bar{v}(\phi \vee \psi) = \bar{v}(\phi) \vee \bar{v}(\psi)$, e $\bar{v}(\phi \Rightarrow \psi) = \bar{v}(\phi) \Rightarrow \bar{v}(\psi)$ para todos os $\phi, \psi \in \mathcal{F}_V$.

Note-se, que nas igualdades da definição acima, os símbolos $\neg, \wedge, \vee, \Rightarrow$ no lado direito designam os respetivos operadores booleanos.

Para simplificar a notação, iremos simplesmente chamar de valoração a uma valoração total.

Definição 2.3.4. Uma fórmula proposicional ϕ diz-se *satisfazível*, se $v(\phi) = 1$ para alguma valoração v.

Definição 2.3.5. Uma fórmula proposicional ϕ diz-se uma *tautologia*, se $v(\phi) = 1$ para toda a valoração v, escrevendo-se $\models \phi$.

Definição 2.3.6. Uma fórmula proposicional ϕ diz-se *contraditória*, se $v(\phi) = 0$ para toda a valoração v.

Se v é uma valoração e ϕ é uma fórmula onde se tenha $v(\phi) = 1$, então diz-se que a valoração v *satisfaz* a fórmula ϕ. Alternativamente, diz-se também que v é um *modelo* de ϕ.

Exemplo 2.3.7. Verifique se são tautologias, fórmulas contraditórias, ou fórmulas satisfazíveis, as seguintes fórmulas: $\phi = p_1 \wedge p_2, \psi = p_1 \wedge \neg p_1, \varphi = p_1 \vee \neg p_1$.

Resolução. Vamos criar tabelas de verdade (que já conhecem de disciplinas anteriores) para estas fórmulas. As tabelas de verdade indicam de modo sucinto todas as valorações possíveis, indicando de forma implícita o valor lógico que está associado a cada variável/fórmula para cada valoração (cada valoração corresponde a uma linha da tabela de verdade).

p_1	p_2	$p_1 \wedge p_2$
0	0	0
0	1	0
1	0	0
1	1	1

p_1	$p_1 \wedge \neg p_1$	$p_1 \vee \neg p_1$
0	0	1
1	0	1

Das tabelas de verdade verificamos que existe uma valoração que satisfaz ϕ (a valoração v definida por $v(p_1) = v(p_2) = 1$), duas valorações que satisfazem φ, e nenhuma valoração que satisfaz ψ. Portanto ϕ e φ são fórmulas satisfazíveis, enquanto que ψ não o é. Verificamos ainda que ψ é uma fórmula contraditória (não o sendo ϕ e φ) porque $v(\phi) = 0$ para toda a valoração v. Finalmente, ϕ e ψ não são tautologias, mas φ já o é, pois é satisfeita por todas as valorações.

Os conceitos apresentados anteriormente não têm necessariamente de se limitar a uma fórmula.

Definição 2.3.8. Um conjunto de fórmulas $\Gamma = \{\phi_1, \ldots, \phi_n\}$ diz-se *satisfazível*, se existe uma valoração v tal que $v(\phi_1) = \ldots = v(\phi_n) = 1$.

Exemplo 2.3.9. O conjunto $\Gamma = \{p_1 \wedge p_2, p_1 \vee \neg p_1\}$ é satisfazível, já que existe pelo menos uma valoração que satisfaz simultaneamente todas as fórmulas de Γ (um exemplo é a valoração v definida por $v(p_1) = v(p_2) = 1$, como se pode ver nas tabelas de verdade acima).

Vamos agora introduzir algumas definições e resultados sobre semântica do cálculo proposicional.

2.3.1 Fórmulas logicamente equivalentes

Às vezes duas fórmulas são aparentemente diferentes, mas comportam-se exatamente da mesma forma do ponto de vista lógico. Nesse caso estas fórmulas dizem-se logicamente equivalentes:

Definição 2.3.10. Sejam $\phi, \psi \in \mathcal{F}_V$. Se $v(\phi) = v(\psi)$ para todas as valorações v, então diz-se que ϕ e ψ são *logicamente equivalentes*, e escrevemos $\phi \equiv \psi$.

Note-se que se pode ter $\phi \neq \psi$ e $\phi \equiv \psi$. Ou seja, formalmente ϕ e ψ não são iguais, embora do ponto de vista semântico sejam equivalentes. Por isso, em rigor absoluto, é incorreto escrever-se $\phi = \psi$, e é por esta razão que se escreve $\phi \equiv \psi$.

Exemplo 2.3.11. Seja $V = \{p_1, p_2\}$ um conjunto de variáveis proposicionais e sejam $\phi = p_1 \wedge p_2$ e $\psi = \neg(\neg p_1 \vee \neg p_2)$. Mostre que $\phi \equiv \psi$.

Resolução. Para mostrar que $\phi \equiv \psi$, temos de mostrar que $v(\phi) = v(\psi)$ para todas as valorações v. Podemos verificar isso enumerando todas as valorações através de uma tabela de verdade

p_1	p_2	$p_1 \wedge p_2$	$\neg(\neg p_1 \vee \neg p_2)$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Da tabela de verdade concluímos que $v(\phi) = v(\psi)$ para todas as valorações v, pelo que $\phi \equiv \psi$.

Não é difícil verificar que o símbolo $\Leftrightarrow$, depois de interpretado por uma valoração, irá ser interpretado como o operador booleano "equivalência". Para isso basta-nos recordar que $\phi \Leftrightarrow \psi = (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$ e interpretar a fórmula $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$ com valorações. Por outras palavras, teremos $v(\phi \Leftrightarrow \psi) = (v(\phi) \Leftrightarrow v(\psi))$ para toda a valoração v. Daqui é possível mostrar o seguinte resultado.

Teorema 2.3.12. *$\phi \equiv \psi$ se e só se $\phi \Leftrightarrow \psi$ é uma tautologia.*

Vamos ver o que acontece quando substituímos variáveis por fórmulas, etc. em expressões de $\mathcal{F}_V$.

Teorema 2.3.13. *Sejam V um conjunto de variáveis proposicionais e ψ uma tautologia (fórmula contraditória) que utiliza somente variáveis $p_1, \ldots, p_n$ que pertencem a V. Sejam $\phi_1, \ldots, \phi_n$ fórmulas do cálculo proposicional, e seja ψ' uma fórmula obtida de ψ substituindo cada ocorrência da variável p_i em ψ por ϕ_i, para $i = 1, \ldots, n$. Então ψ' é uma tautologia (fórmula contraditória, respetivamente).*

Demonstração. Queremos mostrar que ψ' é uma tautologia, i.e. que $v(\psi') = 1$ para toda a valoração v. Seja v uma valoração arbitrária e v' uma valoração definida por

$$v'(p_i) = v(\phi_i), \text{ para } i = 1, \ldots, n$$

Então $v(\psi') = v'(\psi) = 1$. Isto mostra que ψ' é uma tautologia. Um raciocínio semelhante pode ser aplicado para o caso de fórmulas contraditórias. □

Por exemplo, vimos que a fórmula $\psi = p_1 \vee \neg p_1$ é uma tautologia. Então o teorema anterior garante que, para qualquer fórmula ϕ do cálculo proposicional, $\psi = \phi \vee \neg\phi$ é também uma tautologia. Em particular isso garante que, por exemplo, $(p_1 \Rightarrow p_2) \vee$

$\neg(p_1 \Rightarrow p_2)$ será então também uma tautologia. É possível verificar da mesma forma que $\phi \wedge \neg\phi$ é uma contradição.

Se $V = \{p_1, p_2, p_3, \ldots\}$ for o conjunto das variáveis proposicionais, iremos utilizar os símbolos $\bot$ e $\top$ para representar as fórmulas $p_1 \wedge \neg p_1$ e $p_1 \vee \neg p_1$, respetivamente:

$$\bot = p_1 \wedge \neg p_1$$
$$\top = p_1 \vee \neg p_1$$

Tem-se obviamente $v(\bot) = 0$ e $v(\top) = 1$ para qualquer valoração v.

Teorema 2.3.14. *Seja V um conjunto de variáveis proposicionais, e sejam ϕ e ψ fórmulas logicamente equivalentes de $\mathcal{F}_V$ que utilizam somente variáveis $p_1, \ldots, p_n \in V$. Sejam $\phi_1, \ldots, \phi_n$ fórmulas do cálculo proposicional, e sejam ϕ' e ψ' fórmulas obtidas de ϕ e ψ, respetivamente, substituindo cada ocorrência da variável p_i por ϕ_i, para $i = 1, \ldots, n$. Então $\phi' \equiv \psi'$.*

Demonstração. Suponhamos que se tem $\phi \equiv \psi$. Então pelo Teorema 2.3.12 tem-se que $\phi \Leftrightarrow \psi$ é uma tautologia. O Teorema 2.3.13 garante que $\phi' \Leftrightarrow \psi'$ também será uma tautologia. Logo, pelo Teorema 2.3.12, ter-se-á $\phi' \equiv \psi'$. □

Por exemplo, vimos que $\phi = p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2) = \psi$ (Exercício 2.3.11). Então o teorema anterior garante que, para todas as fórmulas $\varphi, \rho \in \mathcal{F}_V$, se tem $\phi' = \varphi \wedge \rho \equiv \neg(\neg\varphi \vee \neg\rho) = \psi'$.

Seguidamente apresentamos uma lista de fórmulas logicamente equivalentes. Estas relações podem ser verificadas por meio de tabelas de verdade para fórmulas que envolvem apenas variáveis proposicionais (para mostrar que, por exemplo, $p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2)$) e depois podemos substituir as variáveis por fórmulas arbitrárias utilizando o Teorema 2.3.14.

$$\begin{aligned}
&\phi \wedge \top \equiv \phi && \text{(elemento neutro)}\\
&\phi \vee \bot \equiv \phi \\
&\phi \wedge \bot \equiv \bot && \text{(elemento absorvente)}\\
&\phi \vee \top \equiv \top \\
&\phi \wedge \phi \equiv \phi && \text{(idempotência)}\\
&\phi \vee \phi \equiv \phi \\
&\phi \wedge \psi \equiv \psi \wedge \phi && \text{(comutatividade)}\\
&\phi \vee \psi \equiv \psi \vee \phi \\
&\phi \vee (\psi \wedge \varphi) \equiv (\phi \vee \psi) \wedge (\phi \vee \varphi) && \text{(distributividade)}\\
&\phi \wedge (\psi \vee \varphi) \equiv (\phi \wedge \psi) \vee (\phi \wedge \varphi) \\
&\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi) && \text{(leis de De Morgan)}\\
&\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)
\end{aligned} \tag{2.1}$$

$$\neg\neg\phi \equiv \phi \qquad \phi \Rightarrow \psi \equiv \neg\phi \vee \psi$$
$$\phi \Leftrightarrow \psi \equiv \psi \Leftrightarrow \phi \quad \phi \Rightarrow \psi \equiv \neg\psi \Rightarrow \neg\phi$$

2.3.2 Consequência semântica

Muitas vezes pretende-se dizer que uma coisa é consequência lógica de outra coisa. No cálculo proposicional, e em termos semânticos, essa noção pode ser definida do seguinte modo.

Definição 2.3.15. Seja Γ um conjunto finito de fórmulas e ϕ uma fórmula. Diz-se que ϕ é uma *consequência semântica* de Γ, e escreve-se $\Gamma \models \phi$, sse sempre que uma valoração v satisfaz simultaneamente todas as fórmulas de Γ, então essa valoração também satisfaz ϕ.

Exemplo 2.3.16. Sejam $\Gamma = \{p_1 \vee \neg p_2, p_1 \Rightarrow p_2\}$ e $\phi = \neg p_1 \vee \neg p_2$. Verifique se $\Gamma \models \phi$.

Resolução. Temos de verificar se, sempre que uma valoração satisfaz *simultaneamente* todas as fórmulas de Γ, então ela também satisfaz ϕ. Isso pode ser visto através de uma tabela de verdade, que enumera todas as possíveis valorações:

p_1	p_2	$p_1 \vee \neg p_2$	$p_1 \Rightarrow p_2$	$\neg p_1 \vee \neg p_2$	
0	0	1	1	1	$\leftarrow$
0	1	0	1	1	
1	0	1	0	1	
1	1	1	1	0	$\leftarrow$

Há duas valorações que satisfazem simultaneamente ambas as fórmulas de Γ (as correspondentes à 1a e à 4a linha). Verificamos que na 4a linha, a valoração satisfaz todas as fórmulas de Γ, mas não satisfaz ϕ. Logo tem-se que $\Gamma \not\models \phi$.

Exemplo 2.3.17. Sejam $\Gamma = \{p_1 \wedge \neg p_2, p_1 \Rightarrow p_2\}$ e $\phi = p_1 \wedge \neg p_1$. Verifique se $\Gamma \models \phi$.

Resolução. Temos de verificar se, sempre que uma valoração satisfaz simultaneamente todas as fórmulas de Γ, então ela também satisfaz ϕ. Isso pode ser novamente visto através de uma tabela de verdade:

p_1	p_2	$p_1 \wedge \neg p_2$	$p_1 \Rightarrow p_2$	$p_1 \wedge \neg p_1$
0	0	0	1	0
0	1	0	1	0
1	0	1	0	0
1	1	0	1	0

Verificamos que nenhuma valoração satisfaz simultaneamente todas as fórmulas de Γ. Se atendermos à definição de consequência semântica, o que o texto diz é: $\Gamma \models \phi$ sse "Se A então B", onde A ="a valoração v satisfaz simultaneamente todas as fórmulas

de Γ" e B ="a valoração v satisfaz ϕ". Por outras palavras $\Gamma \models \phi$ sse $A \Rightarrow B$. Neste caso particular não existe nenhuma valoração que satisfaça o conjunto Γ, i.e. A é sempre falsa. Logo a implicação $A \Rightarrow B$ será sempre verdadeira. Isto implica que se tenha $\Gamma \models \phi$. Da mesma forma se concluí que $\{\bot\} \models \phi$ para qualquer fórmula ϕ.

Os exemplo dados acima, onde $\Gamma = \{p_1 \wedge \neg p_2, p_1 \Rightarrow p_2\}$ ou $\Gamma = \{\bot\}$, são de alguma forma "degenerados", porque nestas situações é possível concluir ϕ e $\neg\phi$ a partir de Γ, onde ϕ é qualquer fórmula do cálculo proposicional, o que não é natural. Para evitar estes casos, é usual introduzir-se a noção de *consistência*, onde um conjunto de fórmulas é consistente se não é possivel concluir a partir desse conjunto uma fórmula e a sua negação. No caso da lógica proposicional, e utilizando uma perspetiva semântica, os conjuntos consistentes correspondem aos conjuntos satisfazíveis. No entanto, não iremos desenvolver mais este tema nesta disciplina, por razões de falta de tempo.

Iremos utilizar algumas convenções. Se Γ é constituído por uma única fórmula ψ, i.e. $\Gamma = \{\psi\}$, então em vez de escrevermos $\{\psi\} \models \phi$, iremos escrever simplesmente $\psi \models \phi$. Também utilizamos a seguinte convenção: $\varnothing \models \phi$ é o mesmo que $\models \phi$, onde $\varnothing$ é o conjunto vazio.

Nos próximos teoremas, Γ designa um conjunto finito de fórmulas. As suas demonstrações são deixadas como exercício (exercícios 13–15).

Teorema 2.3.18. *$\Gamma \cup \{\psi\} \models \phi$ se e só se $\Gamma \models \psi \Rightarrow \phi$.*

Teorema 2.3.19. *Se $\Gamma \models \phi$, então $\Gamma \cup \{\psi\} \models \phi$ para qualquer fórmula ψ.*

Teorema 2.3.20. *Se $\Gamma \models \phi$ e ψ é uma tautologia, então $\Gamma \backslash \{\psi\} \models \phi$.*

2.3.3 Formas normais e conjuntos adequados

Muitas vezes é útil "normalizar" fórmulas ou outras estruturas. Normalizar uma fórmula significa obter uma fórmula que seja logicamente equivalente à fórmula original, mas que tenha uma determinada estrutura que nos interesse (por exemplo, porque é mais fácil escrever um programa que trabalhe com fórmulas que tenham esta estrutura, etc.).

Vamos agora mostrar que qualquer fórmula do cálculo proposicional pode ser normalizada.

Definição 2.3.21. Um *literal* é uma expressão do tipo p_i ou $\neg p_i$, onde p_i é uma variável proposicional. Uma *conjunção elementar* é uma fórmula com o formato $l_1 \wedge \ldots \wedge l_n$, onde $l_1, \ldots, l_n$ são literais. Uma fórmula em *forma normal disjuntiva* é uma fórmula do tipo $\phi_1 \vee \ldots \vee \phi_m$, onde $\phi_1, \ldots, \phi_m$ são conjunções elementares.

Por exemplo $(p_1 \wedge \neg p_2) \vee p_2$ será uma fórmula em forma normal disjuntiva (as conjunções elementares são $p_1 \wedge \neg p_2$ e p_2. Na última conjunção — p_2 — aparecem 0 conjunções). Do mesmo modo concluímos que $p_1 \wedge \neg p_1$ é também uma fórmula normal disjuntiva (só tem uma conjunção elementar), assim como $p_1 \vee \neg p_1$.

Teorema 2.3.22. *Toda a fórmula do cálculo proposicional é logicamente equivalente a uma fórmula em forma normal disjuntiva.*

Demonstração. Seja ϕ uma fórmula do cálculo proposicional. Se ϕ é uma fórmula contraditória, então esta fórmula é logicamente equivalente à fórmula $p_1 \wedge \neg p_1$ que, como vimos, está em forma normal disjuntiva.

Suponhamos então que ϕ não é uma fórmula contraditória, e sejam $v_1, \ldots, v_k$ as valorações que fazem a fórmula ϕ tomar o valor 1. A cada $i = 1, \ldots, k$, associemos uma conjunção elementar $\phi_i = l_{i,1} \wedge \ldots \wedge l_{i,n}$, onde n é o maior índice que aparece nas variáveis de ϕ e

$$l_{i,j} = \begin{cases} p_j & \text{se } v_i(p_j) = 1 \\ \neg p_j & \text{se } v_i(p_j) = 0 \end{cases}$$

para $j = 1, \ldots, n$. É fácil concluir que $v_i(\phi_i) = 1$ e que a fórmula em forma normal disjuntiva procurada é dada por

$$\phi' = \phi_1 \vee \ldots \vee \phi_k$$

□

Note-se que a demonstração do teorema anterior é construtiva. Por outras palavras, dada uma fórmula ϕ, o procedimento indicado na demonstração do teorema anterior permite-nos determinar uma fórmula em forma normal disjuntiva ϕ' que lhe é logicamente equivalente.

Exemplo 2.3.23. Determine uma fórmula em forma normal disjuntiva logicamente equivalente à fórmula $\phi = p_1 \Rightarrow p_2$.

Resolução. Utilizando o procedimento indicado na demonstração do teorema anterior, primeiro determinamos o valor de ϕ para cada uma das possíveis valorações. Isto pode ser efetuado através de uma tabela de verdade

p_1	p_2	$p_1 \Rightarrow p_2$	
0	0	1	←
0	1	1	←
1	0	0	
1	1	1	←

De seguida identificamos as valorações que satisfazem a fórmula, isto é, identificamos todas as valorações v onde se tenha $v(\phi) = 1$ (estão indicadas na tabela por meio de setas). Do algoritmo indicado na demonstração do teorema anterior, concluímos que

$$p_1 \Rightarrow p_2 \equiv (\neg p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2) \vee (p_1 \wedge p_2)$$

Da mesma forma que fizemos para a forma normal disjuntiva, podemos também definir fórmulas em *forma normal conjuntiva*. Em vez de considerarmos conjunções elementares, devemos tomar *disjunções elementares*, com o formato $p_1 \vee \ldots \vee p_n$, onde $p_1, \ldots, p_n$ são literais. Depois basta definir uma fórmula em forma normal conjuntiva como sendo uma fórmula do tipo $\phi_1 \wedge \ldots \wedge \phi_m$, onde $\phi_1, \ldots, \phi_m$ são disjunções elementares. Também se pode provar facilmente o seguinte teorema (fazer como na demonstração do Teorema 2.3.22, tomando as valorações que fazem a fórmula ϕ ser falsa, e tomando o complemento de $p_{i,j}$).

Teorema 2.3.24. *Toda a fórmula do cálculo proposicional é logicamente equivalente a uma fórmula em forma normal conjuntiva.*

Exemplo 2.3.25. Determine uma fórmula em forma normal conjuntiva logicamente equivalente à fórmula $\phi = \neg(p_1 \Rightarrow p_2)$.

Resolução. Utilizando o procedimento esboçado anteriormente, primeiro determinamos o valor de ϕ para cada uma das possíveis valorações. Isto pode ser feito através de uma tabela de verdade

p_1	p_2	$p_1 \Rightarrow p_2$	$\neg(p_1 \Rightarrow p_2)$	
0	0	1	0	$\leftarrow$
0	1	1	0	$\leftarrow$
1	0	0	1	
1	1	1	0	$\leftarrow$

De seguida identificamos as valorações que fazem com que a fórmula seja interpretada com o valor 0 (estão indicadas na tabela por meio de setas). Do algoritmo esboçado anteriormente, concluímos que

$$\neg(p_1 \Rightarrow p_2) \equiv (p_1 \vee p_2) \wedge (p_1 \vee \neg p_2) \wedge (\neg p_1 \vee \neg p_2)$$

Note-se que esta fórmula está em forma normal conjuntiva.

Acabamos de ver que toda a fórmula pode ser reescrita numa fórmula que lhe é logicamente equivalente e que está em forma normal disjuntiva, ou em forma normal conjuntiva. Note-se que para escrever uma fórmula em forma normal disjuntiva ou em forma normal conjuntiva, apenas precisamos dos símbolos de operador $\neg, \wedge, \vee$, mesmo que a fórmula inicial tenha lá outros símbolos tais como $\Rightarrow$, etc.

Definição 2.3.26. Um conjunto de símbolos de operador O diz-se *adequado* quando toda a fórmula do cálculo proposicional é logicamente equivalente a outra fórmula que utiliza apenas símbolos de operador de O.

Corolário 2.3.27. *O conjunto* $\{\neg, \wedge, \vee\}$ *é adequado.*

Corolário 2.3.28. *Os conjuntos* $\{\neg, \wedge\}$, $\{\neg, \vee\}$ *e* $\{\neg, \Rightarrow\}$ *são adequados.*

Demonstração. Vamos considerar o caso em que o conjunto de símbolos de operador é $\{\neg, \wedge\}$. Seja ϕ uma fórmula do cálculo proposicional. Como o conjunto de símbolos de operador $\{\neg, \wedge, \vee\}$ é adequado, é possível construir uma fórmula ϕ' que utiliza apenas estes símbolos e que satisfaz $\phi' \equiv \phi$. Se agora na fórmula ϕ' substituirmos todas as sub-fórmulas do tipo $\psi_1 \vee \psi_2$ de forma indutiva pela fórmula $\neg(\neg\psi_1 \wedge \neg\psi_2)$ que lhe é logicamente equivalente, obtemos uma nova fórmula ϕ'', e facilmente concluímos que $\phi'' \equiv \phi' \equiv \phi$ e ϕ'' utiliza apenas símbolos de operador $\{\neg, \wedge\}$. Logo o conjunto $\{\neg, \wedge\}$ é adequado.

Um argumento similar mostra que os conjuntos $\{\neg, \vee\}$ e $\{\neg, \Rightarrow\}$ também são adequados. □

Em termos de sistemas digitais, podemos ver um conjunto adequado como sendo um conjunto de portas lógicas universais, no sentido que qualquer porta lógica pode ser implementada utilizando apenas as portas lógicas universais. Por exemplo, qualquer porta lógica pode ser implementada com as portas AND e NOT, uma vez que o conjunto $\{\wedge, \neg\}$ é adequado.

2.4 Tableaux semânticos

Vimos que é possível decidir se uma fórmula é satisfazível através da construção de uma tabela de verdade. No entanto, apesar de conceptualmente simples, este método é pouco eficiente do ponto de vista computacional, já que o número de valorações (linhas) a testar é exponencial no número de variáveis. Por exemplo, numa fórmula com 3 variáveis, teremos de testar $2^3 = 8$ valorações distintas, o que é perfeitamente exequível. No entanto, numa fórmula com 1000 variáveis (um número de variáveis desta grandeza pode facilmente aparecer em diversas aplicações), temos $2^{1000} = (2^{10})^{100} = 1024^{100} \simeq (10^3)^{100} = 10^{300}$ valorações a testar, o que está completamente fora do alcance dos computadores atuais. Em geral, não se conhece nenhum algoritmo que receba como *input* uma fórmula do cálculo proposicional e que permita decidir, de forma eficiente, se essa fórmula é satisfazível, embora haja fortes razões para pensar que tal não é possível (iremos falar mais sobre este tópico no Capítulo 5).

No entanto, há algoritmos (por exemplo baseados na regra da resolução, ou no método dos tableaux semânticos) que em muitos casos são mais eficientes do que o método das tabelas de verdade para decidir se uma fórmula é satisfazível (normalmente a eficiência destes algoritmos depende da estrutura da fórmula). Além do mais, têm a vantagem de também se poderem aplicar (com adaptações e algumas limitações) na lógica de primeira ordem, que iremos estudar no próximo capítulo, ao contrário do método das tabelas de verdade que só funciona para o cálculo proposicional. Nesta secção vamos estudar o método dos tableaux semânticos.

Grosso modo, o método dos tableaux semânticos é mais eficiente quando a fórmula em estudo tem muitas variáveis e não é muito comprida (o tamanho da tabela de verdade cresce exponencialmente com o número de variáveis), enquanto que o método das tabelas

de verdade é mais vantajoso quando o número de variáveis é baixo, independentemente do comprimento da fórmula em estudo.

Seja x_i uma variável proposicional. Ao conjunto $\{x_i, \neg x_i\}$ chama-se *par complementar de literais* (recorde a Definição 2.3.21). É fácil ver que um conjunto de literais é satisfazível se e só se não contém um par complementar de literais. Esta é a ideia básica por detrás dos tableaux semânticos.

Um *tableau semântico* é uma árvore, em que associamos a cada vértice l um conjunto de fórmulas $U(l)$. A raiz da árvore consiste num só vértice, associado ao conjunto formado pela fórmula em estudo, que pretendemos saber se é satisfazível.

Dada uma fórmula do cálculo proposicional, iremos aplicar-lhe um algoritmo (descrito abaixo) que irá gerar um tableau semântico associado a esta fórmula. Este algoritmo vai terminar sempre e, dependendo da estrutura do tableau obtido, iremos concluir que a fórmula é satisfazível ou não. Mais concretamente, depois de obter o tableau, iremos marcar as suas folhas. Uma folha l será marcada como *fechada* ($\times$) se houver um par complementar de literais contido em $U(l)$, e será marcada como *aberta* ($\odot$) se $U(l)$ for constituído apenas por literais e não contiver um par complementar de literais. Diz-se que um tableau é *fechado* se todas as folhas estão marcadas como fechadas. Caso contrário diz-se que o tableau é *aberto*.

A um tableau obtido através da aplicação deste algoritmo damos o nome de *tableau completo*. Uma fórmula ϕ será satisfazível se e só se o tableau obtido pela aplicação do algoritmo a esta fórmula é aberto.

Falta apenas descrever o algoritmo de construção do tableau associado a uma fórmula ϕ.

Algoritmo (construção de um tableau semântico):
Input: uma fórmula ϕ do cálculo proposicional.
Output: um tableau completo associado à fórmula ϕ.

1. Criar uma árvore que consiste num só vértice (a raiz) que está associado ao conjunto $\{\phi\}$.

2. Para cada folha l não marcada da árvore, verificar se $U(l)$ é um conjunto de literais e, caso o seja, marcar essa folha l como: (i) fechada se $U(l)$ contiver um par complementar de literais ou (ii) aberta, caso contrário.

3. Se todas as folhas da árvore estiverem marcadas, então o algoritmo termina com a seguinte conclusão:

   (a) A fórmula ϕ é satisfazível se o tableau obtido é aberto.

   (b) A fórmula ϕ é contraditória se o tableau obtido é fechado.

4. Selecionar uma folha l que não está marcada. Vamos expandir a árvore a partir dessa folha da seguinte forma. Escolhemos uma fórmula ϕ de $U(l)$ que não seja um literal.

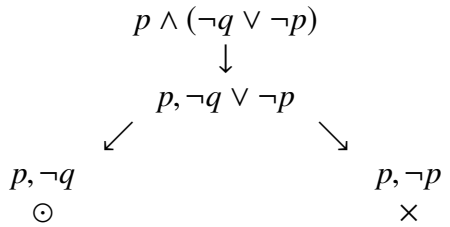
5. Se ϕ é logicamente equivalente a uma fórmula conjuntiva $\phi \equiv \phi_1 \wedge \phi_2$, então criar um vértice descendente l' a partir de l, com $U(l') = (U(l) \cup \{\alpha_1, \alpha_2\}) - \{\phi\}$, onde as fórmulas α_1, α_2 são dadas pela tabela abaixo (no caso de $\neg\neg A$ não existe fórmula α_2), e depois ir para o passo 2.

ϕ	α_1	α_2
$\neg\neg A$	A	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \Rightarrow A_2)$	A_1	$\neg A_2$

6. Se ϕ é logicamente equivalente a uma fórmula disjuntiva $\phi \equiv \phi_1 \vee \phi_2$, então criar dois nós descendentes l' e l'' a partir de l, com $U(l') = (U(l) \cup \{\beta_1\}) - \{\phi\}$ e $U(l'') = (U(l) \cup \{\beta_2\}) - \{\phi\}$, onde as fórmulas β_1, β_2 são dadas pela tabela abaixo, e depois ir para o passo 2.

ϕ	β_1	β_2
$A_1 \vee A_2$	A_1	A_2
$\neg(A_1 \wedge A_2)$	$\neg A_1$	$\neg A_2$
$A_1 \Rightarrow A_2$	$\neg A_1$	A_2

Note-se que este algoritmo não é determinístico, pois a escolha da fórmula a utilizar para definir a expansão de uma folha não marcada é arbitrária. Um exemplo de aplicação pode ser encontrado abaixo.



Como o tableau é aberto, concluímos que $p \wedge (\neg q \vee \neg p)$ é satisfazível (Corolário 2.4.3). Mais, podemos determinar do tableau uma interpretação que satisfaz a fórmula. Esta interpretação é definida pelo ramo que acaba numa folha aberta: $v(p) = 1$ e $v(q) = 0$.

Os seguintes resultados são dados sem demonstração (para mais detalhes ver [Ben12]).

Teorema 2.4.1. *A aplicação do algoritmo anterior para a construção de um tableau semântico termina sempre.*

Teorema 2.4.2 (correção e completude)**.** *Seja ϕ uma fórmula e $\mathcal{T}$ um tableau completo que lhe está associado. ϕ é contraditória se e só se $\mathcal{T}$ é fechado.*

Corolário 2.4.3. *ϕ é satisfazível se e só se $\mathcal{T}$ é aberto.*

Corolário 2.4.4. *$\neg\phi$ é uma tautologia se e só se $\mathcal{T}$ é fechado.*

Em particular, concluímos que o método dos tableaux semânticos nos dá algoritmos para decidir se uma fórmula é satisfazível e se é uma tautologia.

2.5 Sistemas dedutivos (cálculo proposicional)

Tivemos ocasião de ver que é possível ter a noção de consequência lógica utilizando uma perspetiva semântica. Em síntese, para verificar se $\Gamma \models \psi$, basta utilizar uma tabela de verdade, com 2^m linhas, onde m é o número de variáveis distintas que aparecem nas fórmulas $\phi_1, \ldots, \phi_n, \psi$, e proceder como indicado na Definição 2.3.15.

No entanto este tipo de "demonstração" pode ser pouco satisfatório pelas seguintes razões:

1. É inútil quando Γ é infinito. E em muitas aplicações temos potencialmente uma quantidade infinita de informação que podemos utilizar. Por exemplo, imagine que queremos provar um novo resultado da aritmética. Para demonstrar esse resultado podemos utilizar um número infinito de factos (hipóteses) tais como: $1 + 1 = 2$, $2 + 3 = 5$, $a + b = b + a$ para quaisquer números inteiros a e b, etc.

2. Mesmo no caso em que o número de hipóteses é finito, apenas é possível verificar se $\Gamma \models \psi$ em lógicas muito particulares, de que é exemplo o cálculo proposicional. Quando estudarmos a lógica de primeira ordem, iremos ver que nesse caso não é possível verificar se $\Gamma \models \psi$ em tempo finito, mesmo se $\Gamma = \varnothing$

3. Não nos fornece informação sobre onde foi necessário utilizar uma hipótese. Por exemplo se pretendermos provar um resultado envolvendo números primos, gostaríamos de saber onde é que a informação de que os números são primos foi utilizada na demonstração.

Por esta razão, foram desenvolvidas outras abordagens à lógica, que não utilizam semântica, isto é, onde as fórmulas não são interpretadas como sendo verdadeiras ou falsas.

Uma dessas abordagens utiliza *sistemas dedutivos*. Num sistema dedutivo cada fórmula é considerado como uma sequência de símbolos, sem significado. De um conjunto inicial de fórmulas poderemos deduzir novas fórmulas utilizando *regras de inferência*. Nesta abordagem não-semântica não se utilizam valorações. Por esta razão não podemos utilizar resultados obtidos em secções anteriores para mostrar resultados envolvendo sistemas dedutivos.

No entanto, como iremos ver mais adiante, para o cálculo proposicional, a abordagem semântica e a abordagem através de sistemas dedutivos são equivalentes.

2.5.1 Dedução natural

Nesta disciplina iremos estudar sistemas dedutivos utilizando a dedução natural, em que a ênfase é feita na utilização de regras de inferência "naturais". Podem ser encontradas na literatura outras abordagens equivalentes para o estudo de sistemas dedutivos (sistemas de Hilbert, sistemas de Gentzen, etc.). As regras básicas da dedução natural, que serão seguidamente descritas em mais pormenor, são as indicadas na Tabela 2.1.

Em cada uma das regras, as fórmulas que aparecem em cima das barras são as *premissas*, enquanto que a fórmula que aparece em baixo da barra é a *conclusão*. Em geral, irão existir dois tipos de regras: as regras que permitem a introdução de um símbolo de operador (indicadas na coluna do centro da Tabela 2.1), e outras regras que permitem a eliminação desse símbolo (indicadas na coluna da direita da Tabela 2.1). Por exemplo, a regra

$$\frac{\phi \quad \psi}{\phi \wedge \psi}\wedge_i$$

diz-nos que de premissas ϕ e ψ, podemos concluir a fórmula $\phi \wedge \psi$. Reciprocamente a regra

$$\frac{\phi \wedge \psi}{\phi}\wedge_{e_1}$$

diz-nos que da fórmula $\phi \wedge \psi$ podemos concluir a fórmula ϕ. Note-se que ao lado das duas regras estão as siglas "$\wedge_i$" e "$\wedge_{e_1}$" que significam "introdução da conjunção" e "eliminação da conjunção, ficando a 1^a fórmula", respetivamente. Siglas idênticas são utilizadas nas restantes regras.

Definição 2.5.1. Seja Γ um conjunto de fórmulas. Uma dedução a partir de Γ é uma sequência finita de fórmulas $\phi_1, \ldots, \phi_n$ em que cada fórmula dessa sequência é um elemento de Γ ou pode ser deduzida a partir de fórmulas anteriores utilizando uma das regras de inferência da Tabela 2.1. Se existe uma dedução $\phi_1, \ldots, \phi_n$ a partir de Γ, dizemos que a fórmula ϕ_n pode ser deduzida de Γ, e escrevemos $\Gamma \vdash \phi_n$.

Como iremos ver mais adiante, $\vdash$ é a contraparte sintática do operador semântico $\models$. Como no caso semântico, quando Γ for constituído por apenas uma fórmula $\Gamma = \{\psi\}$, escreveremos simplesmente $\psi \vdash \phi$ ao invés de $\{\psi\} \vdash \phi$.

Exemplo 2.5.2. Mostre que $\{p \wedge q, r\} \vdash q \wedge r$.

Resolução. Utilizando as regras de inferência da conjunção, podemos obter a seguinte dedução

$$\frac{\dfrac{p \wedge q}{q}\wedge_{e_2} \quad r}{q \wedge r}\wedge_i$$

Apesar desta dedução em forma de árvore mostrar que $\{p \wedge q, r\} \vdash q \wedge r$, este formato não é muito prático para deduções envolvendo vários passos, porque a árvore pode ocupar

	Introdução	Eliminação
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\wedge_i$	$\dfrac{\phi \wedge \psi}{\phi}\wedge_{e_1}$ $\dfrac{\phi \wedge \psi}{\psi}\wedge_{e_2}$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\vee_{i_1}$ $\dfrac{\psi}{\phi \vee \psi}\vee_{i_2}$	$\dfrac{\phi \vee \psi \quad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline \end{array} \quad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline \end{array}}{\chi}\vee_e$
$\Rightarrow$	$\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \psi \\ \hline \end{array}}{\phi \Rightarrow \psi}\Rightarrow_i$	$\dfrac{\phi \quad \phi \Rightarrow \psi}{\psi}\Rightarrow_e$
$\neg$	$\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \bot \\ \hline \end{array}}{\neg\phi}\neg_i$	$\dfrac{\phi \quad \neg\phi}{\bot}\neg_e$
$\bot$	(ver a regra $\neg_e$)	$\dfrac{\bot}{\phi}\bot_e$
$\neg\neg$	(regra derivada)	$\dfrac{\neg\neg\phi}{\phi}\neg\neg_e$

Tabela 2.1: Regras de inferência para dedução natural (cálculo proposicional).

mais espaço do que o disponível numa folha de papel. Para evitar este problema, vamos escrever a mesma dedução no seguinte formato:

1.	$p \wedge q$	premissa
2.	r	premissa
3.	q	$\wedge_{e_2}$ 1
4.	$q \wedge r$	$\wedge_i$ 3,2

A justificação ao lado da linha 3 quer dizer "aplicar a regra $\wedge_{e_2}$ à fórmula da linha 1", e a justificação ao lado da linha 4 quer dizer "aplicar a regra $\wedge_i$ às fórmulas das linhas 3,2 por esta ordem".

A seguir damos exemplos de como aplicar as restantes regras.

Exemplo 2.5.3. Mostre que $\{p, \neg\neg(q \wedge r)\} \vdash p \wedge r$.

Resolução. Podemos mostrar que $\{p, \neg\neg(q \wedge r)\} \vdash p \wedge r$ através da seguinte dedução:

1.	p	premissa
2.	$\neg\neg(q \wedge r)$	premissa
3.	$q \wedge r$	$\neg\neg_e$ 2
4.	r	$\wedge_{e_2}$ 3
6.	$p \wedge r$	$\wedge_i$ 1,4

Exemplo 2.5.4. Mostre que $\{p \Rightarrow r, p\} \vdash q \Rightarrow r$.

Resolução. Neste exemplo vamos utilizar a regra $\Rightarrow_i$. Na premissa existe uma caixa, em que no topo se encontra a fórmula ϕ, e na parte de baixo se encontra a fórmula ψ. Utilizando essa regra, conclui-se que $\phi \Rightarrow \psi$ (aparece em baixo da caixa). O que quer indicar esta notação? Basicamente diz-nos que, se assumirmos que ϕ é uma hipótese apenas válida dentro da caixa (funciona como uma variável local num programa), e se utilizando regras de inferência e premissas conseguimos chegar à fórmula ψ, então podemos concluir que $\phi \Rightarrow \psi$. Note-se que, fora da caixa, não podemos utilizar a hipótese ϕ, embora possamos depois utilizar a conclusão $\phi \Rightarrow \psi$ (mas só depois de a termos concluído!).

1.	$p \Rightarrow r$	premissa
2.	p	premissa
3.	r	$\Rightarrow_e$ 2, 1
4.	q	hipótese
5.	r	copy 3
6.	$q \Rightarrow r$	$\Rightarrow_i$ 4–5

Note-se que na dedução anterior se utilizou uma regra "copy". Esta regra, tal como o nome indica, apenas copia uma fórmula já obtida numa linha anterior.

O argumento da linha 6 diz que se utilizou a regra $\Rightarrow_i$ das linhas 4 à 5 (por outras palavras, a "fórmula de topo" da caixa é a fórmula da linha 4, $\phi = q$, enquanto que se obtém na linha 5 $\psi = r$. Portanto, da regra $\Rightarrow_i$ pode-se concluir $\phi \Rightarrow \psi$ isto é $q \Rightarrow r$).

É possível encaixar várias regras $\Rightarrow_i$ umas nas outras, de forma semelhante ao que se faz num programa quando se cria uma sub-rotina dentro de outra rotina. As variáveis locais da sub-rotina são apenas acessíveis na sub-rotina, enquanto que as variáveis locais da rotina são acessíveis simultaneamente na rotina e na sub-rotina. Quando encaixamos várias regras $\Rightarrow_i$, as hipóteses funcionam como as variáveis locais: cada variável é apenas acessível dentro da "caixa" onde foi criada (e, eventualmente, nas "sub-caixas" contidas nessa "caixa").

Exemplo 2.5.5. Mostre que $\vdash p \Rightarrow (r \Rightarrow p)$.

Resolução. Note-se que, tal como no caso semântico, a notação $\vdash \phi$ é utilizada como abreviatura de $\varnothing \vdash \phi$, isto é não utilizamos premissas para derivar ϕ. Neste exemplo vamos utilizar duas vezes a regra $\Rightarrow_i$.

1.	p	hipótese
2.	r	hipótese
3.	p	copy 1
4.	$r \Rightarrow p$	$\Rightarrow_i$ 2–3
5.	$p \Rightarrow (r \Rightarrow p)$	$\Rightarrow_i$ 1–4

A utilização das regras para o $\vee$ é semelhante aos casos anteriores. No caso da inclusão, se já tivermos obtido as fórmulas ϕ ou ψ num passo anterior, então podemos concluir a fórmula $\phi \vee \psi$ utilizando as regras $\vee_{i_1}$ ou $\vee_{i_2}$, respetivamente. Na regra da eliminação do $\vee$, é dito que se tivermos a fórmula $\phi \vee \psi$ e se (i) da fórmula ϕ for possível deduzir χ (é o que quer dizer a caixa $\phi \ldots \chi$. Funciona como para o caso da regra $\Rightarrow_i$) (ii) da fórmula ψ for possível deduzir a fórmula χ, então podemos concluir a fórmula χ. Por outras palavras, se sabemos que se tem $\phi \vee \psi$, e que de ϕ ou de ψ conseguimos derivar sempre χ, então de $\phi \vee \psi$ podemos concluir χ.

Exemplo 2.5.6. Mostre que $p \vee q \vdash q \vee p$.

Resolução. Basta utilizar a seguinte derivação

1.	$p \vee q$	premissa
2.	p	hipótese
3.	$q \vee p$	$\vee_{i_2}$ 2
4.	q	hipótese
5.	$q \vee p$	$\vee_{i_1}$ 4
6.	$q \vee p$	$\vee_e$ 1,2–3,4–5

A regra $\perp_e$ diz que da premissa $\perp$ se consegue concluir qualquer fórmula. Em termos semânticos isso equivale ao facto de que $\perp \models \phi$ para qualquer fórmula ϕ (compare com o Exemplo 2.3.17). Se recordarmos a definição de $\perp$, este símbolo designa qualquer fórmula do tipo $\phi \wedge \neg\phi$. É isto que diz a regra $\neg_e$: das premissas ϕ e $\neg\phi$ podemos deduzir $\perp$.

Exemplo 2.5.7. Mostre que $\neg p \vee q \vdash p \Rightarrow q$.

Resolução. Basta utilizar a seguinte derivação

1.	$\neg p \vee q$	premissa
2.	p	hipótese
3.	$\neg p$	hipótese
4.	$\bot$	$\neg_e$ 2,3
5.	q	$\bot_e$ 4
6.	q	hipótese
7.	q	$\vee_e$ 1,3–5,6
8.	$p \Rightarrow q$	$\Rightarrow_i$ 2–7

Finalmente, a regra $\neg_i$ é a *redução ao absurdo*: se de uma hipótese ϕ conseguirmos obter $\bot$ (contradição), então concluímos que $\neg\phi$ terá de ser válida.

Exemplo 2.5.8. Mostre que $\{p \Rightarrow q, p \Rightarrow \neg q\} \vdash \neg p$.

Resolução. Basta utilizar a seguinte derivação

1.	$p \Rightarrow q$	premissa
2.	$p \Rightarrow \neg q$	premissa
3.	p	hipótese
4.	q	$\Rightarrow_e$ 3, 1
5.	$\neg q$	$\Rightarrow_e$ 3, 2
6.	$\bot$	$\neg_e$ 4,5
7.	$\neg p$	$\neg_i$ 3–6

Apesar de a regra de redução ao absurdo ser utilizada na lógica clássica (a estudada nesta disciplina), ela não é admitida numa corrente mais estrita da lógica, a chamada *lógica intuicionista*. Os proponentes desta corrente consideram que um argumento apenas é verdadeiro se existe uma prova construtiva da sua validade. Por exemplo, regras como a demonstração por absurdo, a lei do terceiro excluído (ver secção seguinte), ou a regra $\neg\neg_e$ não são admitidas na lógica intuicionista. Não iremos adotar este ponto de vista nesta disciplina.

2.5.2 Regras deriváveis

Podemos utilizar as regras de inferência da Tabela 2.1 para criar novas regras de inferência. Estas novas regras chamam-se *regras deriváveis*. É possível criar um número infindável de regras deriváveis, mas obviamente que apenas nos interessam regras que sejam úteis na prática. As regras que iremos utilizar são aquelas que constam da Tabela 2.2.

A regra *MT* é chamada de *modus tollens*. A regra *LTE* (lei do terceiro excluído: ou se tem ϕ ou se tem $\neg\phi$) não precisa de premissas para ser aplicada.

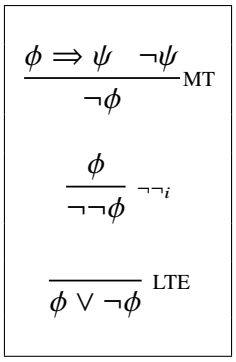


Tabela 2.2: Regras de inferência derivadas para dedução natural (cálculo proposicional).

Teorema 2.5.9. *As regras de inferência da Tabela 2.2 são deriváveis das regras da Tabela 2.1.*

Demonstração. Temos de mostrar que cada uma das regras da Tabela 2.2 pode ser obtida utilizando apenas regras da Tabela 2.1. Vamos mostrar que a regra *modus tollens* pode ser obtida utilizando apenas regras da Tabela 2.1. Para isso queremos, partindo de premissas $\phi \Rightarrow \psi$, $\neg\psi$, chegar à conclusão $\neg\phi$. Isso pode ser feito através da seguinte dedução:

1.	$\phi \Rightarrow \psi$	premissa
2.	$\neg\psi$	premissa
3.	ϕ	hipótese
4.	ψ	$\Rightarrow_e$ 3, 1
5.	$\bot$	$\neg_e$ 4,2
6.	$\neg\phi$	$\neg_i$ 3–5

As outras duas regras podem ser derivadas de forma semelhantes (ver Exercício 29 das folhas teórico-práticas). □

2.5.3 Correção e completude

Já tivemos ocasião de mencionar que são equivalentes no cálculo proposicional a abordagem semântica e a abordagem que utiliza sistemas dedutivos. Por outras palavras, tem-se $\Gamma \models \phi$ se e só se $\Gamma \vdash \phi$. É este resultado que vamos agora mostrar. Como temos uma equivalência, temos duas implicações para demonstrar.

Teorema 2.5.10 (da correção). *Sejam $\psi_1, \ldots, \psi_n, \phi$ fórmulas do cálculo proposicional e $\Gamma = \{\psi_1, \ldots, \psi_n\}$. Se $\Gamma \vdash \phi$, então tem-se $\Gamma \models \phi$.*

Na demonstração do próximo teorema, iremos utilizar efetuar uma *demonstração por indução*. Este método é muito poderoso para mostrar resultados sobre estruturas discretas, que é o tipo de estrutura que tende a aparecer em Informática, como são

exemplos os números inteiros, grafos, árvores, fórmulas do cálculo proposicional, etc. A ideia é sempre a mesma. É preciso mostrar que (i) a propriedade A, que queremos provar, é válida para o caso base, em que os elementos são "básicos" e (ii) se a propriedade é válida para um certo elemento, então ela é válida para o elemento que se lhe "segue" (existe a noção de "elemento seguinte", por a estrutura ser discreta).

Por exemplo, o conjunto $\mathbb{N} = \{1, 2, 3, \ldots\}$ é discreto. Se queremos mostrar uma propriedade A para todos os elementos desse conjunto, utilizando o método de indução, então temos de mostrar que (i) caso-base: a propriedade A é válida para o elemento básico, o número 1 e que (ii) se A é válida para um elemento k (esta assunção chama-se *hipótese de indução*), então A é válida para o elemento que se lhe "segue", i.e. $k+1$. Isto basta para mostrar que a propriedade A é válida para todos os elementos de $\mathbb{N}$. Repare que isto faz sentido. Se mostramos a propriedade para o caso base, então a propriedade A é válida para o número 1. Mas se é válida para o número 1, por ii) será válida para o número 2. Mas se A é válida para o número 2, então novamente por ii), a propriedade A terá de ser válida para o número 3, e por aí fora.

Mas a demonstração por indução não se limitada a número inteiros. Por exemplo, se querermos mostrar uma propriedade A para todas as fórmulas do cálculo proposicional, basta mostrar que: (i) a propriedade A é válida para todas as fórmulas mais "básicas de todas", que são fórmulas constituídas por uma única variável p_i e (ii) se A é válida para fórmulas ϕ, ψ (hipótese de indução), então A é válida para as fórmulas de complexidade "logo acima", i.e. é valida para as fórmulas obtidas adicionando mais um conetivo: $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, e $\phi \Rightarrow \psi$. Se mostrarmos isso, mostramos que a propriedade A é válida para toda a fórmula do cálculo proposicional. Repare que isto faz novamente sentido.

Nesta disciplina iremos utilizar a demonstração por indução para provar vários resultados. A próxima demonstração utiliza indução não sobre $\mathbb{N}$ ou sobre fórmulas, mas sim sobre o número de passos necessários para completar uma dedução (podemos necessitar de 1 ou 2 ou 3 ou 4 ou ... passos para completar uma dedução – ou seja, estamos na presença de uma estrutura discreta, onde a demonstração por indução pode ser utilizada).

Demonstração do Teorema da correção. Como se tem $\Gamma \vdash \phi$, é possível deduzir ϕ a partir das fórmulas $\psi_1, \ldots, \psi_n$ através de uma dedução com um número finito de passos, apenas utilizando as regras de inferência da Tabela 2.1. Vamos demonstrar que se tem $\Gamma \models \phi$ por indução no número de passos utilizados na dedução de $\{\psi_1, \ldots, \psi_n\} \vdash \phi$. Queremos mostrar que a afirmação:

"Se $\{\psi_1, \ldots, \psi_n\} \vdash \phi$ utiliza uma dedução com k passos, então $\{\psi_1, \ldots, \psi_n\} \models \phi$"

é válida para todo o $k \in \mathbb{N}$ e para qualquer fórmula ϕ. Para mostrar, por indução, que a afirmação anterior é válida para todo o $k \in \mathbb{N}$, temos de mostrar que: (i) a afirmação é válida para o caso base, isto é quando $k = 1$; (ii) se a afirmação é válida para todo o $j \in \mathbb{N}$ satisfazendo $j \leq k$, então a afirmação é também válida para uma dedução com $k + 1$ passos.

(i) Caso base: Se a dedução tem um só passo, então ϕ só pode ser uma premissa, i.e.

$\phi = \psi_i$ para algum i. Mas então obviamente que se tem $\psi_i \models \psi_i$, o que implica que $\Gamma \models \psi_i$, ou seja $\Gamma \models \phi$.

(ii) Suponhamos que a dedução $\{\psi_1, \ldots, \psi_n\} \vdash \phi$ tem $k+1$ passos. Queremos mostrar que $\{\psi_1, \ldots, \psi_n\} \models \phi$. Sabemos, por hipótese de indução, que se $\{\psi_1, \ldots, \psi_n\} \vdash \varphi$ utiliza uma dedução com $j \leq k$ passos, então $\{\psi_1, \ldots, \psi_n\} \models \varphi$. Temos dois casos a analisar: (a) ϕ é uma premissa. Este caso já foi tratado no ponto (i); (b) ϕ pode ser obtido por meio de regras de inferência da Tabela 2.1 aplicadas a fórmulas que já tenham sido deduzidas em passos anteriores. Temos agora de ver o que acontece para cada regra da Tabela 2.1:

- Suponhamos que ϕ foi obtida aplicando a regra $\wedge_i$ a duas fórmulas ϕ_1 e ϕ_2 que ocorrem em passos anteriores da dedução. Como estas fórmulas ocorrem em passos anteriores, elas podem ser deduzidas de $\psi_1, \ldots, \psi_n$ em $\leq k$ passos. Logo, *por hipótese de indução*, tem-se que $\Gamma \models \phi_1$ e $\Gamma \models \phi_2$. Mas então é trivial verificar que se tem $\Gamma \models \phi_1 \wedge \phi_2$ i.e. $\Gamma \models \phi$.

- Suponhamos que ϕ foi obtida aplicando a regra $\vee_e$ a uma fórmula $\phi_1 \vee \phi_2$ que ocorre num passo anterior da dedução. Logo, *por hipótese de indução*

$$\Gamma \models \phi_1 \vee \phi_2 \tag{2.2}$$

  Mais, da definição da regra $\vee_e$, sabemos que é possível deduzir ϕ a partir de $\psi_1, \ldots, \psi_n$ e ϕ_1 em $j \leq k$ passos. Por outras palavras, sabemos que a dedução $\{\psi_1, \ldots, \psi_n, \phi_1\} \vdash \phi$ pode ser feita em $j \leq k$ passos. Por hipótese de indução, concluímos que $\{\psi_1, \ldots, \psi_n, \phi_1\} \models \phi$. De forma semelhante, é possível concluir que $\{\psi_1, \ldots, \psi_n, \phi_2\} \models \phi$. Mas estas duas últimas condições e (2.2) implicam que $\{\psi_1, \ldots, \psi_n\} \models \phi$.

- Um raciocínio idêntico pode ser feito para as restantes regras da Tabela 2.1.

□

O teorema da correção é importante para mostrar que uma fórmula ϕ não pode ser derivada a partir de um conjunto de fórmulas Γ.

Exemplo 2.5.11. Mostre que se tem $\nvdash p \wedge (\neg q \vee \neg p)$.

Resolução. Podemos verificar que $p \wedge (\neg q \vee \neg p)$ não é uma tautologia, i.e. não se tem $\models p \wedge (\neg q \vee \neg p)$ através de uma tabela de verdade:

p	q	$p \wedge (\neg q \vee \neg p)$
0	0	0
0	1	0
1	0	1
1	1	0

Logo não se pode ter $\vdash p \wedge (\neg q \vee \neg p)$ porque, caso contrário, pelo teorema da correção ter-se-ia $\models p \wedge (\neg q \vee \neg p)$, o que é um absurdo. Logo $\nvdash p \wedge (\neg q \vee \neg p)$.

Vamos agora mostrar o resultado reciproco do Teorema da correção.

Teorema 2.5.12 (da completude). *Sejam $\psi_1, \ldots, \psi_n, \phi$ fórmulas do cálculo proposicional e $\Gamma = \{\psi_1, \ldots, \psi_n\}$. Se $\Gamma \models \phi$, então tem-se $\Gamma \vdash \phi$.*

Demonstração. Vamos mostrar este resultado em 3 passos. Para isso vamos mostrar que de $\{\psi_1, \ldots, \psi_n\} \models \phi$ podemos efetuar de forma consecutiva os seguintes 3 passos

Passo 1: Mostrar que $\models \psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$
Passo 2: Mostrar que $\vdash \psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$
Passo 3: Mostrar que $\{\psi_1, \ldots, \psi_n\} \vdash \phi$

Passo 1. Este passo é obtido aplicando o Teorema 2.3.18 n vezes a $\{\psi_1, \ldots, \psi_n\} \models \phi$. Aplicando-o uma primeira vez, obtemos

$$\{\psi_1, \ldots, \psi_{n-1}\} \models \psi_n \Rightarrow \phi$$

aplicando-o uma segunda vez obtemos

$$\{\psi_1, \ldots, \psi_{n-2}\} \models \psi_{n-1} \Rightarrow (\psi_n \Rightarrow \phi)$$

e depois de aplicado n vezes o Teorema 2.3.18, obtemos

$$\models \psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$$

Passo 2. Este é o passo mais trabalhoso da demonstração. A ideia é mostrar que para qualquer fórmula φ, se $\models \varphi$ então tem-se $\vdash \varphi$. Se mostrarmos isso, do passo 1 concluímos que

$$\vdash \psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$$

Suponhamos então que φ é uma tautologia. Queremos mostrar que $\vdash \varphi$. Como φ é tautologia, para toda a valoração (linha da tabela verdade), esta fórmula toma o valor 1. Vamos "codificar" cada linha da tabela verdade num conjunto de hipóteses, e mostrar que φ pode ser derivada a partir deste conjunto de hipóteses. Combinando a informação dada por *todas as linhas da tabela verdade*, vai ser então possível mostrar que $\vdash \varphi$. Mas antes temos de trabalhar com cada linha da tabela verdade. Para isso mostramos o seguinte lema.

Lema 2.5.13. *Seja φ uma fórmula do cálculo proposicional. Sejam $p_1, \ldots, p_k$ as variáveis proposicionais que ocorrem na fórmula φ. Seja v uma valoração arbitrária e seja*

$$\hat{p_i} = \begin{cases} p_i & \text{se } v(p_i) = 1 \\ \neg p_i & \text{se } v(p_i) = 0 \end{cases}$$

para $i = 1, \ldots, k$. Então tem-se que:

1. *Se* $v(\varphi) = 1$, *então* $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi$
2. *Se* $v(\varphi) = 0$, *então* $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi$

Demonstração. A demonstração é feita por indução na estrutura da fórmula φ: (i) mostramos que o resultado é válido para uma fórmula básica (só tem uma variável proposicional) e depois (ii) mostramos que se o resultado é válido para as fórmulas φ_1, φ_2, então terá de ser válido para fórmulas proposicionais mais complexas: $\neg\varphi_1$, $\varphi_1 \Rightarrow \varphi_2$, $\varphi_1 \wedge \varphi_2$, e $\varphi_1 \vee \varphi_2$.

(i) Caso base. Se φ é só uma variável proposicional, i.e. $\varphi = p_i$ então temos dois casos a analisar. Se $v(p_i) = 0$, então $\hat{p}_i = \neg p_i$ e obviamente que se tem $\neg p_i \vdash \neg p_i$. Por outras palavras, temos $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi$. Se $v(p_i) = 1$, então $\hat{p}_i = p_i$ e obviamente que se tem $p_i \vdash p_i$. Por outras palavras, temos $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi$.

(ii) Temos de considerar vários símbolos de operador. Vamos começar com o símbolo $\neg$, i.e. suponhamos que $\varphi = \neg\varphi_1$. Primeiro notemos que as variáveis proposicionais de φ são as mesmas que φ_1. Temos mais uma vez dois casos a considerar. Se $v(\varphi) = 0$, então $v(\varphi_1) = 1$. Por *hipótese de indução*, tem-se $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi_1$. Utilizando a regra $\neg\neg_i$ temos $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\neg\varphi_1$ ou seja $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi$, tal como pretendido. Se $v(\varphi) = 1$, então $v(\varphi_1) = 0$. Por *hipótese de indução*, tem-se $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi_1$. Ou seja, tem-se $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi$.
Vamos agora considerar o caso de uma fórmula mais complexa que utiliza o símbolo $\wedge$ i.e. $\varphi = \varphi_1 \wedge \varphi_2$. Se $v(\varphi) = 1$, terá de ser $v(\varphi_1) = 1$ e $v(\varphi_2) = 1$. Por *hipótese de indução* temos então $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi_1$ e $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi_2$. Daqui, utilizando a regra $\wedge_i$, concluímos que $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi_1 \wedge \varphi_2$ i.e. $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi$. Temos agora de analisar o caso em que $v(\varphi) = 0$. Neste caso teremos $v(\varphi_1) = 0$ ou $v(\varphi_2) = 0$: (a) No caso $v(\varphi_1) = v(\varphi_2) = 0$ temos, por *hipótese de indução*, $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi_1$ e $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi_2$. Daqui não é difícil concluir que $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg(\varphi_1 \wedge \varphi_2)$ (Exercício 32 das folhas teórico-práticas) i.e. $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi$; (b) No caso $v(\varphi_1) = 0$ e $v(\varphi_2) = 1$ temos, por *hipótese de indução*, $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi_1$ e também $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi_2$. Daqui não é difícil concluir que $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg(\varphi_1 \wedge \varphi_2)$ (Exercício 32 das folhas teórico-práticas) i.e. $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi$; (c) No caso $v(\varphi_1) = 1$ e $v(\varphi_2) = 0$ temos, por *hipótese de indução*, $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \varphi_1$ e também $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi_2$. Daqui não é difícil concluir que $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg(\varphi_1 \wedge \varphi_2)$ (Exercício 32 das folhas teórico-práticas) i.e. $\{\hat{p}_1, \ldots, \hat{p}_k\} \vdash \neg\varphi$.
Agora que tratamos o caso de fórmulas obtidas utilizando os símbolos $\neg, \wedge$, ainda teríamos de verificar que o resultado continua válido no caso de fórmulas obtidas utilizando os símbolos $\Rightarrow$ e $\vee$. Este caso é deixado como exercício (é semelhante ao caso do símbolo $\wedge$).

□

Agora que temos este lema, vamos então mostrar que se $\models \varphi$, então tem-se $\vdash \varphi$. Para não nos perdermos nos detalhes, vamos considerar o caso em que φ tem só 2 variáveis proposicionais p_1 e p_2, embora um procedimento semelhante possa ser aplicado a fórmulas φ com mais variáveis proposicionais. Como φ é uma tautologia, tem-se sempre $v(\varphi) = 1$ para qualquer valoração v. Logo, o lema anterior implica que

$$\begin{aligned} \{\neg p_1, \neg p_2\} &\vdash \varphi \\ \{\neg p_1, p_2\} &\vdash \varphi \\ \{p_1, \neg p_2\} &\vdash \varphi \\ \{p_1, p_2\} &\vdash \varphi \end{aligned}$$

(no caso em que φ tem k variáveis, teríamos 2^k resultados destes). Vamos "juntar" estas 4 deduções, para concluir que $\vdash \varphi$. Isso pode ser feito através da dedução esquematizada na Tabela 2.3.

Passo 3. Agora que já sabemos que $\vdash \psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$ devido ao passo 2, é óbvio que também se tem

$$\{\psi_1, \ldots, \psi_n\} \vdash \psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$$

Vamos agora tomar as premissas $\psi_1, \ldots, \psi_n$. Utilizando a regra $\Rightarrow_e$, obtemos

1.	ψ_1	premissa
2.	$\psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi)))$	já mostrado
3.	$\psi_2 \Rightarrow (\psi_3 \Rightarrow \ldots (\psi_n \Rightarrow \phi))$	$\Rightarrow_e$ 1,2

Aplicando as restantes premissas $\psi_2, \ldots, \psi_n$ e a regra $\Rightarrow_e$ de forma consecutiva, obtemos $\{\psi_1, \ldots, \psi_n\} \vdash \phi$, como pretendíamos mostrar. □

1.	$p_1 \vee \neg p_1$	LTE
2.	p_1	hipótese
3.	$p_2 \vee \neg p_2$	LTE
4.	p_2	hipótese
⋮	⋮	
6.	φ	(φ pode ser deduzido de p_1 e p_2)
7.	$\neg p_2$	hipótese
⋮	⋮	
9.	φ	(φ pode ser deduzido de p_1 e $\neg p_2$)
10.	φ	$\vee_e$ 3, 4-6,7-9
11.	$\neg p_1$	hipótese
12.	$p_2 \vee \neg p_2$	LTE
13.	p_2	hipótese
⋮	⋮	
15.	φ	(φ pode ser deduzido de $\neg p_1$ e p_2)
16.	$\neg p_2$	hipótese
⋮	⋮	
18.	φ	(φ pode ser deduzido de $\neg p_1$ e $\neg p_2$)
19.	φ	$\vee_e$ 12, 13-15, 16-18
20.	φ	$\vee_e$ 1, 2-10, 11-19

Tabela 2.3: Argumento auxiliar utilizado para mostrar o Teorema da completude para o cálculo proposicional.

Capítulo 3

Lógica de Primeira Ordem

3.1 Linguagens de primeira ordem

No cálculo proposicional utilizamos fórmulas em que as variáveis são (semanticamente) interpretadas como valores lógicos. Apesar de útil, esta abordagem é muitas vezes demasiado limitativa. Por exemplo, como capturar o significado da expressão "para todo o $x \in \mathbb{N}$, tem-se $x \geq 0$" no cálculo proposicional? Poderíamos associar a frase a uma variável proposicional, mas assim perderíamos parte do conteúdo expresso na frase (nomeadamente o facto que a expressão indica uma propriedade sobre números naturais).

Por esta razão, precisamos de lógicas que sejam mais expressivas do que o cálculo proposicional. Nesta secção iremos estudar a *lógica de primeira ordem* (também conhecida como *cálculo de predicado*, ou *cálculo de predicados de primeira ordem*). Na lógica de primeira ordem temos uma linguagem formal mais expressiva, que inclui o uso de *quantificadores* $(\forall, \exists)$. Por outro lado, as definições e resultados para a lógica de primeira ordem tornam-se ligeiramente mais complexos do que para o caso do cálculo proposicional.

Vamos agora ver como podemos obter fórmulas na lógica de primeira ordem. Primeiro, como no caso do cálculo proposicional, precisamos antes de definir o alfabeto utilizado para criar estas fórmulas.

Definição 3.1.1. O *alfabeto* de uma linguagem de primeira ordem $\mathcal{L}$ é constituído pelos seguintes elementos:

1. Variáveis $x_1, x_2, \ldots$ (também podemos designar as variáveis por letras minúsculas $p, q, r, \ldots$), sendo geralmente assumido que conjunto das variáveis é infinito;

2. Símbolos de constante $c_1, c_2, \ldots$;

3. Símbolos de função $f_1^n, f_2^n, \ldots$, para todo o $n \in \mathbb{N}$, onde n é a aridade (= número de argumentos de uma função);

4. Símbolos de predicados: $p_1^n, p_2^n, \ldots$, para todo o $n \in \mathbb{N}$, onde n é a aridade;

5. Símbolos de pontuação: "(", ")", e ",";

6. Conetivos: $\neg$ e $\Rightarrow$;

7. O quantificador $\forall$.

Diz-se que o quantificador $\forall$, os conetivos lógicos $\neg$ e $\Rightarrow$, as variáveis, e os símbolos de pontuação dizem-se os *símbolos lógicos*. Os símbolos de constante, de função, e de predicado dizem-se os *símbolos não-lógicos*.

Há pelo menos duas abordagens diferentes que se podem encontrar na literatura para linguagens de primeira ordem. Numa abordagem assume-se que há uma única linguagem de primeira ordem, em que há uma infinidade de símbolos de constante e, para cada $n \in \mathbb{N}$, existe uma infinidade de símbolos de função e de predicado com aridade n, embora quando se defina uma fórmula apenas se utilize um número finito de símbolos não-lógicos. Na outra abordagem, não se faz qualquer assumpção sobre o número de conetivos não-lógicos podendo, por exemplo, haver apenas um número finito de símbolos de constante, nenhum símbolo de função, etc. Neste segundo caso haverá mais do que uma linguagem de primeira ordem. Nesta disciplina iremos geralmente seguir a segunda abordagem.

Agora que definimos o alfabeto de uma linguagem de primeira ordem, queremos saber como podemos definir uma fórmula. Obviamente, não queremos uma cadeia qualquer de símbolos como $\forall \neg \Rightarrow$ para fórmulas. Em vez disso queremos que tenham uma determinada estrutura. Mas antes de podermos definir o que são fórmulas, temos de introduzir a noção de *termo*.

Definição 3.1.2. O conjunto $Term(\mathcal{L})$ dos *termos* de uma linguagem de primeira ordem $\mathcal{L}$, é o conjunto de todas as cadeias de símbolos definidas pelas seguintes regras:

1. Variáveis e constantes individuais são termos de $\mathcal{L}$;

2. Se f_i^n é um símbolo de função associado a $\mathcal{L}$ e $t_1, \ldots, t_n$ são termos de $\mathcal{L}$, então $f_i^n(t_1, \ldots, t_n)$ é um termo de $\mathcal{L}$.

Como iremos ver mais adiante, os termos são expressões que serão interpretadas como objetos de um dado conjunto, sobre o qual pretendemos fazer raciocínios. Por exemplo, os termos podem significar expressões sobre $\mathbb{N}$. Os símbolos de constante poderão estar associados aos números $0, 1, 2$, e os símbolos de função poderão representar os operadores $+, -, \times$ sobre $\mathbb{N}$. Mas um termo, por si próprio, não pretende ter um valor lógico. Ao invés temos predicados que se aplicam sobre os termos. Exemplos de predicados serão as relações "=", "$\leq$", ou "x é número primo", que retornam os valores lógicos "verdadeiro" ou "falso".

Exemplo 3.1.3. Considere a linguagem de primeira ordem $\mathcal{L}$ em que $C = \{c_1, c_2\}$ é o conjunto dos símbolos de constante, $F = \{f_1^1, f_2^1, f_1^2\}$ é o conjunto dos símbolos de função, e $P = \{p_1^1, p_1^2\}$ é o conjunto dos símbolos de predicado. Então as seguintes expressões são exemplos de termos da linguagem $\mathcal{L}$:

$$x_{10}$$
$$f_2^1(c_2)$$
$$f_1^2(x_{10}, f_2^1(c_2))$$

enquanto que, por exemplo, as seguintes expressões não são termos da linguagem $\mathcal{L}$:

$f_1^2(c_1)$	(falha a aridade de f_1^2)
$(\forall x_{10}) f_2^1(x_{10})$	(termos não podem incluir o símbolo $\forall$)
$x_{10} \Rightarrow f_2^1(c_2)$	(termos não podem incluir o símbolo $\Rightarrow$)

Definição 3.1.4. Se p_i^n é um símbolo de predicado de $\mathcal{L}$ e $t_1, \ldots, t_n$ são termos de $\mathcal{L}$, então $p_i^n(t_1, \ldots, t_n)$ é uma *fórmula atómica* de $\mathcal{L}$. As *fórmulas de* $\mathcal{L}$ são exatamente todas as cadeias de símbolos definidas pelas seguintes regras:

1. Toda a fórmula atómica de $\mathcal{L}$ é uma fórmula de $\mathcal{L}$;

2. Se ϕ e ψ são fórmulas de $\mathcal{L}$, também o são $(\neg\phi)$, $(\phi \Rightarrow \psi)$ e $(\forall x_i \phi)$, onde x_i é uma variável.

Exemplo 3.1.5. Considere novamente a linguagem de primeira ordem $\mathcal{L}$ em que $C = \{c_1, c_2\}$ é o conjunto dos símbolos de constante, $F = \{f_1^1, f_2^1, f_1^2\}$ é o conjunto dos símbolos de função, e $P = \{p_1^1, p_1^2\}$ é o conjunto dos símbolos de predicado. Então as seguintes expressões são exemplos de fórmulas da linguagem $\mathcal{L}$:

$$p_1^1(x_{10})$$
$$p_1^1(f_2^1(c_2))$$
$$((\forall x_{10}) p_1^1(x_{10})) \Rightarrow p_1^2(x_{10}, f_2^1(c_2))$$

enquanto que, por exemplo, as seguintes expressões não são fórmulas da linguagem $\mathcal{L}$:

$p_1^2(c_1)$	(falha a aridade de p_1^2)
$(\forall x_{10}) f_2^1(x_{10})$	(o símbolo $\forall$ deve ser aplicado a uma fórmula)
$f_2^1(c_2)$	(termos não são fórmulas)

Note-se que, de forma semelhante ao que foi feito para o cálculo proposicional, também é possível associar a cada termo ou fórmula de $\mathcal{L}$ uma árvore (no caso das fórmulas, as folhas da árvore correspondem a fórmulas atómicas). Em geral, para

simplificar a notação, iremos assumir que o operador de quantificação tem prioridade sobre os restantes operadores. Por exemplo, escreveremos

$$(\forall x_{10}) p_1^1(x_{10}) \Rightarrow p_1^2(x_{10}, f_2^1(c_2)) \tag{3.1}$$

em vez de $((\forall x_{10}) p_1^1(x_{10})) \Rightarrow p_1^2(x_{10}, f_2^1(c_2))$. Iremos também assumir, tal como no cálculo proposicional, que o operador $\neg$ tem prioridade sobre o operador $\Rightarrow$, e omitimos parêntesis desde que isso não torne a fórmula ambígua (i.e. desde que a fórmula não possa ser interpretada de duas maneiras distintas).

Notas. Neste capítulo, por fórmula entendemos uma fórmula de uma linguagem de primeira ordem. Além disso, de forma semelhante ao que fizemos no cálculo proposicional, vamos incluir outros conetivos, definidos à custa daqueles apresentados na Definição 3.1.1:

1. $(\exists x_i)\phi$ é uma abreviatura para $\neg(\forall x_i)\neg\phi$;
2. $\phi \wedge \psi$ é uma abreviatura para $\neg(\phi \Rightarrow \neg\psi)$;
3. $\phi \vee \psi$ é uma abreviatura para $\neg\phi \Rightarrow \psi$;
4. $\phi \Leftrightarrow \psi$ é uma abreviatura para $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$

Definição 3.1.6. Na fórmula $\forall x_i\, \phi$, dizemos que ϕ é o *âmbito* do quantificador. Mais geralmente, se $(\forall x_i\, \phi)$ é uma sub-fórmula de ψ, então dizemos que o âmbito deste quantificador $(\forall x_i)$ é a fórmula ϕ. Se uma ocorrência da variável x_j não ocorre no âmbito de um quantificador $\forall x_j$, essa ocorrência diz-se *livre*, e caso contrário *ligada* ou *muda* (não são consideradas as variáveis que são utilizadas para definir um quantificador). Uma fórmula diz-se *fechada* se todas as ocorrências de variáveis são ligadas.

Exemplo 3.1.7. O âmbito do quantificador $\forall x_{10}$ na fórmula (3.1) é a fórmula $p_1^1(x_{10})$. A fórmula (3.1) não é fechada, pois tem uma ocorrência de variável livre:

$$(\forall x_{10}) p_1^1(\underbrace{x_{10}}_{ligada}) \Rightarrow p_1^2(\underbrace{x_{10}}_{livre}, f_2^1(c_2))$$

Intuitivamente, e utilizando uma perspetiva semântica, na lógica de primeira ordem os quantificadores apenas podem ser aplicados a variáveis que tomam valores sobre um determinado domínio, como por exemplo $\mathbb{R}$. Isso é suficiente para exprimir muitas propriedades, mas nalguns casos não é suficiente. Por exemplo, se quisermos exprimir numa fórmula a propriedade que todo o conjunto de números reais limitado superiormente tem um supremo, precisamos de utilizar quantificadores que se aplicam a variáveis que tomam valores em $\mathbb{R}$, e a conjuntos de números reais. Quando utilizamos não só quantificadores que se aplicam a elementos do domínio, mas também a conjuntos de elementos do domínio, estamos a falar de lógica de segunda ordem. Se, para além disso, permitirmos quantificar sobre conjuntos de conjuntos de elementos do domínio, estamos no caso da lógica de terceira ordem, etc. No entanto, as lógicas de ordem superior são "menos bem comportadas", e nesta disciplina apenas iremos estudar a lógica de primeira ordem.

3.2 Semântica

3.2.1 Noções básicas

Tal como no cálculo proposicional, quando trabalhamos com semântica, queremos "interpretar" as fórmulas, ou seja queremos dar-lhe um significado. Em particular, gostaríamos de dizer que uma fórmula é "verdadeira" ou "falsa". Mais, gostaríamos que símbolos para variáveis, constantes, e funções pudessem ser interpretados como variáveis, constantes, e funções sobre um domínio (por exemplo $\mathbb{N}$), respetivamente. Isso pode ser feito utilizando a noção de *interpretação*.

Definição 3.2.1. Uma *interpretação* (ou *modelo*) I de uma linguagem de primeira ordem $\mathcal{L}$ é um tuplo (D_I, C_I, F_I, P_I), onde:

1. D_I é um conjunto não-vazio, chamado de *domínio* da interpretação;
2. $C_I = ([c_1]_I, [c_2]_I, \ldots)$ é uma coleção de constantes em D_I, i.e. $[c_i]_I \in D_I$ para todo o $i \in \mathbb{N}$;
3. F_I é uma coleção de funções $[f_i^n]_I : D_I^n \to D_I$, onde $i, n \in \mathbb{N}$;
4. P_I é uma coleção de predicados $[p_i^n]_I \subseteq D_I^n$, onde $i, n \in \mathbb{N}$ (nota: se $x \in D_I^n$ e $x \in [p_i^n]_I$, então diz-se que $[p_i^n]_I(x)$ é verdadeira. Se $x \notin [p_i^n]_I$, então diz-se que $[p_i^n]_I(x)$ é falsa).

Exemplo 3.2.2. Considere novamente a linguagem de primeira ordem $\mathcal{L}$ em que $C = \{c_1, c_2\}$ é o conjunto dos símbolos de constante, $F = \{f_1^1, f_2^1, f_1^2\}$ é o conjunto dos símbolos de função, e $P = \{p_1^1, p_1^2\}$ é o conjunto dos símbolos de predicado. Uma possível interpretação para esta linguagem é a interpretação $I = (\mathbb{R}, C_I, F_I, P_I)$, onde:

$$[c_1]_I = 0,\ [c_2]_I = \pi,\ [f_1^1]_I(x) = x^2,\ [f_2^1]_I(x) = x,\ [f_1^2]_I(x, y) = x \times y,$$
$$[p_1^1]_I(d) = \begin{cases} 0 & \text{se } -1 < d < 1 \\ 1 & \text{caso contrário} \end{cases}, \qquad [p_1^2]_I(d_1, d_2) = \begin{cases} 0 & \text{se } d_1 \neq d_2 \\ 1 & \text{se } d_1 = d_2 \end{cases}.$$

Exemplo 3.2.3. Outra possível interpretação para a linguagem $\mathcal{L}$ considerada no exemplo em cima é a interpretação $I' = (D_{I'}, A_{I'}, F_{I'}, P_{I'})$, onde $D_{I'} = \{azul, verde\}$ e:

$$[c_1]_{I'} = azul,\ [c_2]_{I'} = verde,\ [f_1^1]_{I'}(x) = azul,\ [f_2^1]_{I'}(x) = azul,\ [f_1^2]_{I'}(x, y) = x,$$
$$[p_1^1]_{I'}(d) = \begin{cases} 0 & \text{se } d = verde \\ 1 & \text{se } d = azul \end{cases}, \qquad [p_1^2]_{I'}(d_1, d_2) = \begin{cases} 0 & \text{se } d_1 = d_2 \\ 1 & \text{se } d_1 \neq d_2 \end{cases}.$$

Como verificamos neste exemplo, a uma linguagem de primeira ordem podem estar associadas várias interpretações. No entanto uma interpretação não dá valores às variáveis x_i. Para isso necessitamos da seguinte definição.

Definição 3.2.4. Uma *valoração* em I é uma função $v : Term(\mathcal{L}) \to D_I$ com as seguintes propriedades:

(i) $v(c_i) = [c_i]_I$ para todo o $i \in \mathbb{N}$;

(ii) $v(f_i^n(t_1, \ldots, t_n)) = [f_i^n]_I(v(t_1), \ldots, v(t_n))$, para todos os $i, n \in \mathbb{N}$, e para quaisquer termos $t_1, \ldots, t_n$ de $\mathcal{L}$.

Não é difícil ver, por indução na estrutura do termo, que dada uma interpretação I, basta saber os valores de uma valoração v nas variáveis $x_1, x_2, \ldots$ para saber o valor da valoração em qualquer termo de $Term(\mathcal{L})$, bastando para tal utilizar as regras (i) e (ii) da Definição 3.2.4.

Exemplo 3.2.5. Se considerarmos a linguagem $\mathcal{L}$ do exemplo anterior e as interpretações I e I', então um exemplo de valoração v em I será a valoração definida por

$$v(x_1) = 1,\ v(x_2) = 2,\ \ldots,\ v(x_i) = i,\ \ldots$$

enquanto que um exemplo de valoração v' em I' será

$$v'(x_i) = verde,\ \ i = 1, 2, \ldots$$

Escolhendo, por exemplo, o termo $t = f_1^2(f_1^1(x_2), c_1)$, tem-se

$$v(f_1^2(f_1^1(x_2), c_1)) = [f_1^2]_I([f_1^1]_I(v(x_2)), [c_1]_I) = [f_1^2]_I([f_1^1]_I(2), 0) = [f_1^2]_I(4, 0) = 0$$
$$v'(f_1^2(f_1^1(x_2), c_1)) = [f_1^2]_{I'}([f_1^1]_{I'}(v'(x_2)), [c_1]_{I'}) = [f_1^2]_{I'}([f_1^1]_{I'}(verde), azul) = azul$$

Definição 3.2.6. Dada uma valoração v e um $d \in D_I$, $v[x_i \leftarrow d]$ designa a valoração definida por

$$v[x_i \leftarrow d](x_j) = \begin{cases} v(x_j) & \text{se } i \neq j \\ d & \text{se } i = j. \end{cases}$$

Por outras palavras $v[x_i \leftarrow d]$ é idêntica a v, exceto em x_i onde retorna o valor d.

Exemplo 3.2.7. Nas condições do exemplo anterior, temos

$$v[x_2 \leftarrow 5](x_1) = 1,\ v[x_2 \leftarrow 5](x_2) = 5,\ v[x_2 \leftarrow 5](x_3) = 3, \ldots,\ v(x_i) = i,\ \ldots$$

Definição 3.2.8. Seja ϕ uma fórmula de $\mathcal{L}$ e seja v uma valoração em I, onde I é uma interpretação. Diz-se que uma valoração v *satisfaz* ϕ, denotado por $\models_I^v \phi$ (muitas vezes escreveremos simplesmente $\models^v \phi$ pois a uma valoração está sempre subjacente uma interpretação), se (definição por indução na estrutura de ϕ):

a) $\phi = p_i^n(t_1, \ldots, t_n)$ é uma fórmula atómica e $[p_i^n]_I(v(t_1), \ldots, v(t_n))$ é verdadeira;

b) $\models^v \neg\phi$ se e só se $\not\models_I^v \phi$;

c) $\models^v \phi \Rightarrow \psi$ se e só se $\not\models^v \phi$ ou $\models^v \psi$;

d) $\models^v \forall x_i \phi$ se e só se $\models^{v[x_i \leftarrow d]} \phi$ para todo o $d \in D_I$.

É fácil verificar da definição anterior e das definições de $\vee, \wedge$ e de $\Leftrightarrow$, que também se tem:

1. $\models^v \phi \vee \psi$ se e só se $\models^v \phi$ ou $\models^v \psi$;
2. $\models^v \phi \wedge \psi$ se e só se$\models^v \phi$ e $\models^v \psi$;
3. $\models^v \phi \Leftrightarrow \psi$ se e só se ($\models^v \phi$ se e só se $\models^v \psi$).

Exemplo 3.2.9. Nas condições do Exemplo 3.2.5, verifique se $\models^v p_1^1(x_2) \Rightarrow p_1^1(x_1)$, onde x_1, x_2 são variáveis da linguagem $\mathcal{L}$.

Resolução. Temos que $\models^v p_1^1(x_2) \Rightarrow p_1^1(x_1)$ se e só se $\not\models^v p_1^1(x_2)$ ou $\models^v p_1^1(x_1)$ se e só se $[p_1^1]_I(v(x_2)) = 0$ ou $[p_1^1]_I(v(x_1)) = 1$ se e só se $[p_1^1]_I(2) = 0$ ou $[p_1^1]_I(1) = 1$. Ora, como $[p_1^1](1) = 1$, a última condição ($[p_1^1]_I(2) = 0$ ou $[p_1^1]_I(1) = 1$) é verdadeira e, voltando atrás através da sequência de "se e só se", concluímos que a condição $\models^v p_1^1(x_2) \Rightarrow p_1^1(x_1)$ é verdadeira.

Exemplo 3.2.10. Nas condições do Exemplo 3.2.5, verifique se $\models^v \forall x_1 p_1^1(f_1^1(x_1))$, onde x_1 é uma variável da linguagem $\mathcal{L}$.

Resolução. Temos que $\models^v \forall x_1 p_1^1(f_1^1(x_1))$ se e só se $\models^{v[x_1 \leftarrow d]} p_1^1(f_1^1(x_1))$ para todo o $d \in D_I$ se e só se $[p_1^1]_I([f_1^1]_I(v[x_1 \leftarrow d](x_1))) = 1$ para todo o $d \in \mathbb{R}$ se e só se $[p_1^1]_I([f_1^1]_I(d)) = 1$ para todo o $d \in \mathbb{R}$ se e só se $[p_1^1]_I(d^2) = 1$ para todo o $d \in \mathbb{R}$ se e só se para todo o $d \in \mathbb{R}$ não se tem $-1 < d^2 < 1$. Mas a última condição (para todo o $d \in \mathbb{R}$ não se tem $-1 < d^2 < 1$) é obviamente falsa, pois existem $d \in \mathbb{R}$ tais que $-1 < d^2 < 1$ (um exemplo pode ser obtido tomando $d = 0$). Logo, como obtivemos uma condição falsa a partir da condição $\models^v \forall x_1 p_1^1(f_1^1(x_1))$ através de "se e só se", concluímos que $\not\models^v \forall x_1 p_1^1(f_1^1(x_1))$.

Teorema 3.2.11. *$\models^v (\exists x_i)\phi$ se e só se existe um $d \in D_I$ tal que $\models^{v[x_i \leftarrow d]} \phi$.*

Demonstração. $\models^v (\exists x_i)\phi$ sse (= se e só se) $\models^v \neg(\forall x_i)\neg\phi$ sse $\not\models^v (\forall x_i)\neg\phi$ sse existe um $d \in D_I$ tal que $\not\models^{v[x_i \leftarrow d]} \neg\phi$ sse existe um $d \in D_I$ tal que $\models^{v[x_i \leftarrow d]} \phi$. □

Exemplo 3.2.12. Nas condições do Exemplo 3.2.5, verifique se $\models^v \exists x_1 p_1^1(f_1^1(x_1))$, onde x_1 é uma variável da linguagem $\mathcal{L}$.

Resolução. Temos que $\models^v \exists x_1 p_1^1(f_1^1(x_1))$ se e só se $\models^{v[x_1 \leftarrow d]} p_1^1(f_1^1(x_1))$ para algum $d \in D_I$ se e só se $[p_1^1]_I([f_1^1]_I(d)) = 1$ para algum $d \in \mathbb{R}$ se e só se $[p_1^1]_I(d^2) = 1$ para algum $d \in \mathbb{R}$ se e só se para algum $d \in \mathbb{R}$ não se tem $-1 < d^2 < 1$. Mas a última condição (para algum $d \in \mathbb{R}$ não se tem $-1 < d^2 < 1$) é verdadeira, pois existem $d \in \mathbb{R}$ tais que não se tem $-1 < d^2 < 1$ (um exemplo pode ser obtido tomando $d = 2$). Logo, como obtivemos uma condição verdadeira a partir da condição $\models^v \exists x_1 p_1^1(f_1^1(x_1))$ através de "se e só se", então a condição $\models^v \exists x_1 p_1^1(f_1^1(x_1))$ é também verdadeira.

Definição 3.2.13. Uma fórmula ϕ diz-se *válida numa interpretação I*, o que se denota por $\models_I \phi$, se para toda a valoração v em I se tem $\models_I^v \phi$. Neste caso diz-se que I é um *modelo* para ϕ.

Exemplo 3.2.14. Utilizando a interpretação I definida no Exemplo 3.2.2, verifique se $\models_I p_1^1(x_1) \Rightarrow p_1^1(x_1)$, onde x_1 é uma variável da linguagem $\mathcal{L}$.

Resolução. Seja v uma valoração arbitrária. Temos de mostrar que para toda a valoração v em I se tem $\models_I^v p_1^1(x_1) \Rightarrow p_1^1(x_1)$. Tem-se $\models_I^v p_1^1(x_1) \Rightarrow p_1^1(x_1)$ se e só se $\not\models_I^v p_1^1(x_1)$ ou $\models_I^v p_1^1(x_1)$. Ora, independentemente da escolha da valoração v, a última afirmação é sempre verdadeira. Logo, voltando atrás através da sequência de "se e só se", concluímos que a condição $\models_I^v p_1^1(x_1) \Rightarrow p_1^1(x_1)$ é verdadeira para toda a valoração v em I. Portanto tem-se que $\models_I p_1^1(x_1) \Rightarrow p_1^1(x_1)$.

Definição 3.2.15. Uma fórmula ϕ diz-se *logicamente válida*, escrevendo-se $\models \phi$, se $\models_I \phi$ para toda a interpretação I.

Exemplo 3.2.16. Verifique se $\models p_1^1(x_1) \Rightarrow p_1^1(x_1)$, onde x_1 é uma variável da linguagem $\mathcal{L}$.

Resolução. Seja I uma interpretação arbitrária. Temos de mostrar que se tem $\models_I p_1^1(x_1) \Rightarrow p_1^1(x_1)$. Isso pode ser feito mostrando que para toda a valoração v em I se tem $\models_I^v p_1^1(x_1) \Rightarrow p_1^1(x_1)$. Mas $\models_I^v p_1^1(x_1) \Rightarrow p_1^1(x_1)$ se e só se $\not\models_I^v p_1^1(x_1)$ ou $\models_I^v p_1^1(x_1)$. Ora, independentemente da valoração v, a última afirmação é sempre verdadeira ($[p_1^1](v(x_1))$ é falsa ou $[p_1^1](v(x_1))$ é verdadeira). Logo, voltando atrás através da sequência de "se e só se", concluímos que a condição $\models_I^v p_1^1(x_1) \Rightarrow p_1^1(x_1)$ é sempre verdadeira. Como este resultado é válido para toda a valoração v em I, concluímos que $\models_I p_1^1(x_1) \Rightarrow p_1^1(x_1)$. Mais, este resultado também não depende da escolha da interpretação I, pelo que concluímos que a fórmula $p_1^1(x_1) \Rightarrow p_1^1(x_1)$ é logicamente válida: $\models p_1^1(x_1) \Rightarrow p_1^1(x_1)$.

Se repararmos no exemplo anterior, a fórmula $p_1^1(x_1) \Rightarrow p_1^1(x_1)$ tem o mesmo formato da fórmula $\phi \Rightarrow \phi$ do cálculo proposicional, que é uma tautologia (é fácil ver isto através de uma tabela de verdade e utilizando o Teorema 2.3.13). Este resultado não é pura coincidência: a partir de tautologias do cálculo proposicional, podemos construir novas fórmulas da lógica de primeira ordem, que serão sempre logicamente válidas.

Definição 3.2.17. Uma *tautologia* em $\mathcal{L}$ é uma fórmula obtida a partir de uma tautologia do cálculo proposicional, substituindo as suas variáveis por fórmulas de $\mathcal{L}$.

Teorema 3.2.18. *Toda a tautologia em $\mathcal{L}$ é logicamente válida.*

Demonstração. Seja ϕ uma tautologia em $\mathcal{L}$. Por ser tautologia, ϕ pode ser obtida a partir de uma tautologia ϕ_0 do cálculo proposicional, com variáveis $p_1, \ldots, p_n$, substituindo as variáveis $p_1, \ldots, p_n$ na fórmula ϕ_0 por $\psi_1, \ldots, \psi_n \in \mathcal{L}$, respetivamente. Seja I uma

interpretação e v uma valoração em I. Vamos mostrar que $\models^v \phi$. Definimos uma valoração v' para a fórmula ϕ_0 do seguinte modo

$$v'(p_i) = \begin{cases} 1 & \text{se } \models^v \psi_i \\ 0 & \text{se } \not\models^v \psi_i \end{cases}$$

Vamos agora mostrar que $v'(\phi_0) = 1$ sse $\models^v \phi$. Como ϕ_0 é uma tautologia por hipótese, então terá de ser sempre $v'(\phi_0) = 1$ e logo concluiremos $\models^v \phi_1$, o que mostra o teorema. Falta só mostrar que

$$v'(\phi_0) = 1 \quad \text{sse} \quad \models^v \phi \tag{3.2}$$

A prova é feita por indução na estrutura de ϕ_0.
Caso base. $\phi_0 = p_i$. Então só pode ser $\psi_i = \phi$ e, por definição de v', $v'(\phi_0) = v'(p_i) = 1$ sse $\models^v \psi_i$ i.e. $\models^v \phi$;
Passo de indução. Suponhamos que a propriedade (3.2) é válida para fórmulas φ, χ. Temos de mostrar que (3.2) é válida para as fórmulas de complexidade "logo acima" i.e. temos de mostrar que é válida para $\neg\varphi$ e $\varphi \Rightarrow \chi$:
i) Seja $\phi = \neg\varphi$. Então $\phi_0 = \neg\varphi_0$, onde φ é obtido a partir de φ_0 substituindo $p_1, \ldots, p_n$ por $\psi_1, \ldots, \psi_n$. Por hipótese de indução, $v'(\varphi_0) = 1$ sse $\models^v \varphi$. Logo $v'(\phi_0) = 1$ sse $v'(\neg\varphi_0) = 1$ sse $v'(\varphi_0) = 0$ sse $\not\models^v \varphi$ sse $\models^v \neg\varphi$ sse $\models^v \phi$;
ii) $\phi_0 = \psi_0 \Rightarrow \varphi_0$. Semelhante ao caso anterior e deixado como exercício. □

Exemplo 3.2.19. Já vimos que se tem, no cálculo proposicional, $\models p_1 \vee \neg p_1$, onde p_1 é uma variável do cálculo proposicional. Substituindo a variável p_1 pela fórmula $p_1^1(x_1)$, onde x_1 é uma variável da linguagem $\mathcal{L}$, concluímos do teorema anterior que a fórmula $p_1^1(x_1) \vee \neg p_1^1(x_1)$ é logicamente válida.

Devido ao resultado anterior, e se $x_1, x_2, \ldots$ forem as variáveis da linguagem de primeira ordem, faz sentido introduzir novamente os símbolos $\bot, \top$, que no caso da lógica de primeira ordem são utilizados para representar as seguintes fórmulas:

$$\bot = p_1 \wedge \neg p_1$$
$$\top = p_1 \vee \neg p_1$$

Em particular, tem-se $\models \top$ e $\not\models_I^v \bot$ para qualquer interpretação I e qualquer valoração v em I.

Notamos também que, embora uma tautologia em $\mathcal{L}$ seja uma fórmula logicamente válida, o resultado recíproco não se verifica.

3.2.2 Fórmulas logicamente equivalentes

Definição 3.2.20. Sejam ϕ e ψ duas fórmulas. Se $\models_I^v \phi$ sse $\models_I^v \psi$ para toda a valoração v em I, onde I é uma interpretação arbitrária, então diz-se que ϕ e ψ são *logicamente equivalentes*, escrevendo-se $\phi \equiv \psi$.

Exemplo 3.2.21. Sejam ϕ, ψ fórmulas de uma linguagem de primeira ordem em que x_i não é uma variável livre na fórmula ψ. Mostre que se tem $\forall x_i \phi(x_i) \vee \psi \equiv \forall x_i(\phi(x_i) \vee \psi)$.

Resolução. Seja v uma valoração qualquer em I, onde I uma interpretação arbitrária. Temos que $\models_I^v \forall x_i \phi(x_i) \vee \psi$ se e só se (sse) $\models_I^v \forall x_i \phi(x_i)$ ou $\models_I^v \psi$ sse para todo o $d \in D_I$ se tem $\models_I^{v[x_i \leftarrow d]} \phi(x_i)$ ou $\models_I^v \psi$. Como x_i não é uma variável livre na fórmula ψ, podemos alterar o valor da variável x_i na valoração v, sem que isso afete o facto de ψ ser satisfeita ou não por essa valoração. Por outras palavras, $\models_I^v \psi$ se e só se $\models_I^{v[x_i \leftarrow d]} \psi$. Retomando o raciocínio que iniciamos, temos que $\models_I^v \forall x_i \phi(x_i) \vee \psi$ sse para todo o $d \in D_I$ se tem $\models_I^{v[x_i \leftarrow d]} \phi(x_i)$ ou $\models_I^{v[x_i \leftarrow d]} \psi$ sse para todo o $d \in D_I$ se tem $v \models_I^{v[x_i \leftarrow d]} \phi(x_i) \vee \psi$ sse para toda a valoração v em I se tem $\models_I^v \forall x_i(\phi(x_i) \vee \psi)$.

Por outras palavras, tem-se $\models_I^v \forall x_i \phi(x_i) \vee \psi$ só e só se $\models_I^v \forall x_i(\phi(x_i) \vee \psi)$ para toda a valoração v em I, onde I é uma interpretação arbitrária. Portanto $\forall x_i \phi(x_i) \vee \psi \equiv \forall x_i(\phi(x_i) \vee \psi)$.

Teorema 3.2.22. *Para quaisquer fórmulas ϕ e ψ, tem-se*

$$\phi \equiv \psi \quad sse \quad \models \phi \Leftrightarrow \psi$$

Demonstração. Tem-se que $\models \phi \Leftrightarrow \psi$ sse $\models_I^v \phi \Leftrightarrow \psi$ para toda a valoração v em I, onde I é uma interpretação arbitrária sse ($\models_I^v \phi$ sse $\models_I^v \psi$) para toda a valoração v em I sse $\phi \equiv \psi$. □

O último teorema ajuda-nos em exercícios: permite-nos reduzir um operador de equivalência a uma equivalência lógica, sendo esta última mais fácil de verificar.

Exemplo 3.2.23. Sejam ϕ, ψ fórmulas de uma linguagem de primeira ordem em que x não é uma variável livre na fórmula ψ. Mostre que se tem $\models (\forall x_i \phi(x_i) \vee \psi) \Leftrightarrow (\forall x_i(\phi(x_i) \vee \psi))$.

Resolução. Vimos no exemplo anterior que $\forall x_i \phi(x_i) \vee \psi \equiv \forall x_i(\phi(x_i) \vee \psi)$. Portanto, concluímos automaticamente do Teorema 3.2.22 que se tem

$$\models (\forall x_i \phi(x_i) \vee \psi) \Leftrightarrow (\forall x_i(\phi(x_i) \vee \psi))$$

Apresentamos seguidamente outros exemplos de fórmulas logicamente equivalentes, cuja demonstração é omitida (algumas são pedidas como exercícios nas folhas das aulas teórico-práticas). Note-se que ϕ e ψ são fórmulas. Se aparecerem no âmbito de um quantificador Qx, então a expressão $\phi(x)$ quer dizer que x é uma variável livre em ϕ (mas fechada em $Qx\phi(x)$), e se escrevemos apenas ϕ, quer dizer que a variável x não

aparece livre em ϕ. Esta notação é apenas usada aqui.

$$
\begin{array}{ll}
\forall x\phi(x) \equiv \neg\exists x\neg\phi(x) & \text{(relação entre } \forall \text{ e } \exists\text{)} \\
\exists x\phi(x) \equiv \neg\forall x\neg\phi(x) & \\
 & \\
\forall x\forall y A(x,y) \equiv \forall y\forall x A(x,y) & \text{(comutatividade de quantificadores)} \\
\exists x\exists y A(x,y) \equiv \exists y\exists x A(x,y) & \\
 & \\
(\forall x\phi(x) \vee \psi) \equiv \forall x(\phi(x) \vee \psi) & \text{(extracção de quantificadores} \\
(\psi \vee \forall x\phi(x)) \equiv \forall x(\psi \vee \phi(x)) & \text{em disjunções)} \\
(\exists x\phi(x) \vee \psi) \equiv \exists x(\phi(x) \vee \psi) & \qquad (3.3) \\
(\psi \vee \exists x\phi(x)) \equiv \exists x(\psi \vee \phi(x)) & \\
\exists x\phi(x) \vee \exists x\psi(x) \equiv \exists x(\phi(x) \vee \psi(x)) & \\
 & \\
(\forall x\phi(x) \wedge \psi) \equiv \forall x(\phi(x) \wedge \psi) & \text{(extracção de quantificadores} \\
(\psi \wedge \forall x\phi(x)) \equiv \forall x(\psi \wedge \phi(x)) & \text{em conjunções)} \\
(\exists x\phi(x) \wedge \psi) \equiv \exists x(\phi(x) \wedge \psi) & \\
(\psi \wedge \exists x\phi(x)) \equiv \exists x(\psi \wedge \phi(x)) & \\
\forall x\phi(x) \wedge \forall x\psi(x) \equiv \forall x(\phi(x) \wedge \psi(x)) &
\end{array}
$$

Nota. Até agora temos sempre utilizado um índice na variável. Mas em vez de chamarmos $x_1, x_2, \ldots$ às variáveis que vamos utilizar, recordamos que podemos chamá-las, por exemplo, de $a, b, c, \ldots$. Da mesma forma, em vez de chamar x_i a uma variável genérica, podemos atribuir-lhe a designação de x. Isso não altera em nada os resultados. Assim, o resultado do exemplo anterior pode ser reescrito como

$$\models (\forall x\phi(x) \vee \psi) \Leftrightarrow (\forall x(\phi(x) \vee \psi))$$

3.2.3 Consequência semântica

Definição 3.2.24. Seja ψ uma fórmula e Γ um conjunto de fórmulas. Diz-se que ψ é *consequência semântica* de Γ, escrevendo-se $\Gamma \models \psi$, se para toda a interpretação I e para toda a valoração v em I tal que $\models_I^v \phi$ para todo o $\phi \in \Gamma$, se tem $\models_I^v \psi$.

Tal como fizemos no cálculo proposicional, quando Γ for constituído por uma só fórmula, i.e. $\Gamma = \{\phi\}$, escreveremos $\phi \models \psi$ em vez de $\{\phi\} \models \psi$, e quando $\Gamma = \emptyset$, tem-se $\emptyset \models \psi$ sse $\models \psi$.

Exemplo 3.2.25. Sejam ϕ uma fórmula de uma linguagem de primeira ordem. Mostre que se tem $\forall x\, \phi \models \phi$

Resolução. Temos de verificar se para toda a interpretação I e para toda a valoração v em I tal que $\models_I^v \forall x\, \phi$, se tem $\models_I^v \phi$. Ora $\models_I^v \forall x\, \phi$ sse para todo o $d \in D_I$ se tem $\models_I^{v[x \leftarrow d]} \phi$. Em particular, terá de se ter $\models_I^v \phi$. Logo $\forall x\phi \models \phi$.

3.2.4 Fórmulas prenexas

Vamos agora mostrar que, na lógica de primeira ordem, existe um formato padrão no qual qualquer fórmula pode ser reescrita, de forma semelhante ao que acontece com as fórmulas normais disjuntivas (ou conjuntivas) para o cálculo proposicional.

Definição 3.2.26. Uma fórmula ϕ de uma linguagem de primeira ordem diz-se em forma *prenexa* se é escrita como

$$Q_1 x_{i_1} \dots Q_n x_{i_n} \psi,$$

onde cada $Q_1, \dots, Q_n$ é o quantificador $\forall$ ou $\exists$, e ψ é uma fórmula sem quantificadores, designada de *matriz*.

Teorema 3.2.27. *Para toda a fórmula ϕ de uma linguagem de primeira ordem $\mathcal{L}$, existe uma fórmula em forma prenexa ψ tal que $\phi \equiv \psi$.*

Demonstração. A demonstração deste teorema é construtiva, i.e. dá-nos um algoritmo que nos permite converter ϕ em $Q_1 x_{i_1} \dots Q_n x_{i_n} \psi$. Vamos exemplificar os passos da demonstração com um exemplo, embora estes passos sejam válidos para qualquer fórmula de $\mathcal{L}$. Para simplificar a notação, vamos assumir que p e q são símbolos de predicados que admitem um só argumento, e vamos tomar como exemplo a fórmula

$$\phi = \forall x_1 (p(x_1) \Rightarrow q(x_1)) \Rightarrow (\forall x_1 p(x_1) \Rightarrow \forall x_1 q(x_1)).$$

Passo 1. Mudar o nome das variáveis ligadas, de forma a que nenhuma variável ligada apareça no âmbito de dois quantificadores distintos, e que nenhuma variável livre apareça com o mesmo nome de uma variável ligada. No nosso exemplo, fica

$$\forall x_1 (p(x_1) \Rightarrow q(x_1)) \Rightarrow (\forall x_2 p(x_2) \Rightarrow \forall x_3 q(x_3)).$$

Passo 2. Eliminar todos os operadores booleanos binários, à exceção de $\neg$, $\wedge$ e $\vee$ (é fácil verificar que, de forma semelhante ao cálculo proposicional, se tem as seguintes relações

$$A \Rightarrow B \equiv \neg A \vee B$$
$$A \Leftrightarrow B \equiv (\neg A \vee B) \wedge (\neg B \vee A)$$

onde A e B são fórmulas de $\mathcal{L}$. De facto, $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$ é uma tautologia, e portanto esta fórmula é logicamente válida. Do Teorema 3.2.22, concluímos que se tem $A \Rightarrow B \equiv \neg A \vee B$. A outra equivalência lógica pode ser obtida do mesmo modo. No nosso exemplo, fica

$$\neg \forall x_1 (\neg p(x_1) \vee q(x_1)) \vee \neg \forall x_2 p(x_2) \vee \forall x_3 q(x_3).$$

Passo 3. "Empurrar" todos os operadores de negação para dentro, colapsando duplas negações ($\neg\neg A \equiv A$) e utilizando as leis de De Morgan (2.1)

$$\neg(A \vee B) \equiv \neg A \wedge \neg B \quad \text{e} \quad \neg(A \wedge B) \equiv \neg A \vee \neg B,$$

(é fácil verificar que estas relações do cálculo proposicional continuam válidas se A e B são fórmulas de $\mathcal{L}$) e as equivalências (ver o exercício 52)

$$\neg\forall x_1\phi(x_1) \equiv \exists x_1\neg\phi(x_1) \quad \text{e} \quad \neg\exists x_1\phi(x_1) \equiv \forall x_1\neg\phi(x_1).$$

No nosso exemplo, fica

$$\exists x_1(p(x_1) \wedge \neg q(x_1)) \vee \exists x_2\neg p(x_2) \vee \forall x_3 q(x_3).$$

Passo 4. Extrair quantificadores. Basta aplicar as seguintes regras de (3.3), que são sempre válidas, já que nenhuma variável aparece repetida em dois quantificadores

$$A \text{ op } QxB(x) \equiv Qx(A \text{ op } B(x)) \quad \text{e} \quad QxA(x) \text{ op } B \equiv Qx(A(x) \text{ op } B)$$

onde Q é um quantificador e op é $\wedge$ ou $\vee$. A extração de quantificadores deve ser feita passo a passo, escolhendo em cada passo um quantificador que não esteja no âmbito de outro quantificador da "matriz" (colocamos matriz entre aspas, porque neste momento ainda contém quantificadores). No nosso exemplo, fica

$$\exists x_1\exists x_2\forall x_3((p(x_1) \wedge \neg q(x_1)) \vee \neg p(x_2) \vee q(x_3)).$$

Este procedimento pode ser aplicado a qualquer fórmula ϕ da linguagem $\mathcal{L}$, obtendo-se como resultado uma fórmula em forma prenexa ϕ' tal que $\phi \equiv \phi'$. □

Formalmente, não precisávamos de eliminar as implicações, porque nada impede na definição de fórmula prenexa que não hajam implicações na fórmula ψ que se segue aos quantificadores. No entanto teríamos, por exemplo, de ter outras identidades que permitissem extrair quantificadores no passo 3 para o caso de implicações, etc. Portanto, para facilitar a nossa tarefa, optamos por apenas trabalhar com os operadores $\neg, \wedge, \vee$. Além do mais, se quiséssemos, poderíamos ainda (por exemplo adicionando um passo 5 onde seriam utilizadas as propriedades distributivas envolvendo o $\wedge$ e o $\vee$) obter fórmulas prenexas em forma normal conjuntiva e forma normal disjuntiva. Estas são fórmulas prenexas em que ψ está em forma normal conjuntiva ou disjuntiva, respetivamente (pode-se definir fórmulas normais conjuntivas e disjuntivas como no caso da lógica proposicional, apenas com a diferença que na lógica de primeira ordem um literal é uma fórmula atómica, ou uma negação de uma fórmula atómica).

3.3 Sistemas dedutivos

Tal como no cálculo proposicional, é possível estudar a lógica de primeira ordem utilizando sistemas dedutivos, em vez de se utilizar uma abordagem semântica. Veremos, mais adiante, que estas abordagens, tal como no caso do cálculo proposicional, são equivalentes.

Nesta disciplina iremos abordar o estudo de sistemas dedutivos para a lógica de primeira ordem através da dedução natural. Outras abordagens alternativas podem ser encontradas na literatura (por exemplo, sistema de Hilbert, sistema de Gentzen, etc.).

	Introdução	Eliminação
$\forall$	$\dfrac{\begin{array}{|c|}\hline y \\ \vdots \\ \phi[y/x] \\ \hline\end{array}}{\forall x\,\phi}\ \forall x_i$	$\dfrac{\forall x\,\phi}{\phi[t/x]}\ \forall x_e$
$\exists$	$\dfrac{\phi[t/x]}{\exists x\,\phi}\ \exists x_i$	$\dfrac{\exists x\,\phi \qquad \begin{array}{|c|}\hline y\ \ \phi[y/x] \\ \vdots \\ \chi \\ \hline\end{array}}{\chi}\ \exists x_e$

Tabela 3.1: Regras de inferência para quantificadores (lógica de primeira ordem).

3.3.1 Dedução natural

As regras da dedução natural para a lógica de primeira ordem funcionam de forma semelhante ao caso da dedução natural para o cálculo proposicional. Em particular, continuamos a utilizar todas as regras de inferência da Tabela 2.1, agora aplicadas a fórmulas da lógica de primeira ordem, assim como as regras derivadas da Tabela 2.2, às quais acrescentamos ainda as regras da Tabela 3.1, para o caso de quantificadores. Para utilizar estas regras, é necessário ter presente a seguinte definição.

Definição 3.3.1. Uma variável x_i diz-se *livre para um termo t* numa fórmula ϕ, se toda a ocorrência livre de x_i nunca ocorre no âmbito de um quantificador cuja variável de quantificação é uma variável que aparece na expressão do termo t.

Exemplo 3.3.2. Para

$$\phi = (\forall x_4)p_1^1(x_1) \Rightarrow p_1^1(f_1^2(x_2, x_3)) \tag{3.4}$$

x_1 é uma variável livre para o termo $f_1^2(x_2, x_3)$ (não ocorre no âmbito de um quantificador $\forall x_2$ ou $\forall x_3$), enquanto que x_1 já não é livre para o mesmo termo quando

$$\phi = (\forall x_2)p_1^1(x_1) \Rightarrow p_1^1(f_1^2(x_2, x_3)) \tag{3.5}$$

A noção de variável livre para um termo pode parecer estranha, mas é útil para casos em que se substitui uma variável numa fórmula por um termo. Por exemplo, na fórmula (3.4), a ocorrência de x_1 é livre e, se substituirmos x_1 pelo termo $t = f_1^2(x_2, x_3)$, todas as variáveis de t continuam livres. Se fizermos isso na fórmula (3.5), a variável x_2 do termo $t = f_1^2(x_2, x_3)$ passa a estar ligada na fórmula ϕ, e isso é indesejável. Se garantirmos que

só efetuamos substituições de uma variável x_i por um termo t quando x_i é livre para t, então garantimos que este tipo de problema nunca ocorrerá.

Definição 3.3.3. Dada uma variável x, um termo t, e uma fórmula ϕ, definimos $\phi[t/x]$ como sendo a fórmula obtida substituindo cada ocorrência livre de x em ϕ por t, **desde que a variável x seja livre para o termo t**.

Exemplo 3.3.4. Na fórmula ϕ dada por (3.4), tem-se que

$$\phi[f_1^2(x_2, x_3)/x_1] = (\forall x_4)p_1^1(f_1^2(x_2, x_3)) \Rightarrow p_1^1(f_1^2(x_2, x_3))$$

Vamos agora introduzir as regras da dedução natural para a lógica de primeira ordem. Estas regras consistem nas regras da Tabela 2.1 conjugadas com as regras da Tabela 3.1, que introduzem novas regras para os quantificadores $\forall$ e $\exists$. De forma semelhante ao caso da Tabela 2.1, haverá regras de introdução e de eliminação dos quantificadores.

Para não confundir a regra $\forall x_i$ (introdução do quantificador $\forall x$) com o quantificador $\forall x_i$ ("qualquer que seja" x_i), não iremos nesta secção utilizar índices para designar variáveis, isto é tomaremos $x, y, z, \ldots$ como nomes de variáveis em vez de $x_1, x_2, x_3, \ldots$.

As regras da Tabela 3.1 aplicam-se de forma semelhante às da Tabela 2.1. Por exemplo, na regra $\forall x_i$ é dito que sempre que introduzirmos uma variável **nova** y, se conseguirmos deduzir sempre $\phi[y/x]$, então a fórmula ϕ terá de ser verdadeira qualquer que seja o x. Ou seja, tem-se $\forall x\, \phi$. Note-se que a variável y é apenas válida dentro da caixa (contexto) em que foi definida. De forma semelhante, se sabemos que se tem $\forall x\, \phi$, então $\phi[t/x]$ também terá de ser válida, qualquer que seja o termo t (desde que a variável x seja livre para o termo t). É isto que diz a regra $\forall x_e$.

Quanto à regra $\exists x_i$, ela diz-nos que se conseguirmos deduzir que $\phi[t/x]$ (i.e. ϕ pode ser deduzida para um certo "argumento" t), então tem-se $\exists x\, \phi$. Finalmente, a regra $\exists x_e$ diz-nos que, se já tivermos deduzido $\exists x\, \phi$ e se, assumindo uma variável **nova** y, independentemente da sua escolha conseguimos deduzir, a partir de $\phi[y/x]$, a expressão χ, **no qual a variável y não aparece**, então podemos concluir χ. Em termos intuitivos isto quer dizer que se para qualquer variável y, de $\phi[y/x]$ se conclui χ. Como há pelo menos um argumento que satisfaz ϕ (é o que nos diz a condição $\exists x\, \phi$), então devemos poder concluir χ. Vamos agora dar alguns exemplos de como aplicar estas regras.

Exemplo 3.3.5. Mostre que $\{\forall x(P(x) \Rightarrow Q(x)), \forall x\, P(x)\} \vdash \forall x\, Q(x)$, onde P, Q denotam predicados de aridade 1.

Resolução. Basta utilizar a seguinte derivação

1.		$\forall x(P(x) \Rightarrow Q(x))$	premissa	
2.		$\forall x\, P(x)$	premissa	
3.	y	$P(y) \Rightarrow Q(y)$	$\forall x_e$	1
4.		$P(y)$	$\forall x_e$	2
5.		$Q(y)$	$\Rightarrow_e$	4, 3
6.		$\forall x\, Q(x)$	$\forall x_i$	3–5

Exemplo 3.3.6. Mostre que $\{P(t), \forall x(P(x) \Rightarrow \neg Q(x))\} \vdash \neg Q(t)$, para qualquer termo t, onde P, Q denotam predicados de aridade 1.

Resolução. Basta utilizar a seguinte derivação

1.	$P(t)$	premissa
2.	$\forall x(P(x) \Rightarrow \neg Q(x))$	premissa
3.	$P(t) \Rightarrow \neg Q(t)$	$\forall x_e$ 2
4.	$\neg Q(t)$	$\Rightarrow_e$ 1, 3

Exemplo 3.3.7. Mostre que $\forall x\, \phi \vdash \exists x\, \phi$, para qualquer fórmula ϕ.

Resolução. Basta utilizar a seguinte derivação

1.	$\forall x\, \phi$	premissa
2.	$\phi[x/x]$	$\forall x_e$ 1
3.	$\exists x\, \phi$	$\exists x_i$ 2

Exemplo 3.3.8. Mostre que $\{\forall x(P(x) \Rightarrow Q(x)), \exists x P(x)\} \vdash \exists x Q(x)$, onde P, Q denotam predicados de aridade 1.

Resolução. Basta utilizar a seguinte derivação

1.		$\forall x(P(x) \Rightarrow Q(x))$	premissa
2.		$\exists x P(x)$	premissa
3.	y	$P(y)$	hipótese
4.		$P(y) \Rightarrow Q(y)$	$\forall x_e$ 1
5.		$Q(y)$	$\Rightarrow_e$ 3, 4
6.		$\exists x Q(x)$	$\exists x_i$ 5
7.		$\exists x Q(x)$	$\exists x_e$ 2, 3–6

Exemplo 3.3.9. Mostre que $\{\forall x(Q(x) \Rightarrow R(x)), \exists x(P(x) \wedge Q(x))\} \vdash \exists x(P(x) \wedge R(x))$, onde P, Q, R denotam predicados de aridade 1.

Resolução. Basta utilizar a seguinte derivação

1.		$\forall x(Q(x) \Rightarrow R(x))$	premissa
2.		$\exists x(P(x) \wedge Q(x))$	premissa
3.	y	$P(y) \wedge Q(y)$	hipótese
4.		$Q(y) \Rightarrow R(y)$	$\forall x_e$ 1
5.		$Q(y)$	$\wedge_{e_2}$ 3
6.		$R(y)$	$\Rightarrow_e$ 5, 4
7.		$P(y)$	$\wedge_{e_1}$ 3
8.		$P(y) \wedge R(y)$	$\wedge_i$ 7,6
9.		$\exists x(P(x) \wedge R(x))$	$\exists x_i$ 8
10.		$\exists x(P(x) \wedge R(x))$	$\exists x_e$ 2, 3–9

Exemplo 3.3.10. Mostre que $\{\forall x \forall y(P(x) \Rightarrow Q(y)), \exists x P(x)\} \vdash \forall y Q(y)$, onde P, Q denotam predicados de aridade 1.

Resolução. Basta utilizar a seguinte derivação

1.			$\forall x \forall y(P(x) \Rightarrow Q(y))$	premissa
2.			$\exists x P(x)$	premissa
3.	z			
4.		w	$P(w)$	hipótese
5.			$\forall y(P(w) \Rightarrow Q(y))$	$\forall x_e$ 1
6.			$P(w) \Rightarrow Q(z)$	$\forall y_e$ 5
7.			$Q(z)$	$\Rightarrow_e$ 4, 6
8.			$Q(z)$	$\exists x_e$ 2, 4–7
9.			$\forall y Q(y)$	$\forall y_i$ 3–8

3.3.2 Igualdade na lógica de primeira ordem

Como vimos na Definição 3.1.4, na lógica de primeira ordem temos símbolos de predicado p_i^n que são utilizados para construir fórmulas. Quando estudamos a lógica de primeira ordem do ponto de vista semântico, interpretamos estes símbolos em *todas as possíveis interpretações*. Por exemplo, no Exemplo 3.2.2, o símbolo p_1^2 é interpretado como a relação "igualdade" na interpretação I, enquanto que é interpretado como sendo a relação "diferente de" na interpretação I'. Isto implica que tenhamos $\models_I p_1^2(x,x)$, mas não $\models p_1^2(x,x)$ já que $\not\models_{I'} p_1^2(x,x)$. Em geral, é importante não limitar as interpretações, pelo que a validade lógica de uma fórmula ϕ ($\models \phi$) deve ocorrer apenas se ela é válida para qualquer interpretação I ($\models_I \phi$).

No entanto, há um caso que é normalmente considerado como sendo uma exceção, que é o da igualdade. A igualdade é uma construção tão básica que é muitas vezes incluída na definição de fórmula de $\mathcal{L}$, embora isso não seja rigorosamente necessário. Isso pode ser feito, por exemplo, assumindo que $\mathcal{L}$ tem um símbolo de predicado p_1^2 e exigindo que só se considere interpretações I onde $[p_1^2]_I$ é interpretado como sendo a igualdade, i.e. onde para todo o $d_1, d_2 \in D_I$ se tenha

$$[p_1^2]_I(d_1,d_2) = \begin{cases} 0 & \text{se } d_1 \neq d_2 \\ 1 & \text{se } d_1 = d_2 \end{cases}$$

Em geral, na lógica de primeira ordem utiliza-se o símbolo "=" para designar o predicado de igualdade numa fórmula, em vez do símbolo p_1^2 que usamos acima. Na notação que temos vindo a usar, se queremos aplicar o predicado "=" aos termos t_1, t_2, em rigor deveríamos escrever $=(t_1,t_2)$. No entanto, utilizando a prática que é comum, iremos escrever $t_1 = t_2$ em vez de $=(t_1,t_2)$. Também iremos utilizar esta convenção para símbolos de predicado bem conhecidos (por exemplo, se $\leq$ designar um símbolo de predicado, iremos escrever $t_1 \leq t_2$ em vez de $\leq(t_1,t_2)$).

Até agora descrevemos a semântica da igualdade na lógica de primeira ordem. É possível estender as regras da dedução natural para incluir também o caso da igualdade. Para este efeito, para além de se considerarem as regras das Tabelas 2.1 e 3.1, deve-se ainda incluir as regras da Tabela 3.2

	Introdução	Eliminação
$=$	$\dfrac{}{t = t}\ {=_i}$	$\dfrac{t_1 = t_2 \quad \phi[t_1/x]}{\phi[t_2/x]}\ {=_e}$

Tabela 3.2: Regras de inferência para a igualdade na lógica de primeira ordem.

A regra $=_i$ diz que todo o termo é igual a ele próprio. A regra $=_e$ diz que, de premissas $t_1 = t_2$ e $\phi[t_1/x]$ (ϕ é válida quando se substitui x por t_1), então podemos inferir $\phi[t_2/x]$. Vamos ver alguns exemplos, onde iremos utilizar também o símbolo de predicado $>$ de aridade 2, e o símbolo de função $+$, também de aridade 2.

Exemplo 3.3.11. Mostre que

$$\{x + 1 = 1 + x, (x + 1 > 1) \Rightarrow (x + 1 > 0)\} \vdash (1 + x > 1) \Rightarrow (1 + x > 0)$$

Resolução. Basta utilizar a seguinte derivação

1.	$x + 1 = 1 + x$	premissa
2.	$(x + 1 > 1) \Rightarrow (x + 1 > 0)$	premissa
3.	$(1 + x > 1) \Rightarrow (1 + x > 0)$	$=_e$ 1,2

Exemplo 3.3.12. Mostre que $\{t_1 = t_2, t_2 = t_3\} \vdash t_1 = t_3$, onde t_1, t_2, t_3 são termos.

Resolução. Basta utilizar a seguinte derivação

1.	$t_1 = t_2$	premissa
2.	$t_2 = t_3$	premissa
3.	$t_1 = t_3$	$=_e$ 1,2

3.4 Correção e completude

Vimos no cálculo proposicional que tínhamos resultados de correção e completude: se $\psi_1, \ldots, \psi_n, \phi$ são fórmulas do cálculo proposicional e $\Gamma = \{\psi_1, \ldots, \psi_n\}$, então $\Gamma \vdash \phi$ se e só se $\Gamma \models \phi$. Será que o mesmo acontece para a lógica de primeira ordem? Por outras palavras, será que a abordagem semântica à lógica de primeira ordem e a abordagem que utiliza sistemas dedutivos são equivalentes? A reposta é afirmativa, de acordo com o seguinte teorema.

Teorema 3.4.1 (correção e completude). *Sejam $\psi_1, \ldots, \psi_n, \phi$ fórmulas de uma linguagem de primeira ordem e $\Gamma = \{\psi_1, \ldots, \psi_n\}$. Tem-se que $\Gamma \vdash \phi$ se e só se $\Gamma \models \phi$.*

A demonstração deste teorema está fora do âmbito desta disciplina, mas o resultado permite-nos ver que a abordagem semântica e a que utiliza sistemas dedutivos se

complementam uma à outra. De facto, dada uma fórmula ϕ, não é possível verificar de forma finita (exceto nalguns casos particulares) se $\models \phi$, pois para isso teríamos de testar se $\models_I \phi$ para toda a interpretação I, e geralmente há uma infinidade de interpretações possíveis. Mas podemos mostrar que $\vdash \phi$ utilizando uma dedução com um número finito de passos, e depois concluir pelo teorema anterior que $\models \phi$.

Por outro lado, é muito difícil mostrar que se tem $\nvdash \phi$ pois é necessário mostrar que não há nenhuma dedução (e geralmente há uma infinidade delas) que nos permita concluir $\vdash \phi$. Mas se utilizarmos a abordagem semântica, basta verificar que $\not\models_I \phi$ para *uma* interpretação I e daí concluir que $\not\models \phi$. Utilizando o teorema anterior, concluímos que $\nvdash \phi$.

Portanto, ambas as abordagens são importantes no estudo da lógica de primeira ordem, sendo uma complementar à outra.

É também possível mostrar que existe também um resultado de correção e completude para a lógica de primeira com igualdade.

Capítulo 4

Introdução à teoria da computabilidade

Em disciplinas como "Matemática Finita" ou "Algoritmos e Estruturas de Dados", foram apresentadas várias técnicas que nos permitem criar algoritmos eficientes para resolver problemas que nem sempre são triviais.

Mas será que é sempre possível resolver um problema através de um algoritmo? E, no caso de tal ser possível, será que é sempre possível utilizar algum tipo de "truque" para criar um algoritmo que o resolva rapidamente?

Para responder a esta questão, temos de estudar os limites da computação e, como iremos ver, as respostas as ambas as questões são, em geral, negativas.

4.1 Palavras e linguagens

Na secção 2.2 introduzimos a noção de linguagem definida sobre um alfabeto, que será muito importante para nós. Neste capítulo e no capítulo seguinte iremos supor que o alfabeto subjacente à linguagem é finito.

Definição 4.1.1. O *comprimento* de uma palavra w sobre Σ é o número de símbolos presentes em w, denotado por $|w|$. Em particular a *palavra vazia*, denotada de ε, é a palavra com zero ocorrências de símbolos.

Por exemplo, 010010 é uma palavra sobre o alfabeto binário $\Sigma_1 = \{0,1\}$, e da mesma forma $abracadabra$ é uma palavra sobre o alfabeto $\Sigma_2 = \{a,b,\ldots,z\}$, com comprimentos 6 e 11, respetivamente. Estritamente falando, a definição de comprimento da palavra w é incorreta, pois estamos a falar no número de *ocorrências* de símbolos de Σ, e não simplesmente no número de símbolos de w (por exemplo, $abracadabra$ só tem 5 símbolos: a,b,c,d,r, mas há 11 ocorrências de símbolos de Σ_2 nesta palavra). Mas, como normalmente ao utilizar a expressão "número de símbolos" subentende-se "número de ocorrências de símbolos", iremos manter esta definição. O conjunto das

palavras de comprimento k sobre Σ é dado por

$$\Sigma^k = \{a_1 \dots a_k : a_i \in \Sigma, \text{para } i = 1, \dots, k\},$$

onde assumimos que $\Sigma^0 = \{\varepsilon\}$. Por exemplo, se tomarmos $\Sigma = \{0, 1\}$, tem-se $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$, $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$, ...

O conjunto das palavras sobre Σ, que recordamos ser designado por Σ^*, é o conjunto definido por:

$$\Sigma^* = \bigcup_{k \in \mathbb{N}_0} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Às vezes também vamos utilizar um expoente num símbolo ou palavra. Em geral a notação w^k, para $k \in \mathbb{N}$ e $w \in \Sigma^*$, designa a palavra

$$w^k = \underbrace{ww \dots w}_{k \text{ vezes}}.$$

Por exemplo, $0^3 = 000$, $(10)^2 = 1010$, $(101)^0 = \varepsilon$.

Definição 4.1.2. Dadas duas palavras $x = x_1 \dots x_k$, $y = y_1 \dots y_m$ sobre Σ, definimos a sua *concatenação* xy como sendo a palavra

$$x_1 \dots x_k y_1 \dots y_m.$$

Nos exemplos que se seguem $\Sigma = \{0, 1\}$. Por exemplo, dadas palavras $u = 1011$ e $v = 11$, definidas sobre $\Sigma = \{0, 1\}$, tem-se $uv = 101111$ e $vu = 111011$ (note-se que a ordem pela qual se faz a concatenação importa!).

Definição 4.1.3. Dada uma palavra $x = x_1 \dots x_k$, a sua *palavra reversa* é x escrita ao contrário, i.e. é a palavra $x^{\mathcal{R}} = x_k \dots x_1$. Se $x = x^{\mathcal{R}}$, x diz-se um *palíndromo*.

Por exemplo, $u^{\mathcal{R}} = 1101$ e $v^{\mathcal{R}} = 11$. Além do mais, como $v = 11 = v^{\mathcal{R}}$, concluímos que v é um palíndromo.

Antes de introduzir a próxima definição, recordamos que uma linguagem sobre um alfabeto Σ é um subconjunto L de Σ^*. Por exemplo, $L = \{1, 11, 111\}$ é uma linguagem sobre $\Sigma = \{0, 1\}$, assim como o conjunto vazio $\varnothing$, ou o conjunto $\{w \in \Sigma^* | w$ tem o mesmo número de ocorrências de 0's do que 1's$\}$. No entanto, $L = \{a, b, c\}$ já não é uma linguagem sobre Σ.

Definição 4.1.4. Sejam A e B linguagens sobre o alfabeto Σ. Definimos as seguintes operações cujo resultado são também linguagens sobre Σ.

- **União**: $A \cup B = \{x \in \Sigma^* : x \in A \text{ ou } x \in B\}$.
- **Concatenação**: $A \circ B = \{xy \in \Sigma^* : x \in A \text{ e } y \in B\}$.

- **Operador de fecho**: $A^* = \{x_1 x_2 \dots x_k \in \Sigma^* : k \in \mathbb{N}_0 \text{ e } x_i \in A\}$.

Dito de outro modo, A^* é o conjunto de todas as palavras que podem ser obtidas concatenando palavras de A tantas vezes quantas quisermos. Definimos também

$$A^k = \underbrace{A \circ \dots \circ A}_{k \text{ vezes}} = \{x_1 x_2 \dots x_k \in \Sigma^* : x_i \in A\}$$

para $k \in \mathbb{N}_0$, onde $A^0 = \{\varepsilon\}$. Em particular

$$A^* = \bigcup_{k \in \mathbb{N}_0} A^k.$$

Por exemplo, tomando $A = \{10, 11\}$, $B = \{0, 111\}$, tem-se

$$\begin{aligned} A \cup B &= \{0, 10, 11, 111\} \\ A \circ B &= \{100, 10111, 110, 11111\} \\ A^* &= \{\varepsilon, 10, 11, 1010, 1011, 1110, 1111, 101010, 101011, \dots\} \\ A^0 &= \{\varepsilon\} \\ A^1 &= \{10, 11\} \\ A^2 &= \{1010, 1011, 1110, 1111\} \end{aligned}$$

(concatenando zero palavras de A, obtém-se só a palavra vazia. Concatenando uma palavra de A, obtém-se as palavras 10 e 11. Concatenando duas palavras de A, obtém-se $1010, 1011, 1110, 1111$. Concatenando três palavras de A, obtém-se $\dots$).

Se estiver definida uma ordem sobre Σ, podemos definir a *ordem lexicográfica* em Σ^* como sendo a mesma ordem que é utilizada num dicionário, exceto que palavras mais curtas precedem palavras mais longas. Por exemplo, a ordem lexicográfica sobre $\{0, 1\}$ é dada por (assumindo que já temos a ordem definida por $0 < 1$ sobre este alfabeto)

$$\varepsilon <_{\Sigma^*} 0 <_{\Sigma^*} 1 <_{\Sigma^*} 00 <_{\Sigma^*} 01 <_{\Sigma^*} 10 <_{\Sigma^*} 11 <_{\Sigma^*} 000 <_{\Sigma^*} \dots$$

Formalmente diz-se que $u <_{\Sigma^*} v$, para $u, v \in \Sigma^*$ se $|u| < |v|$, ou se $|u| = |v|$ então existe k tal que $u_i = v_i$ para $i = 1, \dots, k-1$ e $u_k < v_k$. Normalmente iremos escrever $u < v$ em vez de $u <_{\Sigma^*} v$, para simplificar a notação.

4.2 Máquinas de Turing

As máquinas de Turing são um modelo computacional introduzido em 1936 por Alan Turing. Apesar da sua simplicidade, este modelo permite simular qualquer programa de computador, permitindo-nos obter resultados que nos ajudam a esclarecer que problemas podem ou não ser resolvidos através de programas de computador.

A ideia básica subjacente a uma máquina de Turing é a seguinte. Qualquer algoritmo ou programa de computador pode ser sempre executado "à mao" por um ser humano munido de lápis, papel, e borracha, embora obviamente de forma muito mais lenta do que se o programa fosse executado num computador. No essencial, o que a máquina de Turing faz é simular o processo de execução de um algoritmo por um ser humano.

Esta analogia não aparece por acaso. No momento em que este modelo foi idealizado por Alan Turing, não haviam computadores digitais. Todos os cálculos necessários para realizar diversas tarefas importantes (contabilidade, obter estatísticas de censos, etc.) eram realizados "à mão", eventualmente com a ajuda de alguns auxiliares de cálculo que não eram verdadeiros computadores por não serem programáveis (réguas de cálculo, etc.) e, em casos mais raros, com a ajuda de computadores analógicos. Posteriormente, durante a Segunda Guerra Mundial, Alan Turing viria a ter um papel crucial no desenvolvimento de diversas máquinas electromecânicas utilizadas com grande sucesso para decifrar as mensagens encriptadas pelas forças armadas alemãs.

No caso da máquina de Turing, o papel a que um humano tem acesso é modelado através de uma fita de comprimento infinito (para modelar o facto de podermos utilizar, em princípio, tanto papel quanto o que quisermos), onde se podem registar símbolos (por exemplo "0", "1", ou "+"). Em cada momento o ser humano apenas consegue focar a sua atenção numa zona restrita do papel. Isso é modelado através de uma cabeça de leitura, que consegue apenas ler uma célula da fita de cada vez. O ser humano também pode mudar a sua atenção para outra zona do papel, pelo que se permite que a cabeça de leitura se desloque para outras células adjacentes, uma célula de cada vez. Finalmente, um ser humano a executar um algoritmo poderá estar num determinado estado mental (por exemplo, ao somar dois números, vamos somando dígitos da direita para a esquerda e, num determinado momento do processo, podemos estar no estado mental "e vai mais um"), o que é modelo através de estados da máquina de Turing.

Vamos apresentar a definição formal de máquina de Turing e depois ver como ela funciona em detalhe.

Definição 4.2.1. Uma *máquina de Turing* é um 7-tuplo $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ onde:

1. Q é um conjunto finito (de *estados*),
2. Σ é um alfabeto (dos *inputs*),
3. Γ é um alfabeto (da fita), contendo o símbolo especial B (o símbolo *branco*), e satisfazendo $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{D, E, P\}$ é a *função de transição*,
5. $q_0 \in Q$ é o *estado inicial*,
6. $F \subseteq Q$ é o conjunto dos *estados finais*.

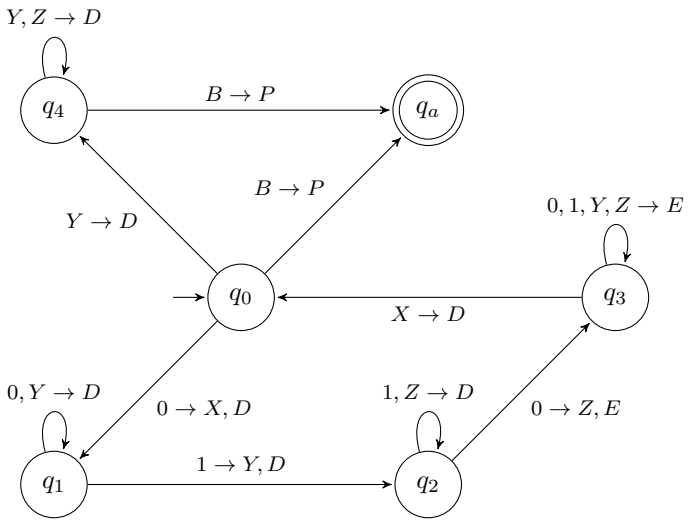


Figura 4.1: Um exemplo de máquina de Turing.

No ponto 4 da definição anterior D, E, P designam movimentos da cabeça de leitura: uma célula para a direita (D), uma célula para a esquerda (E), ou então fica parada (P).

Apesar de podermos utilizar esta definição, na prática é mais conveniente representar máquinas de Turing de forma gráfica, como exemplificado na Fig. 4.1, onde cada círculo representa um estado e as setas representam transições entre estados.

Inicialmente a máquina de Turing recebe como *input* uma palavra $w = a_1 \dots a_n \in \Sigma^*$. Os símbolos $a_1, \dots, a_n$ estão escritos na fita de forma consecutiva em células adjacentes, e supomos que as restantes células contêm o símbolo *branco* B. A cabeça de leitura é colocada sobre a célula que contém a_1 (i.e. sobre o símbolo mais à esquerda do *input*), e começa-se a computação no *estado inicial* q_0, que é representado graficamente como sendo o estado para o qual aponta a única seta sem origem noutro estado. A partir deste momento podemos começar a computação, que vai sendo executada passo-a-passo até atingir um estado *final* (caso em que o *input* w é aceite), representados graficamente com círculos duplos, ou até que não haja nenhuma transição definida (caso em que o *input* w é rejeitado). Pode acontecer que um *input* não seja aceite nem rejeitado, i.e. a computação pode continuar indefinidamente.

Em cada momento a cabeça de leitura da máquina de Turing lê o símbolo da célula onde está posicionada. Dependendo *apenas desse símbolo e do estado atual*, a máquina faz os seguintes três passos simultaneamente: (i) atualiza o estado atual; (ii) altera (ou não) o conteúdo da célula que está a ler; (iii) mantêm a cabeça de leitura no mesmo sítio ou, em alternativa, move-a uma célula para a direita, ou uma célula para a esquerda.

Por exemplo, no caso da Fig. 4.1, a transição que vai de q_0 a q_1 deve-se ler como: "se no estado q_0 e com a cabeça de leitura a ler o símbolo 0, então substituir esse 0 por um X, mover a cabeça de leitura uma célula para a direita (D), e mudar o estado para q_1". Para simplificar a notação, se o símbolo atualmente lido não é alterado, então apenas é

indicado o movimento da cabeça de leitura a realizar. Por exemplo, a transição de q_0 para q_4 deve ser interpretada como "se no estado q_0 e com a cabeça de leitura a ler o símbolo Y, então manter esse Y, mover a cabeça de leitura uma célula para a direita (D), e mudar o estado para q_4". Se num determinado estado fizermos ações semelhantes para símbolos diferentes, podemos agrupar essa informação numa única regra. Por exemplo, a transição de q_3 para ele mesmo deve ser interpretada como "se no estado q_3 e com a cabeça de leitura a ler o símbolo 0 ou 1 ou Y ou Z, então manter esse símbolo, mover a cabeça de leitura uma célula para a esqueda (E), e manter o mesmo estado". O P na transição do estado q_0 para q_a quer dizer que a cabeça de leitura irá ficar parada na mesma célula aquando desta transição.

O diagrama da Fig. 4.2 mostra os primeiros 5 passos da computação da MT da Fig. 4.1 com o *input* 001100. O que faz esta MT? Ela aceita todas as palavras que pertençam à linguagem

$$L = \{0^k 1^k 0^k \in \{0,1\}^* | k \in \mathbb{N}_0\},$$

Como funciona a MT? Ela aceita imediatamente a palavra vazia ε. Se o *input* começa por um 1, ele é imediatamente rejeitado. Se começa com um 0, esse 0 é alterado para um X (é "marcado") e procuramos o 1 seguinte à direita. Se não existir, a palavra é rejeitada. Se houver um 1, ele é alterado para Y. Agora procuramos um zero à direita desse 1. Se não existir, o *input* é rejeitado. Se existir, alteramos o 0 para Z. Agora voltamos para o início da palavra (mais corretamente até ao primeiro X já marcado — não interessa voltar mais para trás do que isto) — e repetimos este processo até esgotar os zeros do primeiro bloco de 0's (é quando transitamos de q_0 para q_4). Aí vamos percorrendo a palavra, garantindo que já não há 0's e 1's, isto é que só há Y's e Z's, até chegar ao símbolo branco. Neste momento sabemos que podemos aceitar o *input*. Se um *input* não deve ser aceite, então há sempre um ponto onde já não há é possível efetuar mais transições com este *input*, pelo que ele é rejeitado.

Note-se que esta MT para[1] sempre, aceitando ou rejeitando o *input* (diz-se que a MT *decide* a linguagem L), mas pode acontecer que para certos *inputs* uma MT não pare. Por exemplo, a MT da Fig. 4.3 não para para o *input* $w = 0$ (a cabeça de leitura desloca-se indefinidamente para a direita).

Em cada momento da computação apenas um número finito de células não terá o símbolo B. Portanto, o conteúdo da fita a partir da primeira célula mais à esquerda que não contém um B até à célula imediatamente à esquerda da célula atualmente lida pela cabeça de leitura pode ser escrito como uma palavra $v \in \Sigma^*$. Da mesma forma, o conteúdo da fita começando a partir da célula lida pela cabeça de leitura, inclusive, até à célula mais à direita que não contém o símbolo B, pode ser codificado noutra palavra $w \in \Sigma^*$. Então se q_i for o estado atual da MT, o triplo (v, q_i, w) determina completamente o estado atual da computação: conhecendo apenas esta informação podemos continuar

[1]Segundo o novo acordo ortográfico, a conjugação "pára" do verbo parar passa a escrever-se "para". Embora nos pareça que a nova grafia é menos clara, pois pode-se confundir com a preposição "para", iremos utilizar a nova grafia, uma vez que nestes apontamentos estamos a utilizar o novo acordo ortográfico.

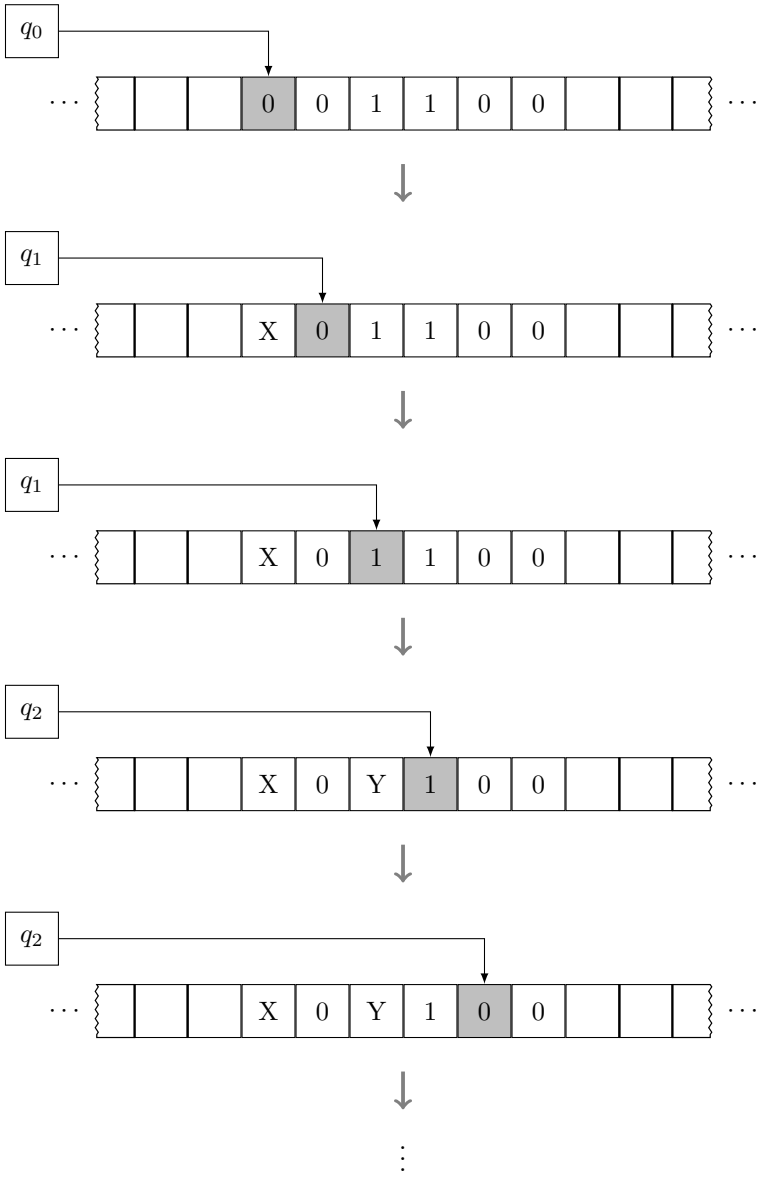


Figura 4.2: Os primeiros 5 passos da computação da máquina de Turing da Fig. 4.1 com *input* 001100. Os símbolos *B* não são exibidos.

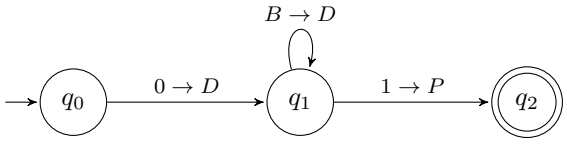


Figura 4.3: Uma máquina de Turing cuja computação não para para o *input* $w = 0$.

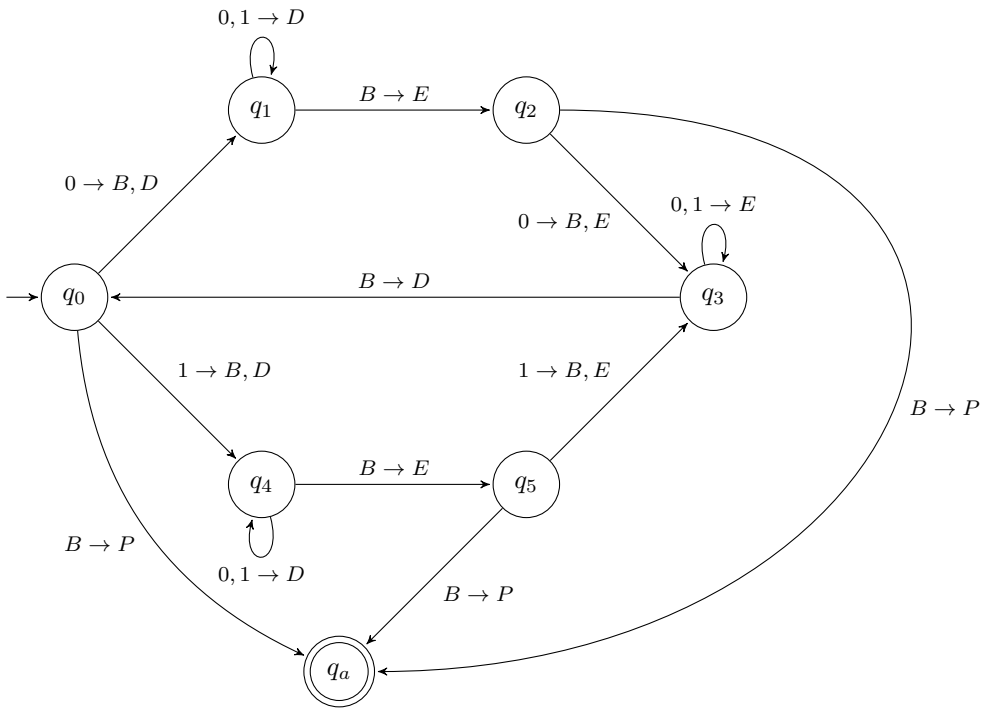


Figura 4.4: Uma máquina de Turing que decide a linguagem formada por todos os palíndromos em $\{0, 1\}^*$.

a computação sem quaisquer problemas. A estes triplos dá-se o nome de *configurações*. Por exemplo, a configuração correspondente ao 3º passo da computação indicada na Fig. 4.2 é dada por $(X0, q_1, 1100)$, enquanto que a configuração correspondente ao 1º passo da mesma computação é dada por $(\varepsilon, q_0, 001100)$.

Repare-se que a uma dada configuração segue-se outra configuração e por aí fora, bastando para isso ir seguindo as regras de transição da máquina de Turing, até chegar a uma configuração (u, q, w) em que q é um estado final, a que se chama uma *configuração aceitadora*, ou até chegar a uma configuração a que não se lhe segue nenhuma outra configuração (a computação termina neste momento, sem aceitar), chamada *configuração rejeitadora*. Uma *configuração de paragem* é uma configuração que é ou aceitadora, ou rejeitadora.

Em particular, no início da computação, a configuração será do tipo (ε, q_0, w), onde w é o *input* da MT e q_0 é o estado inicial. Este tipo de configuração é chamado de *configuração inicial*.

Definição 4.2.2. Uma máquina de Turing *aceita* a palavra $w \in \Sigma^*$ se existe uma sequência de configurações $C_1, \ldots, C_k$ tal que:

1. C_1 é a configuração inicial de M para a palavra w,
2. A regra de transição de M determina que a cada configuração C_i se siga a configuração C_{i+1},
3. C_k é uma configuração aceitadora.

A *linguagem* de uma máquina de Turing M (ou *linguagem aceite* por M, ou ainda *linguagem reconhecida* por M) é a classe

$$L(M) = \{w \in \Sigma^* | M \text{ aceita } w\}.$$

Definição 4.2.3. Uma linguagem diz-se *recursivamente enumerável* (r.e.) se é a linguagem de alguma máquina de Turing.

Por exemplo, as linguagens das MT das Fig. 4.1, 4.4, e 4.3 são, respetivamente, $\{0^k1^k0^k \in \{0,1\}^* : k \in \mathbb{N}\}$, $\{w \in \{0,1\}^* : w = w^{\mathcal{R}}\}$, e $\{w \in \{0,1\}^* : w \text{ começa com } 01\} = \{01v \in \{0,1\}^* : v \in \{0,1\}^*\}$, pelo que estas linguagens são recursivamente enumeráveis. A designação "recursivamente enumerável" pode parecer algo obscura, mas deve-se essencialmente a razões históricas.

Repare-se que há uma diferença entre as MT das Fig. 4.1, 4.4, e 4.3: na primeira e segunda MT, dado um *input* w, a MT para sempre em tempo finito e diz-nos (em tempo finito) se uma palavra é aceite ou não (i.e. se pertence ou não à linguagem $\{0^k1^k0^k \in \{0,1\}^* | k \in \mathbb{N}\}$ no caso da primeira MT, ou se pertence ou não à linguagem $\{w \in \{0,1\}^* : w = w^{\mathcal{R}}\}$ no caso da segunda MT), enquanto que na terceira MT há *inputs* (neste caso, 0) para os quais a MT nunca chega a parar, e portanto nunca nos

	O que faz a MT M com o *input* w?	
	Se M aceita L	Se M decide L
$w \in L$	M com *input* w acaba por parar e aceitar w	M com *input* w acaba por parar e aceitar w
$w \notin L$	M com *input* w: • Acaba por parar e rejeitar w ou • Nunca para	M com *input* w acaba por parar e rejeitar w

Tabela 4.1: Quadro que resume as diferenças entre *aceitar* e *decidir* uma linguagem L.

chega a dar uma resposta em tempo finito à questão de saber se a palavra é aceite ou não. Como normalmente é desejável que a MT pare ao fim de algum tempo para nos dar uma resposta, seja ela positiva ou negativa, introduzimos a seguinte definição.

Definição 4.2.4. Uma linguagem L diz-se *recursiva* se é a linguagem de alguma máquina de Turing M, com a propriedade adicional que, para qualquer $w \in \Sigma^*$, a computação de M com *input* w tem sempre de acabar numa configuração de paragem. Neste caso dizemos também que M *decide* L.

Por exemplo, a MT da Fig. 4.1 decide a linguagem $\{0^k 1^k 0^k \in \{0,1\}^* | k \in \mathbb{N}\}$ mas, apesar de $\{w \in \{0,1\}^* : w \text{ começa com } 01\}$ ser a linguagem da MT da Fig. 4.3, essa MT não decide $\{w \in \{0,1\}^* : w \text{ começa com } 01\}$, porque há *inputs* (por exemplo, o *input* 0) para os quais a computação nunca chega a uma configuração de paragem (por outras palavras, utilizando tempo finito e apenas a computação da MT, não conseguimos dizer se o *input* pertence ou não à linguagem).

Note-se que toda a linguagem recursiva é obviamente r.e., mas como teremos ocasião de ver mais à frente, o resultado recíproco não é verdadeiro.

Repare-se que, até agora, apenas consideramos máquinas de Turing com *outputs* do tipo sim/não (i.e., um *input* ou é aceite, ou é rejeitado, desde que a computação da MT acabe). Às vezes temos necessidade de considerar *outputs* mais elaborados, que também podem ser palavras sobre o alfabeto Σ. Ou seja, pretendemos computar uma função $f : \Sigma^* \to \Sigma^*$ com uma MT. Por outras palavras, dada uma palavra inicial $w \in \Sigma^*$, queremos calcular a partir desse *input* uma nova palavra $f(w) \in \Sigma^*$. Por exemplo, tomemos $f : \{1\}^* \to \{1\}^*$ como sendo uma função que, dada uma palavra $1^k \in \{1\}^*$, retorna $f(1^k) = 1^{2k}$, i.e. f duplica o tamanho do *input*. Esta função é computada pela MT da Fig. 4.5.

Dizemos que uma MT computa uma função $f : \Sigma^* \to \Sigma^*$ se, dado um *input* w, inicialmente a cabeça de leitura está a ler o símbolo mais à esquerda do *input* w. Para além do *input*, a fita só tem brancos. Depois a computação é iniciada, e continua até chegar a uma configuração aceitadora. Neste instante a fita conterá somente o resultado $f(w)$ (o resto da fita só terá símbolos brancos), estando a cabeça de leitura a ler o símbolo mais à esquerda de $f(w)$. Nesse caso diz-se que a função f é computável.

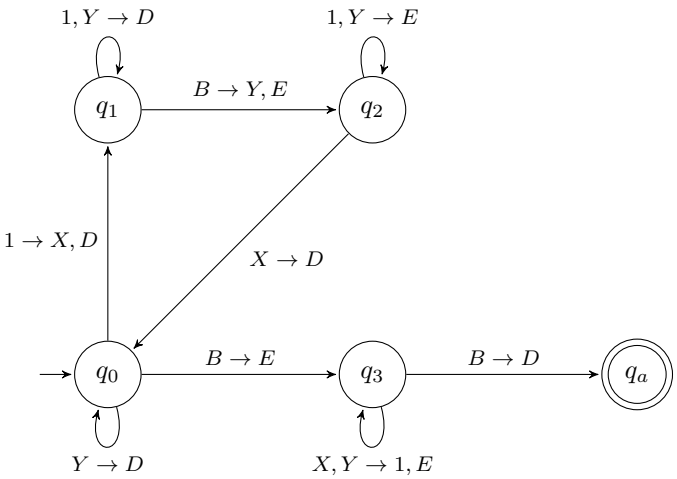


Figura 4.5: Máquina de Turing que duplica o tamanho do *input*, quando este é constituído só por 1's.

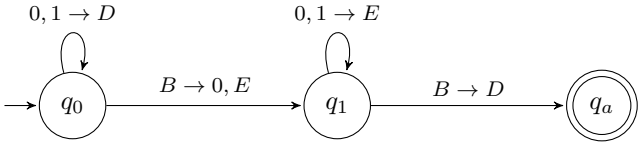


Figura 4.6: Máquina de Turing que multiplica por dois um número inteiro, quando o *input* e o resultado estão escritos em binário.

Note-se que a partir do momento em que conseguimos computar funções sobre Σ^*, podemos computar funções sobre outros domínios, utilizando representações adequadas. Por exemplo, um número inteiro não negativo pode ser representado como uma expressão binária sobre o alfabeto $\Sigma = \{0, 1\}$. Nesse caso, a MT de Turing da Fig. 4.6 permite-nos computar a função $f : \mathbb{N} \to \mathbb{N}$ definida por $f(n) = 2n$, quando o *input* está escrito em binário (o resultado também virá escrito em binário). Por exemplo, se quisermos calcular o dobro de 5, então escrevemos 101 como *input* (=5 em binário), e obtemos como resultado 1010 (=10 em binário).

Às vezes, por razões de simplicidade, iremos também utilizar a *representação unária* de números, que é definida da seguinte forma. Consideremos um alfabeto com um símbolo, normalmente $\Sigma = \{1\}$. Então podemos representar o número k por 1^k. Por exemplo, neste último caso, podemos representar o número 3 pela palavra $111 \in \Sigma^*$. Utilizando esta última representação dos números naturais, concluímos que a MT da Fig. 4.5 computa a função $g : \mathbb{N} \to \mathbb{N}$ definida por $g(k) = 2k$.

Também é possível calcular funções com vários argumentos: basta codificá-los todos na mesma palavra, utilizando um símbolo especial para separar diferentes argumentos. Por exemplo, se quisermos calcular a função soma $S : \mathbb{N}^2 \to \mathbb{N}$ dada por $S(x, y) = x + y$, poderíamos representar cada *input* em binário, separando-os pelo símbolo $\#$, e poderíamos retornar o resultado final em formato binário. Assim, se quiséssemos calcular $S(3, 5)$ uma MT que calculasse a função S com esta representação, com o *input* $11\#101$, deveria retornar o valor 1000 (pois $3 + 5 = 8$).

Note-se que a possibilidade de codificar objetos em binário, etc. não se encontra limita a números. Isso é semelhante ao que acontece nos computadores digitais, onde toda a informação é guardada na memória ou no disco rígido em binário, mas esta informação pode representar diversos objetos, como documentos Word, Excel, vídeos, músicas, fotos, etc.

Por exemplo, consideremos o grafo (não-orientado) G da Fig. 4.7 e o alfabeto $\Sigma = \{0, 1\}$. Então o grafo pode ser codificado na seguinte palavra de Σ^* :

$$\langle G \rangle = \underbrace{0000}_{\text{nº vértices}}\ 11\ \underbrace{0100}_{\text{1ª aresta}}\ \underbrace{1101000011}_{\text{2ª aresta}}\ \underbrace{001000}_{\text{3ª aresta}}. \tag{4.1}$$

O primeiro bloco de zeros dá-nos o número de vértices do grafo (ou seja, 4). Depois escrevemos 11 para dizer que vem uma aresta. Como de seguida vem um 0, separado por um 1 de dois 0's, isso quer dizer que o vértice 1 está ligado ao vértice 2 por uma aresta. De seguida vêm mais dois 1's para assinalar outra aresta: a que vai do vértice 1 ao vértice 4. Finalmente vem a aresta que vai do vértice 2 ao vértice 3. Da mesma forma podemos codificar qualquer grafo G numa palavra de $\{0, 1\}^*$. Note-se (será importante para a próxima secção), que se G tem k vértices, então a descrição de cada aresta utiliza, no máximo, $2k + 3$ símbolos (dois 1's para marcar o início da aresta, $i \leq k$ zeros para marcar o primeiro vértice da aresta, mais um 1 que separa os vértices na aresta, e $j \leq k$ zeros para marcar o segundo vértice). Como não há mais do que k^2 arestas no grafo G, a descrição do grafo necessitará de, no máximo, $k^2(2k + 3) + k$ símbolos de $\{0, 1\}$, i.e.

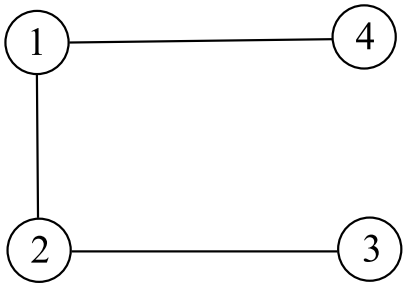


Figura 4.7: Exemplo de grafo.

a descrição do grafo tem um tamanho que é polinomial no número de vértices.

Este procedimento não está limitado a grafos. Por exemplo podemos também codificar em binário uma MT. Para mostrar como é que isso pode ser efetuado, tomemos o caso concreto da MT M da Fig. 4.3. A máquina tem 3 estados e utiliza três símbolos. Então podemos codificá-la de forma semelhante ao que fizemos para o grafo, na seguinte palavra binária:

$$\underbrace{000}_{\text{nº estados}}\ 11\ \underbrace{000}_{\text{nº símbolos}}\ 11\overbrace{\underbrace{0100}_{\text{lido}}\underbrace{1001001011}_{\text{resultado}}}^{\text{transição}}\overbrace{\underbrace{0010}_{\text{lido}}\underbrace{1001010111}_{\text{resultado}}}^{\text{transição}}\overbrace{\underbrace{0010001}_{\text{lido}}\underbrace{0001000111000}_{\text{resultado}}}^{\text{transição}} \tag{4.2}$$

onde

$$\overbrace{\underbrace{00}_{\text{estado}}\ 1\ \underbrace{000}_{\text{símbolo}}}^{\text{lido}} \quad \text{e} \quad \overbrace{\underbrace{000}_{\text{estado}}\ 1\ \underbrace{000}_{\text{símbolo}}\ 1\ \underbrace{000}_{\text{movimento}}}^{\text{resultado}}$$

em que

estados	símbolos	movimentos
$0 \to q_0$	$0 \to B$	$0 \to D$
$00 \to q_1$	$00 \to 0$	$00 \to E$
$000 \to q_2$	$000 \to 1$	$000 \to P$

Normalmente não existe uma única codificação possível. Por exemplo, outra forma de descrever o grafo G da Fig. 4.7 no alfabeto $\Sigma = \{0, 1\}$ seria representá-lo numa forma mais matemática, por exemplo $G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 4\}, \{2, 3\}\})$, e depois codificar cada símbolo de G em binário para escrever G em binário segundo essa codificação. Por exemplo, precisamos dos símbolos $0, 1, 2, \ldots, 9$ para codificar os vértices de um grafo, e ainda dos caracteres "(", ")", "{", "}", e ",". Necessitamos de codificar 15 símbolos em

binário, e para isso precisamos de 4 bits por símbolo. Portanto, utilizando as associações

$$\begin{aligned} 0 &\to 0000 \\ 1 &\to 0001 \\ 2 &\to 0010 \\ &\vdots \\ 9 &\to 1001 \\ (&\to 1010 \\ ) &\to 1011 \\ \{ &\to 1100 \\ \} &\to 1101 \\ , &\to 1110 \end{aligned}$$

a codificação do grafo $G = (\{1,2,3,4\}, \{\{1,2\},\{1,4\},\{2,3\}\})$ neste esquema seria

$$\langle G \rangle = \underbrace{1010}_{(}\ \underbrace{1100}_{\{}\ \underbrace{0001}_{1}\ \underbrace{1110}_{,}\ \underbrace{0010}_{2} \quad \ldots \quad \underbrace{0011}_{3}\ \underbrace{1101}_{\}}\ \underbrace{1101}_{\}}\ \underbrace{1011}_{)}.$$

Regra geral, fixamos uma codificação em binário, que para o caso de um grafo G assumimos ser a codificação exemplificada em (4.1), e que para o caso de uma MT M assumimos ser a codificação exemplificada em (4.2), sendo ambas as codificações representadas por $\langle G \rangle$ e $\langle M \rangle$, respetivamente. De forma semelhante, podemos codificar uma MT M e o seu *input* w numa palavra binária $\langle M, w \rangle$.

4.3 Variantes de máquinas de Turing

Na literatura é possível encontrar outras variantes de máquinas de Turing. Por exemplo, é possível encontrar máquinas de Turing:

- Onde a cabeça de leitura só pode ir uma célula para a esquerda e para a direita, i.e. nunca pode estar parada;
- Com uma fita que só é infinita numa direção, tendo uma célula inicial (se a fita é infinita para a direita, então na célula inicial a cabeça de leitura não se pode mover para a esquerda);
- Com várias fitas;
- Onde é permitido que a função de transição seja não-determinística, i.e. que possa haver mais do que uma transição para um dado estado e símbolo.

Pode-se mostrar que, numa perspetiva de computabilidade, todos estes modelos são equivalentes. Iremos considerar nesta secção os últimos dois tipos de MT, que nos serão úteis no que se segue.

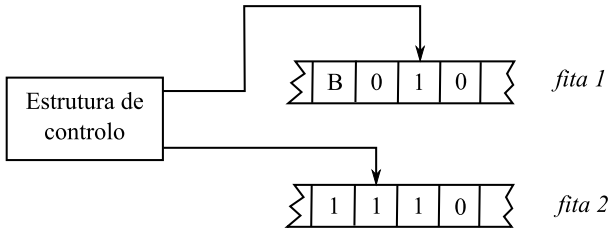


Figura 4.8: Esquema de uma máquina de Turing com duas fitas.

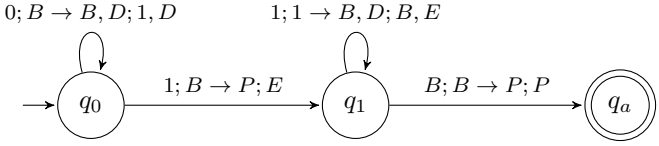


Figura 4.9: Uma máquina de Turing com duas fitas.

4.3.1 Máquinas de Turing com k fitas

Às vezes, por ser mais prático, vamos utilizar MT's com 2 ou mais fitas. O esquema geral de uma MT com duas fitas é dado na Fig. 4.8, e um exemplo concreto é dado na Fig. 4.9. O que se passa em cada fita é separado pelo símbolo ';'. Por exemplo, na Fig. 4.9, a transição do estado q_0 para ele próprio deve-se ler: "se no estado q_0, se a cabeça de leitura da 1ª fita está a ler um 0, e se a cabeça de leitura da 2ª fita está a ler um B, então: manter-se no estado q_0, substituir o 0 por um B na 1ª fita, substituir o B por um 1 na 2ª fita, e mover ambas as cabeças de leitura da 1ª e 2ª fita uma célula para a direita". Utilizamos também as simplificações já descritas para máquinas de Turing com uma fita (por exemplo, a transição entre o estado q_0 e o estado q_1 deve ler-se como "se no estado q_0, se a cabeça de leitura da 1ª fita está a ler um 1, e se a cabeça de leitura da 2ª fita está a ler um B, então: ir para o estado q_0, manter o 1 na 1ª fita, manter o B na 2ª fita, manter a posição da cabeça de leitura da 1ª fita e mover a cabeça de leitura da 2ª fita uma célula para a esquerda"). Esta MT decide a linguagem $\{0^k 1^k \in \{0,1\}^* | k \geq 1\}$ da seguinte forma: por cada 0 do *input* na 1ª fita, escreve um 1 na 2ª fita. Assim vão haver tantos 1's na 2ª fita como 0's no início do *input*. Quando finalmente aparecer 1's na 1ª fita, vamos ver se existem tantos 1's quantos os da segunda fita, rejeitando o *input* se isso não acontecer, ou se aparecerem 0's pelo meio.

Também é possível computar funções com uma MT com várias fitas. Para isso assumimos que inicialmente a primeira fita só contém o *input* w, estando a cabeça de leitura a ler o símbolo mais à esquerda, enquanto que as restantes fitas só têm brancos. Quando a computação termina num estado aceitador, a primeira fita só contém o *output* $f(w)$, estando a cabeça de leitura a ler o símbolo mais à esquerda, enquanto que as restantes fitas só têm brancos. Como exemplo, a MT da Fig. 4.10 computa a função

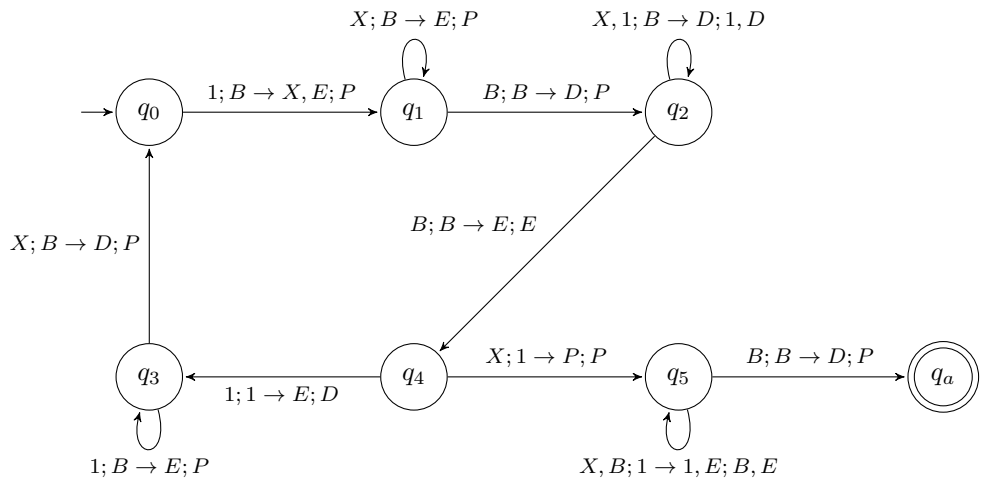


Figura 4.10: Uma máquina de Turing com duas fitas que computa a função $f : \mathbb{N} \to \mathbb{N}$ definida por $f(n) = n^2$.

$f : \mathbb{N} \to \mathbb{N}$ definida por $f(n) = n^2$.

No que se segue, ao mencionarmos que estamos a utilizar uma MT, iremos implicitamente admitir a possibilidade de se usar mais do que uma fita, se isso nos for conveniente.

4.3.2 Máquinas de Turing não-determinísticas

Iremos agora analisar outra variante da MT que muitas vezes é útil. Nas máquinas de Turing que temos vindo a estudar até agora, cada transição é determinística, isto é, a cada configuração segue-se apenas, no máximo, uma só configuração.

Nesta outra variante de máquina de Turing (máquinas de Turing não-determinísticas), iremos considerar não-determinismo: a cada configuração podem-lhe suceder várias configurações, estabelecendo-se assim vários possíveis caminhos de computação para um *input*. Do ponto de vista teórico é interessante considerar MT's não-determinísticas, já que a partir destas máquinas podemos introduzir conceitos que são úteis em aplicações envolvendo criptografia, otimização, etc.

Um exemplo de uma MT não-determinística (MTND) é dado na Fig. 4.11 (note-se que no estado q_0 há 2 transições possíveis quando se está a ler o símbolo 0 com a cabeça de leitura). Por exemplo, quando esta MTND é iniciada com o *input* 000, há dois ramos de computação possíveis, correspondentes às seguintes configurações:

Ramo 1: $(\varepsilon, q_0, 000) \to (0, q_1, 00) \to (00, q_2, 0) \to (000, q_1, B)$ [rejeita]
Ramo 2: $(\varepsilon, q_0, 000) \to (\varepsilon, q_3, 00) \to (0, q_4, 0) \to (00, q_5, B) \to (00, q_a, B)$ [aceita]

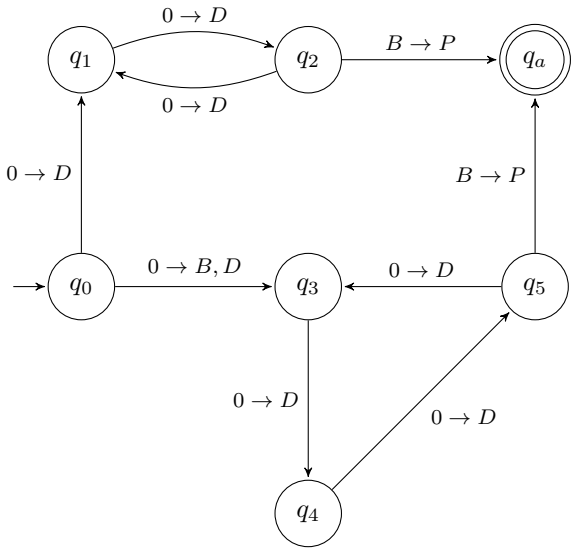


Figura 4.11: Uma máquina de Turing não-determinística.

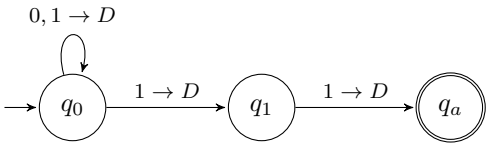


Figura 4.12: Outra máquina de Turing não-determinística.

Dada uma MTND e uma palavra w, a palavra é *aceite* se existe pelo menos um ramo de computação que, iniciada com a palavra w na fita e no estado inicial, termina num estado final. Por exemplo, no caso da MTND anterior, a palavra 000 é aceite já que existe pelo menos um ramo de computação (o que passa pelo estado q_2) que aceita a palavra w. Não é difícil ver que uma palavra w é aceite por esta MTND se e só se não for a palavra vazia ε e se for constituída por um número de zeros que é múltiplo de 2 ou múltiplo de 3 (o ramo de cima aceita as palavras com um número de 0's múltiplo de 2, enquanto que o ramo de baixo aceita as palavras com um número de 0's múltiplo de 3). Logo esta máquina aceita todas as palavras do conjunto $L_1 = \{0^{2k} \in \{0\}^* | k \geq 1\} \cup \{0^{3k} \in \{0\}^* | k \geq 1\}$ (por exemplo, $0 \notin L_1, 00000 \notin L_1$, mas $00 \in L_1, 000 \in L_1$). De forma semelhante ao que fizemos para as máquinas de Turing, podemos dizer que esta MTND *aceita* a linguagem L_1. Mais, como todos os ramos de computação terminam em tempo finito, podemos dizer que a MTND *decide* a linguagem L_1.

De forma semelhante, a MTND da Fig. 4.12 decide a linguagem $L_2 = \{w \in \{0,1\}^* :$ 11 é uma subpalavra de $w\}$, onde se diz que $y \in \Sigma^*$ é uma subpalavra de $w \in \Sigma^*$ se existem $x, z \in \Sigma^*$ tais que $w = xyz$.

Formalmente, uma MTND é definida da seguinte forma.

Definição 4.3.1. Uma *máquina de Turing não-determinística* é um 7-tuplo $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ definido como no caso da máquina de Turing, com a exceção que $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q \times \Sigma \times \{D, E, P\})$, onde $\mathcal{P}(A) = \{X | X \subseteq A\}$.

Repare-se que agora a uma configuração podem-se seguir várias configurações. Uma palavra $w \in \Sigma^*$ é *aceite* se existe uma sequência de configurações $C_1, \ldots, C_k$, tal que C_1 é a configuração inicial para w, C_{i+1} é uma das possíveis configurações que se segue a C_i, e C_k é uma configuração aceitadora. Se M é uma MTND, então a sua linguagem é conjunto de todas as palavras aceites por M:

$$L(M) = \{w \in \Sigma^* | w \text{ é aceite por } M\}$$

Se, adicionalmente, a MTND termina cada ramo da computação num tempo finito, para qualquer *input*, então M *decide* a linguagem $L(M)$.

Obviamente que toda a MT é também não-determinística (a definição de MTND também inclui o caso em que a cada configuração se lhe segue, no máximo, apenas uma configuração, tal como acontece para MT "normais"). A relação recíproca também é verdadeira, como mostra o seguinte resultado.

Teorema 4.3.2. *Se $L(M)$ é a linguagem aceite (decidida) por uma máquina de Turing não-determinística M, então existe uma máquina de Turing que também aceita (decide) $L(M)$.*

Demonstração. A ideia desta demonstração é relativamente simples, embora precisemos de ter cuidado nos detalhes. Basicamente criamos uma MT que simula todos os caminhos de computação de M, e que aceita um *input* w se houver um caminho de computação de M que aceita w. Deste modo a MT vai aceitar exatamente as mesmas palavras de M.

Vamos primeiro considerar o caso em que $L(M)$ é a linguagem decidida pela MTND M. Para simular a MT não-determinística M, e porque é mais prático, iremos utilizar uma MT com 3 fitas, que já sabemos ser equivalente a uma MT "standard" com apenas uma fita, pela tese de Church-Turing (mais concretamente, poderia-se mostrar que uma MT com k fitas pode ser sempre simulada por uma MT com uma única fita). A ideia é utilizar uma estratégia exaustiva, em que simulamos cada possível caminho de computação de M com uma MT, mantendo o registo dos caminhos explorados e dos não-explorados. Se nalgum caminho o *input* é aceite, então a MT aceita o *input*. Se, depois de simulados todos os caminhos, se verificou que o *input* não é aceite em nenhum deles, então o *input* é rejeitado.

Utilizamos uma MT com 3 fitas para simular M: mantemos uma cópia do *input* na fita 1, utilizamos a fita 2 para simular uma computação de M (de entre as várias possibilidades de caminhos de computação), e a fita 3 é utilizada para manter um registo de todos os caminhos de computação explorados até agora. Inicialmente copia-se o *input* da fita 1 para a fita 2, e testa-se um caminho de computação na fita 2. Se esse

caminho aceita, aceitamos o *input*, caso contrário, reiniciamos a computação com um novo caminho de computação, e vamos repetindo o processo até que não reste mais nenhum caminho de computação por explorar.

Dada a função de transição de M, δ, terá de existir um número k tal que, para cada $(q, s) \in Q \times \Sigma$, existem no máximo k escolhas para a próxima ação a realizar. Portanto, em cada transição não-determinística, podemos registar um número em $\{1, \ldots, k\}$ para saber qual a ação que foi tomada (note-se que em certas transições não-determinísticas, pode-se ter "apenas" $j \geq 2$ transições não-determinísticas, com $j < k$. Nesses casos pode-se, por exemplo, dizer que as transições correspondentes aos números $j + 1, \ldots, k$ correspondem à transição associada ao número 1, i.e. pode-se repetir transições, ou melhor ainda, podemos logo terminar as computações correspondentes a estes casos). O que a nossa MT vai fazer é registar esses números na 3ª fita e testar todas as possibilidades em $\{1, \ldots, k\}$ i.e. vai testar todas as possíveis transições, uma de cada vez, de forma a simular todas as possíveis computações de M (que acabam sempre e são em número finito, pois estamos a *decidir* uma linguagem — todas as computações têm de acabar). Vamos agora descrever a MT procurada:

1. Inicialmente a fita 1 contém o *input* w e as fitas 2 e 3 estão vazias.

2. Copiar o conteúdo da fita 1 para a fita 2.

3. Utilizar a fita dois para simular a máquina M com *input* w. Cada vez que temos de efetuar uma nova transição não-determinística, lemos o símbolo atualmente lido na fita 3 pela respetiva cabeça de leitura. Se esse símbolo for um símbolo em $\{1, \cdots, k\}$, efetuamos a transição de acordo com o símbolo de $\{1, \cdots, k\}$ que estiver lá escrito, e depois movemos a cabeça de leitura da fita 3 uma célula para a direita. Se o símbolo lido for um B, escrevemos um 1 por cima e efetuamos a transição respetiva, e depois movemos a cabeça de leitura da fita 3 uma célula para a direita. Se no decurso da simulação chegarmos a uma configuração aceitadora, então aceitamos o *input*. Se chegarmos a uma configuração rejeitadora, então ir para o passo 4.

4. Se a fita 3 está vazia, rejeitar o *input*. Se a fita 3 tem a palavra $s = s_1 \ldots s_j \neq \varepsilon$, escrita no alfabeto $\{1, \ldots, k\}$, então: (i) Se $s_j \neq k$, então devemos substituir s_j pelo símbolo seguinte da ordenação dos símbolos (ou seja pelo símbolo $s_j + 1$), e depois colocar a cabeça de leitura da fita 3 no símbolo s_1 da nova palavra s atualizada. Depois devemos ir novamente para o passo 1. Se $s_j = k$ (ou seja, se não há símbolo seguinte), apagar s_j e repetir o passo 4.

Para o caso das linguagens *aceites* por M, a situação é mais complicada, pois pode haver ramos em que a computação nunca acabe, pelo que não podemos testar cada ramo até ao fim, de forma sequencial, como fizemos no caso anterior (se a computação num ramo nunca termina, o algoritmo anterior fica "encravado" nesse ramo, e nunca chega a conseguir explorar outros ramos que poderiam aceitar o *input*). Mas esse problema

pode ser evitado utilizando mais uma 4^a fita, que mantém um registo de um contador de passos. Sempre que o número de passos de um ramo de computação ultrapassar o valor definido pelo contador, terminamos essa computação (como se ela rejeitasse), e passamos para o ramo de computação seguinte, impedindo que a MT "encrave" num ramo. O algoritmo é o seguinte:

1. Inicialmente a fita 1 contém o *input* w e as fitas 2 e 3 estão vazias. A fita 4 contém o valor $i = 1$
2. `While` $i > 0$ `do`
   (a) Copiar o conteúdo da fita 1 para a fita 2.
   (b) Utilizar a fita dois para simular a máquina M com *input* w. Cada vez que temos de efetuar uma nova transição não-determinística, lemos o símbolo atualmente lido na fita 3 pela respetiva cabeça de leitura. Se esse símbolo for um símbolo em $\{1, \cdots, k\}$, efetuamos a transição de acordo com o símbolo de $\{1, \cdots, k\}$ que estiver lá escrito, e depois movemos a cabeça de leitura da fita 3 uma célula para a direita. Se o símbolo lido for um B, escrevemos um 1 por cima e efetuamos a transição respetiva, e depois movemos a cabeça de leitura da fita 3 uma célula para a direita. Se no decurso da simulação chegarmos a uma configuração aceitadora, então aceitamos o *input*. Se chegarmos a uma configuração rejeitadora, ou a computação necessitar mais do que i passos, ir para o passo (c).
   (c) Se a fita 3 está vazia, ir para o passo (d). Se a fita 3 tem a palavra $s = s_1 \dots s_j \neq \varepsilon$, escrita no alfabeto $\{1, \dots, k\}$, então: (i) Se $s_j \neq k$, então devemos substituir s_j pelo símbolo seguinte da ordenação dos símbolos (ou seja pelo símbolo $s_j + 1$), e depois colocar a cabeça de leitura da fita 3 no símbolo s_1 da nova palavra s atualizada. Depois devemos ir novamente para o passo (a). Se $s_j = k$ (ou seja, se não há símbolo seguinte), apagar s_j e repetir o passo (c).
   (d) Incrementar o valor de i.

□

Vamos dar um exemplo de como esta demonstração funciona na prática (caso em que se decide uma linguagem). Consideremos a MT da Fig. 4.11. Temos transições não-determinísticas partindo de q_1, q_2, q_3, que utilizam duas escolhas possíveis. Codifiquemos essas escolhas nos números binários 1 e 2 da seguinte forma:

$$\delta(q_1, B) = \begin{cases} (q_2, B, D) & \text{se o código é 1} \\ (q_3, B, D) & \text{se o código é 2} \end{cases} \qquad \delta(q_2, 0) = \begin{cases} (q_4, 0, D) & \text{se o código é 1} \\ (q_5, 0, D) & \text{se o código é 2} \end{cases}$$

$$\delta(q_3, 1) = \begin{cases} (q_7, 1, D) & \text{se o código é 1} \\ (q_9, 1, D) & \text{se o código é 2.} \end{cases}$$

Consideremos o *input* 00. Começamos no estado q_0 e depois vamos para o estado q_1. Daqui vamos querer fazer uma nova transição, mas temos duas possibilidades. Vamos ter de registar essas possibilidades na fita 3. A fita 3 está inicialmente vazia, pelo que escrevemos um 1 nessa fita, e depois deslocamos a cabeça desta fita uma célula para a direita, e vamos para o estado q_2. Temos novamente uma transição não-determinística no estado q_2, pois estamos a ler um B. Escrevemos novamente um 1 na fita 3 e continuamos com a computação (a fita 3 tem agora o conteúdo "11"), chegando ao estado q_4 que rejeita esta computação. Agora vamos experimentar outro caminho de computação. Para isso, vamos para o passo 4 da simulação, alterando o conteúdo da fita 3 para "12", e metendo a cabeça de leitura dessa fita sobre o 1 inicial. Repetimos a computação com o *input* 00: começamos em q_0, vamos para q_1, e como estamos a ler um 1 na fita 3, vamos para o estado q_2, movendo a cabeça desta fita uma célula para a direita. Depois temos nova transição não-determinística. Como estamos a ler um 2 na 2ª fita, vamos agora para o estado q_5 que também rejeita o *input*. Fazemos novamente o passo 4 (experimentar outro caminho de computação). Como já não há mais símbolos para além do 2 em $\{1, 2\}$, devemos apagar o 2 de "12" (o conteúdo da fita 3 passa a ser "1") e aplicamos novamente o passo 4, ficando o conteúdo da fita 3 "2". Repete-se a computação, passando pelos estados q_0, q_1, q_3 e a computação "morre" nesse momento. Repete-se novamente o passo 4, apagando o "2". Neste momento a 3ª fita fica vazia (já não há novas possibilidades de caminhos de computação), e o *input* é rejeitado.

4.4 Tese de Church-Turing

Uma pergunta que um engenheiro informático se pode colocar é a seguinte:

> *Dado um determinado problema, será que existe um algoritmo (programa) que o resolve?*

A história, aliás, começa em 1900. Por ocasião da mudança do novo século, um dos mais famosos matemáticos da altura, David Hilbert, apresentou no Congresso Internacional da Matemática, em Paris, uma lista de 23 problemas matemáticos em aberto, que ele julgava serem problemas fundamentais. O décimo problema consistia na seguinte questão:

> *Será que existe um algoritmo que permite determinar se uma dada equação diofantina tem soluções?*

Por outras palavras, será que existe um algoritmo geral que receba como *input* a descrição de um polinómio p, com coeficientes inteiros, e que como *output* indique se a equação $p(x) = 0$ tem ou não soluções inteiras. Na altura pensava-se que não existia nenhum algoritmo nessas condições, mas a grande questão era como provar essa conjetura.

Se pensarmos um pouco no assunto, verificamos que o principal problema é que não temos uma definição precisa para a noção de algoritmo. Esta noção é sobretudo intuitiva: sabemos identificar um algoritmo quando o vemos, mas não sabemos defini-lo exatamente. É como tentar identificar um(a) homem/mulher bonito(a).

O que vários investigadores fizeram, sobretudo durante a década de 1930, foi formalizar o conceito de algoritmo. Vários cientistas levaram a cabo esta tarefa, obtendo modelos mais ou menos abstratos, que não convenceram toda a comunidade cientifica. Mas, em 1936, com a sua máquina tão simples e convincente, Turing conseguiu impor o seu modelo.

Para além da sua simplicidade e naturalidade, uma das razões que possibilitou a aceitação generalizada da máquina de Turing como sendo um modelo de computação que captura a noção de algoritmo, foram diversos resultados que mostraram que todos os outros modelos apresentados anteriormente eram equivalentes à máquina de Turing. Isso está refletido na seguinte afirmação, que é hoje aceite na comunidade científica:

> **Tese de Church-Turing:** *Uma função é "efectivamente computável" se e só se pode ser computada através de uma máquina de Turing.*

Por uma função "efectivamente computável" entende-se qualquer função que seja "naturalmente considerada computável", no sentido que essa função pode ser calculada por um algoritmo que tenha as seguintes propriedades:

- O algoritmo consiste num número finito de instruções simples e precisas, que são descritas com um número finito de símbolos;
- O algoritmo pode, em princípio, ser executado por um ser humano com apenas papel e lápis:
- A execução do algoritmo não requer inteligência do ser humano para além do necessário para entender e executar as instruções.

Em particular uma função que seja computada através de um algoritmo definido numa qualquer linguagem de programação (por exemplo, *Java*, *Python*, *C/C++*, etc.) satisfaz as condições indicadas acima, sendo portanto efetivamente computável.

Note-se que a Tese de Church-Turing não se pode provar, daí o nome de "tese" porque, como já vimos, a noção de algoritmo é informal. O que esta tese vem fazer é dar um sentido preciso à noção de algoritmo. Nesta disciplina iremos sempre assumir que a tese de Church-Turing é válida.

Por esta razão, a não ser que seja pedida uma descrição formal de uma MT (caso em que se deve utilizar um diagrama ou a Definição 4.2.1), iremos muitas vezes descrever uma MT através de um algoritmo que especifique os seus passos, sem entrar em detalhes de estados, etc. No entanto, a descrição deve ser suficientemente precisa para não deixar dúvidas a quem estiver a ler a descrição.

4.5 Indecibilidade e o problema da paragem

Dissemos anteriormente que a MT pode ser considerada como um modelo teórico de um computador. Mas, do que vimos até agora, parece que para cada algoritmo diferente, somos obrigados a utilizar MT's com "hardware" (i.e. estados e transições) diferentes, enquanto que na prática o hardware mantém-se, e apenas se troca de programa. Não haverá aqui um problema? Não, porque não é difícil conceber (mais detalhes podem ser encontrados em [HMU06]) uma MT U que recebe como *input* $\langle M, w\rangle$, onde $\langle M, w\rangle$ designa a descrição da MT M e da palavra w numa palavra do alfabeto de U. Com base nessa descrição, U simula M com o *input* w, para quaisquer M, w, de forma semelhante ao que um interpretador de C/C++ ou Java faz. A esta máquina dá-se o nome de *máquina de Turing universal*.

Agora vamos voltar a nossa atenção para um problema prático que pode aparecer. Normalmente os ambientes de programação trazem embutidos ferramentas que permitem detetar facilmente erros de sintaxe, como por exemplo um comando "`for`" que foi erroneamente escrito como "`fro`". Mas os erros que dão mais dores de cabeça aos programadores são os erros de semântica: o programa é sintaticamente correto, mas não faz o que é pretendido. Um caso particular é quando o programa "encrava", porque entrou numa computação infinita.

Uma pergunta que é natural fazer é porque é que ninguém desenvolveu um software que permita dizer antecipadamente se, dado um programa em C/C++, por exemplo, se este vai ou não entrar num ciclo infinito de computação? A razão é simples e é uma consequência da teoria da computação: tal software não existe!

Como mostrar isso? O problema que queremos tratar, traduzido na linguagem de teoria da computação, será: dada a linguagem

$$H_{paragem} = \{\langle M, w\rangle | \text{ a máquina de Turing } M \text{ para com o } \textit{input } w\} \tag{4.3}$$

será que ela é recursiva? Se sim, o software mencionado acima existe, caso contrário o software não existe. Apesar desta linguagem ser recursivamente enumerável, ela não é recursiva, como mostra o seguinte teorema. Esta linguagem é geralmente designada de $HALT_{TM}$ na literatura (vem do nome em inglês).

Note-se que a uma linguagem A está associado o *problema de decisão* "$u \in A$?". Na literatura é comum trabalhar simultaneamente trabalhar com linguagens e o seu problema de decisão, sem fazer uma distinção explícita entre os dois.

Teorema 4.5.1 (O problema da paragem é indecidível). *A linguagem definida por (4.3) é recursivamente enumerável, mas não é recursiva.*

Demonstração. Primeiro vamos mostrar que a linguagem $H_{paragem}$ é recursivamente enumerável. A MT M_{ac} descrita pelo seguinte algoritmo aceita $H_{paragem}$:

- Com *input* $u = \langle M, w\rangle$:

1. Se u não codifica uma MT M e uma palavra w, então rejeitar o *input* u
2. Simular M com *input* w.
3. Se M para, então aceitar o *input* $u = \langle M, w \rangle$.

Note-se que se M para com *input* w, fá-lo-á em tempo finito, e logo a MT M_{ac} vai aceitar $\langle M, w \rangle$ em tempo finito. Se M não para com *input* w, então M_{ac} não para com *input* $\langle M, w \rangle$. Logo M_{ac} aceita $H_{paragem}$, e portanto $H_{paragem}$ é recursivamente enumerável. Vamos mostrar que $H_{paragem}$ não é recursiva, utilizando uma técnica conhecida como *diagonalização*.

Por absurdo, suponhamos que a linguagem (4.3) é recursiva. Então há-de existir uma MT N tal que

$$N(\langle M, w \rangle) = \begin{cases} \text{aceita} & \text{se } \langle M, w \rangle \in H_{paragem} \\ \text{rejeita} & \text{se } \langle M, w \rangle \notin H_{paragem}. \end{cases}$$

Por outras palavras,

$$N(\langle M, w \rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ para com } \textit{input } w \\ \text{rejeita} & \text{se } M \text{ não para com } \textit{input } w. \end{cases}$$

Agora construímos outra MT P, que utiliza N como sub-rotina da seguinte forma:

$$P(\langle M \rangle) = \begin{cases} \text{aceita } \langle M \rangle & \text{se } N \text{ rejeita } \langle M, \langle M \rangle \rangle \\ \text{entra num ciclo infinito} & \text{se } N \text{ aceita } \langle M, \langle M \rangle \rangle \end{cases}$$

o que também pode ser lido como

$$P(\langle M \rangle) = \begin{cases} \text{aceita } \langle M \rangle & \text{se } M \text{ não para com } \textit{input } \langle M \rangle \\ \text{entra num ciclo infinito} & \text{se } M \text{ para com } \textit{input } \langle M \rangle. \end{cases}$$

Se não parecer óbvio o que é a MT P, podemos utilizar uma abordagem em que utilizamos programas informáticos. Por exemplo, se $H_{paragem}$ é decidível, isso quer dizer que existe um programa que computa uma função $ProblemaParagem$ com dois argumentos tal que $ProblemaParagem(\langle M \rangle, w)$ retorna 1 se $\langle M, w \rangle \in H_{paragem}$ e retorna 0 caso contrário. Então usando o programa que calcula a função $ProblemaParagem$, podemos criar um novo programa P que se comporta com a MT P descrita acima, do seguinte modo:

Então temos

$$P(\langle P \rangle) = \begin{cases} \text{aceita } \langle P \rangle & \text{se } P \text{ não para com } \textit{input } \langle P \rangle \\ \text{entra num ciclo infinito} & \text{se } P \text{ para com } \textit{input } \langle P \rangle. \end{cases}$$

o que é absurdo. No caso do programa P dado em cima isso é equivalente a dar ao programa como *input* o seu próprio código. Concluiríamos que P retorna 1 se nunca parar – um absurdo – ou então entra num ciclo infinito se parar – o que também é absurdo). Portanto a linguagem (4.3) não pode ser recursiva. □

Algoritmo 1 Programa P

Require: String binária w

```
1: function P(w)
2:     if ProblemaParagem(w, w) = 0 then
3:         return 1                                  ▷ Aceita o input
4:     else
5:         while 0 ≠ 1 do                            ▷ Entra num ciclo infinito
6:             loop=1
7:         end while
8:     end if
9: end function
```

Comentários:

- **Linha 2:** O programa (rotina) que computa a função $ProblemaParagem$ é-nos fornecida previamente. Este programa recebe duas palavras binárias u e w como *input*. Se u codifica uma máquina de Turing M, então $ProblemaParagem(u, w)$ retorna 1 se $\langle M, w\rangle \in H_{paragem}$, e retorna 0 caso contrário.

De forma semelhante, podemos mostrar que outras linguagens são também indecidíveis.

Teorema 4.5.2 (**O problema da aceitação é indecidível**). *A linguagem definida por*

$$A_{TM} = \{\langle M, w\rangle | \text{ a máquina de Turing } M \text{ aceita o input } w\}$$

é recursivamente enumerável, mas não é recursiva.

Demonstração. A demonstração é muito semelhante à que foi utilizada para demonstrar que $H_{paragem}$ é indecidível. Podemos mostrar que a linguagem A_{TM} é recursivamente utilizando uma MT M_{ac} descrita pelo seguinte algoritmo:

- Com *input* $u = \langle M, w\rangle$:
  1. Se u não codifica uma MT M e uma palavra w, então rejeitar o *input* u
  2. Simular M com *input* w.
  3. Se M para e:
     - Aceita, então aceitar o *input* $\langle M, w\rangle$.
     - Rejeita, então rejeitar o *input* $\langle M, w\rangle$.

Note-se que se M aceita o *input* w, fá-lo-á em tempo finito, e logo a MT M_{ac} vai aceitar $\langle M, w\rangle$ em tempo finito. Se M não aceita o *input* w, então M_{ac} ou rejeita o *input* w ou não para com *input* $\langle M, w\rangle$. Logo M_{ac} aceita A_{TM}, e portanto A_{TM} é recursivamente enumerável. Vamos mostrar que A_{TM} não é recursiva, utilizando novamente a técnica de diagonalização.

Por absurdo, suponhamos que a linguagem A_{TM} é recursiva. Então há-de existir uma MT N tal que

$$N(\langle M, w\rangle) = \begin{cases} \text{aceita} & \text{se } \langle M, w\rangle \in A_{TM} \\ \text{rejeita} & \text{se } \langle M, w\rangle \notin A_{TM}. \end{cases}$$

Por outras palavras,

$$N(\langle M, w\rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ aceita o } \textit{input } w \\ \text{rejeita} & \text{se } M \text{ não aceita o } \textit{input } w. \end{cases}$$

Agora construímos outra MT P, que utiliza N como sub-rotina da seguinte forma:

$$P(\langle M\rangle) = \begin{cases} \text{aceita } \langle M\rangle & \text{se } N \text{ rejeita } \langle M, \langle M\rangle\rangle \\ \text{rejeita} & \text{se } N \text{ aceita } \langle M, \langle M\rangle\rangle \end{cases}$$

o que também pode ser lido como

$$P(\langle M\rangle) = \begin{cases} \text{aceita } \langle M\rangle & \text{se } M \text{ não aceita o } \textit{input } \langle M\rangle \\ \text{rejeita} & \text{se } M \text{ aceita o } \textit{input } \langle M\rangle. \end{cases}$$

Então temos

$$P(\langle P\rangle) = \begin{cases} \text{aceita } \langle P\rangle & \text{se } P \text{ não aceita o } \textit{input } \langle P\rangle \\ \text{rejeita} & \text{se } P \text{ aceita o } \textit{input } \langle P\rangle \end{cases}$$

o que é absurdo. Portanto a linguagem A_{TM} não pode ser recursiva. □

Também se poderia mostrar o resultado anterior sabendo de antemão que $H_{paragem}$ é indecidível. Para simplificar o argumento, suponhamos que estamos a lidar com programas de computador. Se supormos, por absurdo, que existe um programa de computador ProblemaAceitacao que decide a linguagem A_{TM}, i.e. este programa de computador recebe duas palavras binárias u, v e retorna 1 se u codifica uma MT M e esta MT aceita w, e retorna 0 caso contrário. Então daí podemos concluir que o programa ProblemaParagem dado pelo Algoritmo 2 também decide $H_{paragem}$, como iremos ver de seguida, o que é um absurdo.

Todo o argumento indicado acima poderia ser escrito à custa de máquinas de Turing. Vamos agora considerar um programa TransformarInput que modifica o código da MT M, codificada pela palavra u, de forma a criar uma nova MT M' que para sempre que M para e que, além do mais, sempre que M' para então aceita. Ou seja

$$\textsc{TransformarInput}(M)(w) = \begin{cases} \text{aceita } w & \text{se } M \text{ para com } \textit{input } w \\ \text{nunca para} & \text{se } M \text{ não para com } \textit{input } w \end{cases}$$

Logo $\langle M, w\rangle \in H_{paragem}$ sse $\langle \textsc{TransformarInput}(\langle M\rangle, w)\rangle \in A_{TM}$. Dado que a função TransformarInput é computável, então se conseguirmos decidir o problema da aceitação, conseguimos também decidir o problema da paragem, o que é absurdo. Esta observação pode ser generalizada.

Definição 4.5.3. Dadas duas linguagens $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, diz-se que há uma redução computável de L_1 a L_2, ou simplesmente que L_1 se reduz a L_2, o que se denota por $L_1 \leq L_2$ se existe existe uma função total (i.e definida para todos os seus argumentos) computável $f : \Sigma_1^* \to \Sigma_2^*$ tal que para todo o $w \in \Sigma_1^*$ se tem

$$w \in L_1 \text{ se e só se } f(w) \in L_2$$

No caso do exemplo anterior, o que nós essencialmente mostramos foi que $H_{paragem} \leq A_{TM}$, utilizando como função f a função $\mathrm{TransformarInput}$.

Teorema 4.5.4. *Se $L_1 \leq L_2$ e L_2 é decidível, então L_1 é decidível*

Demonstração. Se L_2 é decidível, então existe uma MT (programa) M_2 que decide esta linguagem. Além do mais, como f é computável, o valor desta função também pode ser determinada através de uma MT (programa). Então a MT descrita pelo algoritmo 3 decide L_1. □

Algoritmo 2 Programa ProblemaParagem

Require: String binária u
Require: String binária w

```
 1: function TransformarInput(u, w)
 2:     if u não codifica nenhuma máquina de Turing then
 3:         return u, v                          ▷ Não faz nada neste caso
 4:     else
 5:         u=Replace(u, 'Return 0', 'Return 1')
 6:         return u, v
 7:     end if
 8: end function
 9: function ProblemaParagem(u, w)
10:     if ProblemaAceitacao(TransformarInput(u, w)) = 0 then
11:         return 0                             ▷ Rejeita o input
12:     else
13:         return 1                             ▷ Aceita o input
14:     end if
15: end function
```

Comentários:

- **Linha 5:** A instrução 'Replace' substitui todos os comandos 'Return 0' por comandos 'Return 1' no código da MT u.
- **Linha 10:** O programa que computa a função ProblemaAceitacao é-nos fornecida previamente. Este programa recebe duas palavras binárias u e w como *input*. Se u codifica uma máquina de Turing M, então ProblemaAceita(u, w) retorna 1 se M aceita w, e retorna 0 caso contrário.

Algoritmo 3 Programa $DecideL_1$

Require: String binária w

```
1: function DecideL1(w)
2:     if M2 rejeita f(w) then
3:         return 0                          ▷ Rejeita o input
4:     else
5:         return 1                          ▷ Aceita o input
6:     end if
7: end function
```

Comentários:

- **Linha 2:** f é a função computável utilizada para mostrar que $L_1 \leq L_2$ e M_2 é a MT que decide L_2.

Corolário 4.5.5. *Se $L_1 \leq L_2$ e L_1 é indecidível, então L_2 é indecidível*

No caso de A_{TM}, como vimos que $H_{paragem}$ é indecidível e que $H_{paragem} \leq A_{TM}$, o corolário anterior dá-nos que A_{TM} também tem de ser indecidível.

Antes de terminar este capítulo, vamos ver mais um exemplo de linguagem indecidível.

Teorema 4.5.6. *A linguagem*

$$NOTEMPTY_{TM} = \{\langle M \rangle : M \text{ é uma MT e } L(M) \neq \emptyset\}$$

não é recursiva.

Demonstração. Vamos mostrar que $A_{TM} \leq NOTEMPTY_{TM}$. Como já sabemos que A_{TM} é indecidível, isso irá permitir-nos concluir que $NOTEMPTY_{TM}$ é indecidível. Logo, temos de determinar uma função computável f tal que $u \in A_{TM}$ sse $f(u) \in NOTEMPTY_{TM}$. Desta forma, se houvesse uma MT que decidisse $NOTEMPTY_{TM}$, então conseguiríamos criar uma MT que decidisse A_{TM}, o que é um absurdo.

Note-se que se $u \in A_{TM}$, então $u = \langle M, w \rangle$, e se $f(u) \in A_{TM}$, então $f(u)$ tem de codificar uma MT $N_{M,w}$, cuja definição irá depender de M e w, se quisermos que $\langle M, w \rangle \in A_{TM}$ sse $f(\langle M, w \rangle) = N_{M,w} \in NOTEMPTY_{TM}$. A ideia será que a linguagem desta máquina de Turing seja a seguinte:

$$L(N_{M,w}) = \begin{cases} \emptyset & \text{se } M \text{ não aceita } w \\ \{w\} & \text{se } M \text{ aceita } w. \end{cases}$$

Nesse caso, não é difícil verificar que se tem $\langle M, w \rangle \in A_{TM}$ sse $f(\langle M, w \rangle) = N_{M,w} \in NOTEMPTY_{TM}$. Falta apenas determinar como se constrói a MT $N_{M,w}$, i.e. como se determina $f(w)$.

Dado um *input* v, o que $N_{M,w}$ faz com este *input* é verificar se $v = w$. Se $v \neq w$, então $N_{M,w}$ automaticamente rejeita v. Se $v = w$, então $N_{M,w}$ simula M com *input* w e $N_{M,w}$ aceita $v = w$ sse M aceita w. Note-se que nesse caso se tem que $L(N_{M,w}) \neq \emptyset$ sse M aceita w, como pretendido. □

4.5.1 Indecibilidade na lógica de primeira ordem

Tivemos ocasião de ver no cálculo proposicional que existe uma forma automática de ver se uma fórmula é válida ou não: basta construir a sua tabela verdade, e ver se o resultado dá sempre 1. Alternativamente podemos utilizar um tableau semântico, e sabemos que a sua construção termina sempre.

Será que isto pode ser efetuado para a lógica de primeira ordem, i.e. será que existe uma forma algorítmica de ver se uma fórmula é logicamente válida ou não? A resposta é não, como mostra o seguinte teorema, que enunciamos sem demonstração (note-se que qualquer fórmula ϕ da lógica de primeira ordem pode também ser codificada, por exemplo, numa palavra binária $\langle\phi\rangle$).

Teorema 4.5.7 (**Church**)**.** *O problema de determinar se uma fórmula da lógica de primeira ordem é válida ou não é indecidível (e, por dualidade, também não é possível decidir se a fórmula é satisfazível), i.e. a linguagem*

$$\{\langle\phi\rangle : \phi \textit{ é uma fórmula logicamente válida da lógica de primeira ordem}\}$$

não é recursiva.

Capítulo 5

Complexidade computacional

5.1 Introdução

No capítulo anterior vimos que há problemas, como o problema da paragem, que não podem ser resolvidos (decididos) por meio de algoritmos. No entanto, a experiência diz-nos que a separação decidível versus indecidível não é suficientemente fina para capturar a classe dos problemas que se conseguem resolver na prática.

É que, embora certos problemas sejam teoricamente resolúveis através de algoritmos, não o são na prática porque o algoritmo que os resolve poderá demorar biliões de anos a ser executado. Por isso temos necessidade de dividir a classe dos problemas decidíveis em subclasses, de acordo com a quantidade de recursos utilizados (sejam eles tempo de computação, memória utilizada, ou outros. Neste capítulo vamos apenas cingir-nos a limitações no tempo de computação).

Em geral, o tempo exato que um algoritmo demora a ser executado pode ser exprimido através uma expressão complexa. Mas, de forma a não nos perdermos nos detalhes, estaremos apenas interessados na ordem de grandeza dos tempos de execução de um algoritmo. Por exemplo, se um algoritmo pode ser executado em $\leq 5n^2 + 7n + 2$ passos para um *input* de tamanho n (é natural que a computação demore mais tempo para *inputs* maiores, não significando isso que o algoritmo seja menos eficiente), diremos simplesmente que o algoritmo demora tempo da ordem de n^2 a ser executado, pois o grosso do incremento do tempo de computação vem da potência n^2, dizendo-se neste caso que o algoritmo necessita de tempo $O(n^2)$. Vamos agora introduzir formalmente estas noções.

Definição 5.1.1. Sejam $f, g : \mathbb{N} \to \mathbb{N}$ duas funções. Dizemos que g é um *limite superior (assimptótico)* para f, e escreve-se $f \in O(g)$, se existem inteiros c, n_0 tais que para todo o inteiro $n \geq n_0$ se tem

$$f(n) \leq cg(n).$$

Por exemplo, $n \in O(n)$, $n^2 \in O(n^4)$, $5n \in O(n)$, mas $n^2 \notin O(n)$. Note-se que, por abuso de notação, é frequente escrever $f(n) \in O(g(n))$ em vez de $f \in O(g)$ (no

primeiro caso estamos a falar de *imagens* de funções, enquanto que no segundo caso estamos a falar de funções, como pretendido). O seguinte teorema é muitas vezes útil para determinar se $f \in O(g)$.

Teorema 5.1.2. *Sejam $f, g : \mathbb{N} \to \mathbb{R}_0^+$ funções tais que*

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = r \in \mathbb{R}_0^+.$$

Então $f \in O(g)$

Vamos agora introduzir uma definição que será bastante importante para o que se segue.

Definição 5.1.3. Seja M uma máquina de Turing que para em todos os *inputs*. O *tempo de execução* de M é uma função $f : \mathbb{N} \to \mathbb{N}$, onde $f(n)$ é o maior número de passos que M utiliza para concluir uma computação com um *input* de tamanho n.

Definimos também a seguinte classe:

$$TIME(f) = \{L : L \text{ é uma linguagem decidida em tempo } O(f) \text{ por alguma MT}\}.$$

Por exemplo, a MT da Fig. 4.1 tem tempo de execução $O(n^2)$. De facto, dado um *input* $w = 0^k 1^k 0^k$ com comprimento $n(= 3k)$, a MT marca um zero do 1º bloco de zeros, e depois vai para à direita até ao final do *input*, marcando um 1 e um 0 do último bloco de 0's no caminho. Depois volta para trás até ao 1º bloco de zeros. Esta passagem demora tempo $\leq 2n$, ou seja demora tempo $O(n)$. No máximo vão haver $k \leq n$ passagens destas (uma por cada 0 do primeiro bloco de 0's), pelo que o tempo total de computação é limitado por $n \times O(n) \in O(n^2)$. Se o *input* não é do tipo $0^k 1^k 0^k$, i.e. se o *input* é rejeitado, a computação acaba prematuramente, pelo que o tempo de execução continua a ser $O(n^2)$. Como esta máquina decide a linguagem $L_1 = \{0^k 1^k 0^k \in \{0,1\}^* | k \in \mathbb{N}\}$, concluímos que $L_1 \in TIME(n^2)$.

De forma semelhante, concluímos que a MT da Fig. 4.9 decide a linguagem $L_2 = \{0^k 1^k \in \{0,1\}^* | k \geq 1\}$ em tempo $O(n)$, e que $L_2 \in TIME(n)$.

Da mesma forma que definimos o tempo de execução para uma MT, podemos também fazê-lo para uma MTND.

Definição 5.1.4. Seja M uma máquina de Turing não-determinística que para em todos os ramos de computação para qualquer *input*. O *tempo de execução* de M é uma função $f : \mathbb{N} \to \mathbb{N}$, onde $f(n)$ é o maior número de passos que M utiliza para concluir um ramo de computação para um *input* de tamanho n. Definimos também a seguinte classe:

$$NTIME(f) = \{L : L \text{ é uma linguagem decidida em tempo } O(f) \text{ por uma MTND}\}.$$

Por exemplo, a MTND da Fig. 4.11 tem tempo de execução $O(n)$ já que, com *input* $w \in \{0\}^*$ em que $|w| = n$, cada ramo de computação necessita de tempo $O(n)$ para ser concluído. Logo $L_3 = \{0^{2k} \in \{0\}^* | k \geq 1\} \cup \{0^{3k} \in \{0\}^* | k \geq 1\} \in NTIME(n)$. O tempo de execução da MTND da Fig. 4.11 é $O(1)$, já que todo o *input* pode ser decidido em tempo constante (no máximo em 5 passos).

O seguinte teorema permite relacionar $NTIME(f)$ com $TIME(f)$.

Teorema 5.1.5. *Seja $f : \mathbb{N} \to \mathbb{N}$ uma função. Então $TIME(f) \subseteq NTIME(f)$.*

Demonstração. Para mostrar este resultado basta observar que uma MT é um caso especial de uma MTND em que, dado um estado e um símbolo lido, apenas uma transição é possível (note que a definição 4.3.1, apesar de permitir mais do que uma transição por cada par estado/símbolo lido, não obriga a que isso aconteça – pode existir apenas uma transição por cada par estado/símbolo lido, pelo que uma MT é um caso particular de uma MTND). Logo, dada uma linguagem $A \in TIME(f)$, existe uma MT M que a decide em tempo $O(f)$. Mas como M é também uma MTND, concluímos que existe uma MTND (M) que decide A em tempo $O(f)$, e portanto $A \in NTIME(f)$. Logo $TIME(f) \subseteq NTIME(f)$. □

Embora em geral não se saiba se $NTIME(f) \subseteq TIME(f)$, é possível estabelecer a relação indicada no seguinte teorema, onde $2^{O(f)}$ designa o conjunto das funções $g : \mathbb{N} \to \mathbb{N}$ que têm a seguinte propriedade: existem inteiros c, n_0 (que podem depender de g) tais que para todo o inteiro $n \geq n_0$ se tem

$$g(n) \leq 2^{cf(n)}.$$

Teorema 5.1.6. *Seja $f : \mathbb{N} \to \mathbb{N}$ uma função satisfazendo $f(n) \geq n$ para todo o $n \in \mathbb{N}$. Então $NTIME(f) \subseteq TIME(2^{O(f)})$.*

Demonstração. Basta seguir a demonstração do Teorema 4.3.2. Tomemos um *input* de comprimento n. De notar que, no início da simulação de cada ramo de computação, copiamos o *input* da fita 1 para a fita 2. Isso vai tomar tempo n. Por essa razão assumimos que $f(n) \geq n$, já que cada ramo de computação termina em tempo $O(f)$. Como a palavra escrita na 3ª fita regista as possibilidades de movimentos não-determinísticos, esta palavra não pode ter um comprimento superior ao número de passos de computação da MTND, já que esta palavra pode ir de

$$\varepsilon \quad \text{a} \quad \underbrace{kk\ldots k}_{O(f(n)) \text{ vezes}}$$

Ou seja, temos $k^{O(f(n))} = 2^{O(f(n))}$ possíveis simulações a efetuar, cada uma levando tempo $O(f(n))$. O número total de simulações determinísticas será então $O(f(n)) \times 2^{O(f(n))}$. Mas como $f(n) \leq 2^{f(n)}$, isto é tempo $2^{O(f(n))}$. □

Por outra palavras, uma MT consegue sempre simular uma MTND, mas utilizando uma simulação que tem um custo exponencial em tempo. Por exemplo, como a MTND da Fig. 4.11 tem tempo de execução $O(n)$, concluímos que $L_3 = \{0^{2k} \in \{0\}^* | k \geq 1\} \cup \{0^{3k} \in \{0\}^* | k \geq 1\} \in TIME(2^{O(n)})$, isto é, L_3 pode ser decidida por uma MT em tempo $2^{O(n)}$.

O seguinte teorema mostra que adicionar várias fitas a uma MT apenas permite, no máximo, um ganho quadrático no tempo de execução relativamente a uma MT com uma só fita.

Teorema 5.1.7. *Seja $f : \mathbb{N} \to \mathbb{N}$ uma função satisfazendo $f(n) \geq n$ para todo o $n \in \mathbb{N}$. Se uma linguagem L é decidida em tempo $f(n)$ por uma máquina de Turing com k fitas, então L é decidida em tempo $O(f^2(n))$ por uma máquina de Turing com uma só fita.*

Demonstração. Antes de começar a demonstração, consideremos um exemplo. Suponhamos que temos uma MT com 3 fitas, cujo conteúdo das fitas é num dado momento

$$\ldots BB\underset{\triangle}{0}10110BB\ldots$$
$$\ldots BB\underset{\triangle}{B}BBB\ldots$$
$$\ldots BB\underset{\triangle}{1}1111BB\ldots$$

(o triângulo indica o que a cabeça de leitura está a ler). Então o conteúdo da primeira fita pode ser codificado como $01\blacklozenge0110$ (os símbolos à esquerda do $\blacklozenge$ são os símbolos não-brancos à esquerda da cabeça de leitura, a palavra 0110 é o que está na posição da cabeça de leitura, inclusive, até à sua direita, não se contando a porção infinita de brancos que se lhe segue). Então o conteúdo das 3 fitas, assim como a posição das respetivas cabeças de leitura, pode ser registado numa única palavra

$$\#01\blacklozenge0110\#\blacklozenge\#11\blacklozenge111\#.$$

Utilizando um procedimento semelhante para uma MT M com k fitas, podemos registar o conteúdo das k fitas, assim como a posição das respetivas cabeças de leitura, utilizando uma única fita. Assim podemos criar uma MT M_1, com uma única fita, que simula M. A MT simula M passo a passo da seguinte forma:

1. Lê o conteúdo da sua fita para determinar quais os símbolos lidos nas k fitas de M_1;
2. Com base nesses símbolos e no estado atual, atualiza o estado e o conteúdo das k fitas (ou melhor dizendo, a sua codificação numa única fita).

Note-se que às vezes vamos ter de "empurrar" (fazer um shift) quando efetuamos os movimentos virtuais das cabeças das fitas de M. Por exemplo, se na configuração

$$\#\blacklozenge010110\#\ldots\#$$

o próximo passo será manter o símbolo 0 atualmente lido na fita 1 e deslocar a cabeça de leitura uma casa para a esquerda, o conteúdo virtual resultante das 3 fitas será codificado como

$$\#\blacklozenge B010110\# \ldots \#.$$

Por outras palavras, fizemos um "shift" no conteúdo da fita de M_1.

Vamos agora ver quanto tempo se demora a simular M com a máquina M_1 para um *input* de tamanho n. Como M utiliza tempo $f(n) \geq n$, cada uma das fitas de M nunca vai ter mais de $f(n)$ símbolos (mais precisamente $O(f(n))$). Logo a descrição das k fitas de M na única fita de M_1 precisa de $k \times O(f(n)) \in O(f(n))$ símbolos.

A simulação de um passo de M pela MT M_1 necessita de $2O(f(n))$ passos (ler a fita e voltar atrás – passo 1 indicado atrás) + $k \times O(f(n))$ (para atualizar o conteúdo da fita. Pode haver até k shifts, necessitando cada um deles de tempo $O(f(n))$ para ser executado). Resumindo, cada passo de M pode ser executado em tempo $O(f(n))$ por M_1.

Mas, com *input* de tamanho n, M utiliza no máximo $O(f(n))$ passos de computação. Logo M_1 necessita, no total da simulação, de tempo $O(f(n)) \times O(f(n)) = O(f^2(n))$. □

Por exemplo, já sabemos que a MT da Fig. 4.9, que utiliza duas fitas, decide a linguagem $L_2 = \{0^k 1^k \in \{0,1\}^* | k \geq 1\}$ em tempo $O(n)$. Então o teorema anterior permite-nos concluir que a linguagem L_2 pode ser decidida por uma MT com uma só fita em tempo $O(n^2)$.

5.2 As classes P e NP

A partir das definições da secção anterior, podemos definir algumas classes de complexidade.

Definição 5.2.1. Definimos as seguintes classes

$$P = \bigcup_{k \in \mathbb{N}} TIME(n^k)$$
$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$
$$EXPTIME = \bigcup_{k \in \mathbb{N}} TIME(2^{n^k})$$

Por outras palavras, a classe P (NP, respetivamente) é a classe das linguagens decididas por uma MT (MTND, respetivamente) em tempo polinomial. A classe $EXPTIME$ é a classe das linguagens decididas por uma MT em tempo exponencial. Note-se que, na definição de P ou de $EXPTIME$, não importa se utilizamos uma MT com uma só fita, ou com várias fitas, devido ao Teorema 5.1.7.

Como exemplos,

$$L_1 = \{0^k 1^k 0^k \in \{0,1\}^* | k \in \mathbb{N}\} \in P$$
$$L_2 = \{0^k 1^k \in \{0,1\}^* | k \geq 1\} \in P$$
$$L_3 = \{0^{2k} \in \{0\}^* | k \geq 1\} \cup \{0^{3k} \in \{0\}^* | k \geq 1\} \in NP$$
$$L_4 = \{0, 1, 01, 10\} \in NP$$

pois L_1 pode ser decidida em tempo $O(n^2)$ pela MT da Fig. 4.1, L_2 pode ser decidida em tempo $O(n)$ pela MT da Fig. 4.9, L_3 pode ser decidida em tempo $O(n)$ pela MTND da Fig. 4.11, e L_4 pode ser decidida em tempo $O(1)$ pela MTND da Fig. 4.11. A experiência mostra que os problemas que podem ser resolvidos em tempo razoável são aqueles que estão em P. É claro que um problema que necessita de tempo $n^{1000000}$ para ser resolvido não é resolúvel na prática (para n grande), mas normalmente os problemas "úteis" em P podem ser resolvidos em tempo com expoentes baixos, tipicamente n ou n^2.

Que relações existem entre estas classes? Sabe-se que

$$P \subseteq NP \subseteq EXPTIME.$$

A relação $P \subseteq NP$ é imediata do Teorema 5.1.5, e a inclusão $NP \subseteq EXPTIME$ vem do Teorema 5.1.6 (note-se que $2^{O(n^k)} \subseteq O(2^{n^{k+1}})$). Sabe-se ainda que $P \neq EXPTIME$, mas não se conhece mais nenhuma relação entre estas 3 classes. Por exemplo, não se sabe se $NP = EXPTIME$ ou não, nem se $P = NP$ ou não.

Em particular, o problema "$P = NP$?" tem sido bastante investigado devido às várias implicações que tem em domínios com bastante interesse para a Informática (criptografia, obtenção de algoritmos eficientes para a resolução de problemas importantes, etc.), Matemática, etc.[1]

Problema para 1 milhão de dólares: $P = NP$?

Porquê que este problema é importante? Para isso podemos adotar a seguinte visão, utilizando uma nova caracterização de NP que iremos apresentar de seguida, onde por problema de decisão se entende um problema onde, dado um *input* w, se pretende uma resposta do tipo *sim/não* (este tipo de problema pode ser sempre reduzido a um problema do tipo "$w \in A$?", onde A é uma linguagem):

- P = classe dos problemas de decisão cuja solução pode ser **obtida** em tempo útil;
- NP = classe dos problemas de decisão cuja solução pode ser **verificada** em tempo útil;

[1]Existe um prémio de 1 milhão de dólares para quem resolver este problema: http://www.claymath.org/millennium-problems/p-vs-np-problem

onde por tempo útil se entende tempo polinomial. A diferença entre obter uma solução e verificá-la pode parecer subtil, mas é fundamental. Por exemplo, num sistema criptográfico, queremos que uma solução (*password*) possa ser verificada em tempo útil, mas não queremos que possa ser obtida em tempo útil por um *hacker*. Ou seja, na sua essência, o problema "$P = NP$?" resume-se a saber se:

É mais díficil encontrar soluções do que verificá-las?

Acredita-se que $P \neq NP$ devido à experiência que temos que, em muitos contextos, é fácil verificar uma solução, mas é quase impossível obtê-la sem ajuda (é quase impossível encontrar uma agulha num palheiro, mas essa tarefa é muito mais fácil se alguém nos disser onde ela está).

Por exemplo, dado um grafo não-orientado G e dois vértices v, w de G, será que G admite uma caminho hamiltoneano (i.e. um caminho que passa exatamente uma vez por cada vértice de G) que começa no vértice v e acaba no vértice w? Se alguém nos der uma solução para este problema, esta é fácil de verificar. No entanto, se pretendermos determinar essa solução, não se conhece algoritmos que sejam muito mais eficientes do que o algoritmo de "força bruta". Neste tipo de algoritmo todos os possíveis caminhos (em número de $k!$, onde k é o número de vértices do grafo) são explorados, sendo por isso um algoritmo altamente ineficiente para grafos com mais do que algumas dezenas de vértices.

Outros exemplos (que também poderiam ser codificados em problemas de decisão) em que é fácil verificar uma solução, mas pode ser muitíssimo difícil obtê-la, incluem o caso da Matemática onde, dado um teorema, é em geral muito mais fácil verificar se uma demonstração deste teorema é correta (verificação de solução) do que obtê-la (se $P = NP$, então verificar uma solução daria essencialmente o mesmo trabalho do que obtê-la, o que alguns autores denominam de "morte da criatividade"). Também se pode incluir o caso da Física onde, dado um conjunto de dados experimentais, se pretende saber se existe uma teoria que se encaixa nesses dados. Encontrar a teoria é difícil mas, se nos derem uma teoria, é bem mais fácil verificar se a teoria é consistente com os dados. Mais informação acerca destes exemplos pode ser encontrada no livro [MM11, páginas 179–180].

Vamos agora apresentar uma caracterização alternativa de NP, que normalmente é mais útil na prática.

Teorema 5.2.2. *$L \in NP$ se e só se existe uma máquina de Turing M, cujo tempo de execução é polinomial (em w), tal que*

$$L = \{w \in \Sigma^* | M \text{ aceita } \langle w, c\rangle \text{ para algum } c \in \Sigma^*\}. \tag{5.1}$$

Demonstração. Demonstração da direção $\Longrightarrow$ do "se e só se". Se $L \in NP$, então existe uma MTND N que aceita L. Podemos criar uma MT M que recebe como *input* $\langle w, c\rangle$,

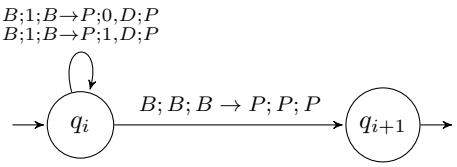


Figura 5.1: MTND que gera de forma não-determinística uma palavra em $\{0,1\}$ de comprimento n^k na 2ª fita, dada uma sequência de n^k 1's na segunda fita.

e que basicamente simula N, com *input* w, para o caminho de computação indicado por c (ver a demonstração do Teorema 4.3.2 e os comentários que se lhe seguem). Se esse caminho aceitar, M aceita $\langle w, c\rangle$, caso contrário rejeita $\langle w, c\rangle$. Ora $w \in L$ sse existe uma computação de N que aceita w sse M aceita $\langle w, c\rangle$ para algum $c \in \Sigma^*$. Por outras palavras, a relação (5.1) é satisfeita.

Para mostrar a implicação inversa, suponhamos que (5.1) é satisfeita. Suponhamos ainda que M corre em tempo n^k, onde $n = |w|$. Então M nunca vai ler mais do que n^k símbolos de c. Portanto, podemos supor que $|c| \leq n^k$. Iremos ainda supor que $\Sigma = \{0,1\}$, embora o procedimento seja idêntico para outros alfabetos Σ. Então podemos criar uma MTND N, com três fitas, que faz o seguinte (com *input* w):

1. Escrever na 2ª fita uma sequência consecutiva de 1's com comprimento n^k, delimitada em ambas as extremidades por um B, e posicionar a respetiva cabeça de leitura no 1 mais à esquerda. Por exemplo, se $k = 2$, então a Fig. 4.10 mostra um exemplo de MT que computa n^2. Em geral isso demora tempo $O(n^k)$.

2. Se estiver a ler um 1 na 2ª fita e um B na 3ª fita, escrever não-deterministicamente um 0 ou um 1 na 3ª fita e mover as cabeças de leitura da 2ª e 3ª fita para a direita, repetindo este passo até se chegar a um branco na 2ª fita. Isso pode ser feito, por exemplo, pela MT da Fig. 5.1 em tempo n^k (esta máquina começa a computação em q_i e começa a partir do resultado obtido no passo 1, assumindo que a cabeça de leitura da 1ª fita está a ler um B e que a terceira fita só tem brancos).

3. Simular a computação da MT M com *input* $\langle w, c\rangle$, onde $c \in \Sigma^*$ é a palavra de comprimento n^k gerada na fita 3 no passo anterior. (tempo n^k).

4. Se M aceitar $\langle w, c\rangle$, aceitar w, caso contrário rejeitar (tempo 1).

Note-se que cada possível palavra $c \in \Sigma^*$ de comprimento n^k irá ser gerada num dos ramos de computação do passo 2. Nesse caso diz-se que a MTND gerou não-deterministicamente a palavra $c \in \Sigma^*$.

Portanto N corre em tempo $O(n^k)$ e w é aceite por N sse existe um caminho de computação aceitador sse existe um $c \in \Sigma^*$ de comprimento n^k tal que M aceita $\langle w, c\rangle$ sse $w \in L$. Logo, N decide L em tempo polinomial, donde $L \in NP$. □

Vamos agora ver alguns exemplos de como mostrar que um problema pertence a P ou a NP, neste último caso utilizando a caracterização através de certificados. Regra geral, não vamos apresentar explicitamente a MT (isso só é viável para casos simples. Em termos de linguagens de programação corresponde a programar em código de máquina, o que não é viável na maioria das tarefas), mas vamos antes descrevê-la através de um algoritmo, utilizando a Tese de Church-Turing. Vamos também fazer as seguintes assumpções sobre o custo computacional de operações aritméticas simples:

- Adição: dados dois *inputs* de tamanho $\leq n$, a sua adição pode ser calculada em tempo $O(n)$.
- Multiplicação, divisão inteira (`div`), resto da divisão inteira (`mod`): dados dois *inputs* de tamanho $\leq n$, estas operações podem ser calculadas em tempo $O(n^2)$.

(assumimos que os números são escritos em notação unária, ou em base $d \geq 2$. Os tempos das operações podem ser obtidos em qualquer livro de Análise Numérica, ou inspecionando o tempo necessário para executar os cálculos "à mão"). Tipicamente, iremos codificar números inteiros em binário, pelo que o tamanho de um *input* $\langle n \rangle$, onde $n \in \mathbb{N}$, será da ordem de $\log_2 n$.

Exemplo 5.2.3. Mostre que

$$UNARY - RELPRIME = \{1^x \# 1^y : x, y \in \mathbb{N} \text{ são primos entre si}\} \in P.$$

Resolução. Recordamos que $x, y \in \mathbb{N}$ são primos entre si se $mdc(x, y) = 1$. Inicialmente verificamos que o *input* tem o formato pretendido. Isso pode facilmente ser feito em tempo $O(n)$ por uma MT que determine $mdc(x, y)$, utilizando o algoritmo de Euclides (Algoritmo 4).

Note-se o interior de cada ciclo While pode ser executado em tempo $O(n^2)$ e, no máximo, vai haver $\min(x, y) + 1$ ciclos, pelo que o algoritmo para calcular MDC(x,y) corre em tempo polinomial, da ordem de $O(n^3)$, o mesmo acontecendo para a função UnaryRelprime. No final, basta testar se $mdc(x, y) = 1$, o que também pode ser feito em tempo polinomial, da ordem de $O(1)$. Logo este algoritmo permite-nos decidir se uma palavra pertence a $UNARY - RELPRIME$ em tempo $O(n^3)$, pelo que $UNARY - RELPRIME \in P$.

Exemplo 5.2.4. Mostre que

$$NOTPRIMES = \{\langle x \rangle : x \in \mathbb{N} \text{ não é um número primo}\} \in NP.$$

Resolução. Recordemos que um número x é primo se é um inteiro ≥ 2 que só é divisível por 1 e por ele mesmo. Também assumimos, como indicado atrás, que $\langle x \rangle$ é o número x codificado em binário. Uma estratégia para determinar se um número não é primo é dividi-lo por todos os inteiros entre 2 e x. No entanto, o problema é que isto custa demasiado tempo. Como o tamanho do *input* n é da ordem de $\log_2 x$, e como há $x - 2$

Algoritmo 4 Programa para verificar se $1^x\#1^y \in UNARY - RELPRIME$

Require: $1^x\#1^y \in \{1, \#\}$

```
 1: function MDC(x, y)
 2:     while y ≠ 0 do
 3:         aux = x mod y
 4:         x = y
 5:         y = aux
 6:     end while
 7:     return x
 8: end function
 9: function UnaryRelprime(1^x#1^y)
10:     if o input não tem o formato desejado then
11:         return 0                                   ▷ Rejeita
12:     end if
13:     if MDC(x,y)=1 then
14:         return 1                                   ▷ Aceita
15:     else
16:         return 0                                   ▷ Rejeita
17:     end if
18: end function
```

divisores, isso teria um custo exponencial em n (poderíamos só dividir pelos números até $\sqrt{x}$, mas mesmo assim o custo continuaria a ser exponencial). Portanto determinar uma solução para este problema parece ser demasiado custoso. Mas verificar uma solução não o é – se nos derem um inteiro k entre 2 e $x - 1$ que divida x podemos verificar em tempo polinomial que k divide x, permitindo-nos verificar que $x \in NOTPRIMES$.

Inicialmente verificamos que o *input* tem o formato pretendido. Como assumimos que codificação $\langle\cdot\rangle$ é "razoável", esta verificação pode ser feita em tempo polinomial. Por esta razão este passo é normalmente omitido (o tempo exato depende da codificação utilizada).

Para mostrar que $NOTPRIMES \in NP$, vamos utilizar a caracterização do Teorema 5.2.2. Isso pode ser feito com o Algoritmo 5.

Mais uma vez assumimos que se pode verificar em tempo polinomial se c não codifica ou não um número inteiro k (assumimos sempre que a codificação é "razoável", como a codificação binária). Todos os restantes passos do algoritmo podem ser realizados em tempo $O(n^2)$. Logo uma solução pode ser verificada em tempo polinomial, pelo que $NOTPRIMES \in NP$.

Vamos agora considerar exemplos de problemas com grafos, onde o *input* não será um grafo G, mas a sua codificação $\langle G\rangle$ utilizando, por exemplo, as codificações dadas no final da Secção 4.4. Note-se que se G tem k vértices, então para as codificações indicadas atrás, tem-se $k \leq |\langle G\rangle| \leq k^3$. Portanto, se o nosso *input* for um grafo G com k

Algoritmo 5 Programa para verificar se x não é primo

Require: $x \in \mathbb{N}$
Require: $c \in \{0,1\}^*$ ▷ Certificado
1: **function** NotPrime(x, c)
2: **if** c não codifica um número inteiro $0 < k < x$ **then**
3: **return** 0 ▷ Rejeita
4: **else**
5: **if** $x \; mod \; k = 0$ **then**
6: **return** 1 ▷ Aceita
7: **else**
8: **return** 0 ▷ Rejeita
9: **end if**
10: **end if**
11: **end function**

vértices, e se mostrarmos que existe um algoritmo que é executado em tempo polinomial no número de vértices de G, então esse algoritmo também será executado em tempo polinomial relativamente ao tamanho n do input.

Exemplo 5.2.5. Mostre que

$$TREE = \{\langle G\rangle : G \text{ é um grafo não-orientado e é uma árvore}\} \in P.$$

Resolução. Recordemos que um grafo conexo é uma árvore se não tem ciclos. Além do mais, um grafo com k vértices é uma árvore sse é um grafo conexo com exatamente $k - 1$ arestas.

Então, utilizando esta última caracterização, podemos utilizar o Algoritmo 6 para verificar se $\langle G\rangle$ pertence a P:

As linhas 5–19 testam se o grafo é conexo. Basicamente a estratégia é começar do vértice v_1 e verificar recursivamente que vértices lhe estão ligados. Note-se que, no pior dos casos, será adicionado um só novo vértice ao conjunto *new* em cada iteração do ciclo *While*, pelo que este ciclo será iterado, no máximo, $k \leq n$ vezes. Assim, os comandos das linhas 9 a 11 não serão iterados mais do que n vezes. Além do mais, os conjuntos $marked$ e $notmarked$ têm no máximo $k \leq n$ vértices, pelo que os comandos das linhas 14 a 16 não vão ser iterados mais do que n^2 vezes por cada iteração do ciclo *While*, num total limitado por $n \times n^2 = n^3$ vezes. Por sua vez, os comandos das linhas 14 a 15 demoram tempo $O(n)$ a serem executados por cada iteração. Portanto este algoritmo pode ser executado em tempo polinomial, nomeadamente da ordem $O(n^4)$, donde $TREE \in P$

Note-se que no exemplo anterior assumimos novamente, de forma implícita, que $\langle G\rangle$ é uma codificação "razoável" de um grafo, podendo-se verificar em tempo polinomial se a codificação é válida e podendo-se também obter em tempo polinomial informação sobre os vértices, arestas, etc., a partir de $\langle G\rangle$.

Algoritmo 6 Programa para verificar se um grafo G é uma árvore

Require: $\langle G \rangle$, onde $G = (V, A)$ e $V = \{v_1, \ldots, v_k\}$ é o conjunto dos vértices

```
 1: function TREE(⟨G⟩)
 2:     if numero de arestas de G ≠ k − 1 then
 3:         return 0                                          ▷ Rejeita
 4:     else                                  ▷ Verificar se o grafo é conexo
 5:         marked = ∅
 6:         notMarked = V
 7:         new = {v_1}
 8:         while new ≠ ∅ do
 9:             marked = marked ∪ new
10:             notMarked = V \ marked
11:             new = ∅
12:             for each v ∈ marked do
13:                 for each w ∈ notMarked do
14:                     if {v, w} é uma aresta de G then
15:                         new = new ∪ {w}
16:                     end if
17:                 end for
18:             end for
19:         end while
20:         if marked = V then
21:             return 1                                      ▷ Aceita
22:         else
23:             return 0                                      ▷ Rejeita
24:         end if
25:     end if
26: end function
```

Exemplo 5.2.6. Mostre que

$$HAMPATH = \{\langle G, s, t\rangle \text{ onde } G \text{ é um grafo e } s, t \text{ são vértices} \mid \text{existe um caminho hamiltoneano de } s \text{ a } t\} \in NP, \tag{5.2}$$

Resolução. Para mostrar que $HAMPATH \in NP$, vamos utilizar a caracterização do Teorema 5.2.2. Isso pode ser feito com o Algoritmo 7.

Algoritmo 7 Programa para verificar se uma grafo admite um caminho hamiltoneano

Require: $\langle G\rangle$, onde $G = (V, A)$ e $V = \{v_1, \ldots, v_k\}$ é o conjunto dos vértices
Require: s, t, onde $s, t \in V$
Require: $c \in \{0,1\}^*$ ▷ Certificado

```
1:  function TREE(⟨G, s, t⟩)
2:      if c codifica uma sequência w_1, w_2, ..., w_k de k vértices distintos de G then
3:          if w_1 ≠ s ou w_k ≠ t then
4:              return 0                                          ▷ Rejeita
5:          end if
6:          for i = 1 to k − 1 do
7:              if {w_i, w_{i+1}} não é aresta de G then
8:                  return 0                                      ▷ Rejeita
9:              end if
10:         end for
11:         return 1                                              ▷ Aceita
12:     else
13:         return 0                                              ▷ Rejeita
14:     end if
15: end function
```

O passo 2 pode ser executado em tempo $O(n^2)$, já que é necessário ler o *input* uma vez por cada vértice que aparece na descrição de c, para ver se ele pertence a G, e no máximo há $k \leq n$ vértices. Depois de verificar que c contém k vértices de G, é ainda necessário verificar que os vértices são distintos entre si. Isso pode ser feito selecionando um vértice em c e comparando-o com os restantes (tempo $O(n)$). Como esse procedimento tem de ser efetuado $k \leq n$ vezes, no total demora tempo $O(n^2)$.

O ciclo *For* verifica se c codifica de facto um caminho de s a t. As instruções das linhas 3–5 e 7–9 demoram tempo $O(n)$ a serem executadas, e o ciclo *For* é iterado $k-1 \leq n$ vezes, pelo que concluímos que este algoritmo verifica se o certificado é válido em tempo $O(n^2)$, i.e. em tempo polinomial, pelo que $HAMPATH \in NP$.

5.3 Problemas NP-completos

Em 1971 Stephen Cook introduziu a noção de problemas NP-completos como ferramenta para abordar o problema "$P = NP$?" Esta noção é importante, não só pela

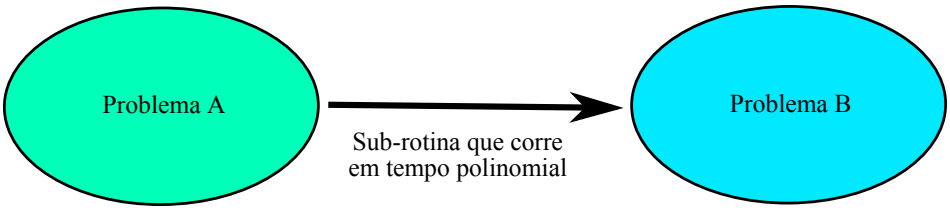


Figura 5.2: Redutibilidade de linguagens.

contribuição que poderá ter na resolução do problema "$P = NP$?", mas também porque permite concluir que certos problemas em NP são especialmente difíceis. Antes de introduzir esta noção, precisamos de algumas definições.

Definição 5.3.1. Uma função $f : \Sigma^* \to \Sigma^*$ diz-se *computável em tempo polinomial* se existe uma máquina de Turing que computa f em tempo polinomial (i.e. o tempo de execução da MT é $O(n^k)$ para algum $k \in \mathbb{N}$).

Por exemplo, a função $f : \{0\}^* \to \{0\}^*$ dada por $f(0^n) = 0^{2n}$ pode ser calculada em tempo $O(n^2)$ pela MT da Fig. 4.5. Logo f é computável em tempo polinomial.

Definição 5.3.2. Uma linguagem A é *redutível em tempo polinomial* a uma linguagem B, escrito como $A \leq_P B$, se existe uma função $f : \Sigma^* \to \Sigma^*$ computável em tempo polinomial, tal que para todo o $w \in \Sigma^*$,

$$w \in A \quad \text{se e só se} \quad f(w) \in B.$$

Por outras palavras, existe um sub-rotina que permite converter, em tempo polinomial, o problema "$x \in A$?" para o problema "$x \in B$?", como sugere a Fig. 5.2. Assim, se for fácil decidir a linguagem B, também será fácil decidir A, utilizando a sub-rotina de conversão. Vamos agora ver várias consequências da redutibilidade em tempo polinomial.

Teorema 5.3.3. *Se $A \leq_P B$ e $B \in P$, então $A \in P$.*

Demonstração. Seja M a MT que decide B em tempo polinomial e seja a f a redução polinomial de A para B. Então o seguinte algoritmo decide A em tempo polinomial:
1. Para um *input* w calcule $f(w)$;
2. De seguida utilize $f(w)$ como *input* para B. Se B aceita $f(w)$, aceitar w, caso contrário rejeitar w. □

Definição 5.3.4. Uma linguagem B é *NP-completa* se satisfaz as seguintes definições:

1. $B \in NP$;
2. Todo o $A \in NP$ é redutível em tempo polinomial Ba B.

Teorema 5.3.5. *Se B é NP-completo e $B \in P$, então $P = NP$.*

Demonstração. Imediata a partir do Teorema 5.3.3. □

Isto mostra que se conseguirmos arranjar um algoritmo que resolva um problema NP-completo em tempo polinomial, então mostramos que $P = NP$. Como é normalmente conjeturado que $P \neq NP$, esse resultado pode ser visto de outra forma: os problemas de tipo NP-completo são os mais "difíceis" da classe NP e, se nos depararmos com um, é muito provável que não exista um algoritmo que o resolva em tempo útil (polinomial) pelo que, na prática, teremos de recorrer a heurísticas ou algoritmos probabilísticos. Mas como identificar problemas NP-completos? Os seguintes dois teoremas dão-nos informações valiosas.

Teorema 5.3.6. *Seja $\mathcal{F}_V$ o conjunto das fórmulas do cálculo proposicional. Então a seguinte linguagem é NP-completa*

$$SAT = \{\langle \phi \rangle \text{ onde } \phi \in \mathcal{F}_V \mid \phi \text{ é satisfazível}\}.$$

A linguagem SAT foi a primeira linguagem que se demonstrou ser NP-completa. A partir deste conhecimento, torna-se muito mais fácil mostrar que outras linguagens são NP-completas, através do seguinte teorema.

Teorema 5.3.7. *Se B é NP-completo e $B \leq_P C$, onde $C \in NP$, então C é NP-completo.*

Demonstração. Seja $A \in NP$. Como B é NP-completo, $A \leq_P B$. Mas como $B \leq_P C$, conclui-se que $A \leq_P C$. Além do mais $C \in NP$, donde C será NP-completo. □

Em particular, utilizando o teorema anterior em conjugação com a linguagem SAT, é possível mostrar que a linguagem $HAMPATH$ definida pela Equação (5.2) é NP-completa (ver, por exemplo, [Sip12]). Na realidade, existem inúmeras linguagens NP-completas, muitas envolvendo problemas relevantes na prática. Por exemplo, o livro [GJ79] tem mais de 300 linguagens NP-completas.

Finalmente introduzimos uma definição que é muitas vezes utilizada em aplicações onde se pretende classificar o "grau de dificuldade" de um problema.

Definição 5.3.8. Uma linguagem L diz-se *NP-difícil* (*NP-hard* em inglês), se toda a linguagem $A \in NP$ é redutível em tempo polinomial a L (a diferença em relação a linguagens NP-completas é que L não tem necessariamente de pertencer a NP).

Em particular, toda a linguagem NP-completa é NP-difícil. Na Fig. 5.3 é apresentado um diagrama que relaciona as principais classes de linguagens estudadas nesta disciplina. Há um resultado que não foi demonstrado nestes apontamentos, mas que se verifica: classe das linguagens r.e. $\subsetneq$ classe de todas as linguagens (pois o complemento da linguagem $H_{paragem}$ define uma linguagem que não é r.e. – ver exercício 110 das folhas teórico-práticas). A fronteira entre P e NP está a tracejado, uma vez que não se sabe se $P \neq NP$, embora se conjeture que estas classes são diferentes.

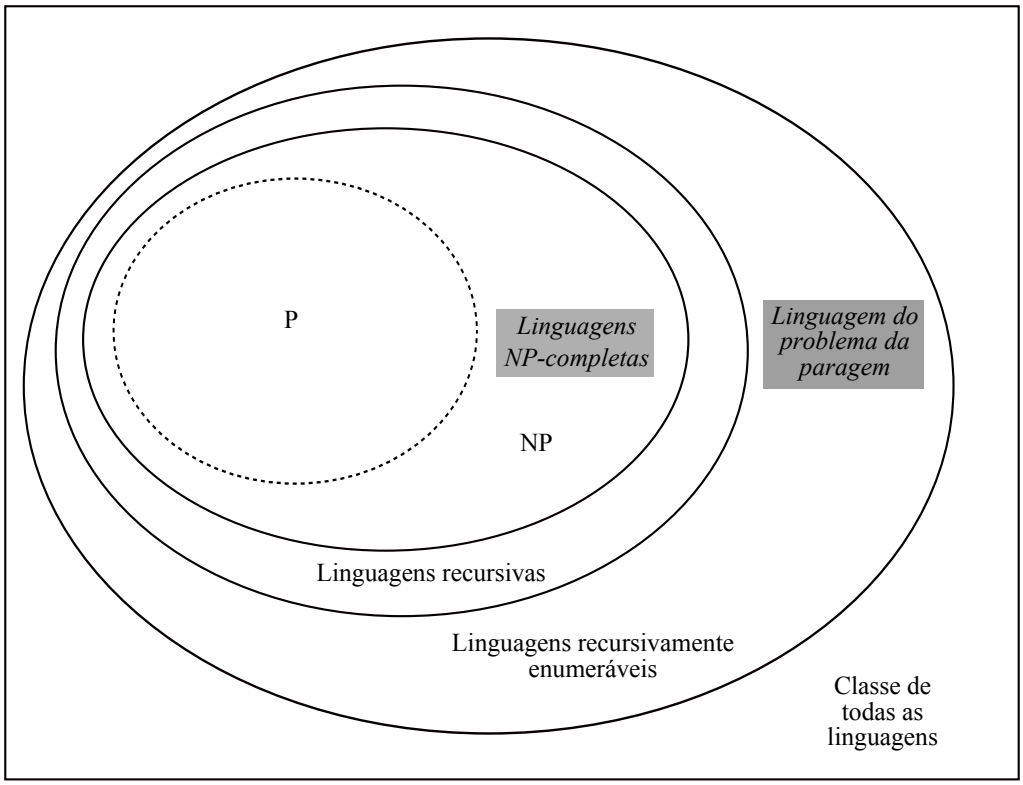


Figura 5.3: Relação entre as principais linguagens estudadas nesta disciplina, assumindo que as linguagens são tomadas sobre um alfabeto Σ.

Bibliografia

Lógica

[Ben12] M. Ben-Ari. *Mathematical Logic for Computer Science*. 3rd edition. Springer, 2012.

[Gor16] V. Goranko. *Logic as a Tool: A Guide to Formal Logical Reasoning*. Wiley, 2016.

[HR04] M. Huth e M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2nd edition. Cambridge University Press, 2004.

Teoria da Computação

[CG19] J. F. Costa e P. Gouveia. *Matemática Discreta*. IST Press, 2019.

[GJ79] M. R. Garey e D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[HMU06] J. E. Hopcroft, R. Motwani e J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd edition. Addison-Wesley, 2006.

[MM11] C. Moore e S. Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[Sip12] M. Sipser. *Introduction to the Theory of Computation*. 3rd edition. Cengage Learning, 2012.