

Introdução à Programação com AutoLisp

António Menezes Leitão

Fevereiro 2007

Conteúdo

1	Introdução	2
1.1	Linguagens de Programação	5
1.2	Lisp	6
1.3	Exercícios	7
2	A Linguagem Auto Lisp	7
2.1	Sintaxe, Semântica e Pragmática	7
2.2	Sintaxe e Semântica do Lisp	8
2.3	O Avaliador	9
2.4	Elementos da Linguagem	9
2.5	Números	10
2.6	Combinações	11
2.7	Indentação	13
2.8	Avaliação de Combinações	15
2.9	Cadeias de Caracteres	15
2.10	Operadores de <i>Strings</i>	16
2.11	Definição de Funções	17
2.12	Símbolos	20
2.13	Funções de Múltiplos Parâmetros	22
2.14	Encadeamento de Funções	22
2.15	Funções Pré-Definidas	23
2.16	Aritmética de Inteiros em Auto Lisp	27
2.17	Aritmética de Reais em Auto Lisp	30
2.18	Avaliação de Símbolos	32
2.19	Símbolos como Valores	33
2.20	Expressões Condicionais	35
2.21	Expressões Lógicas	35
2.22	Valores Lógicos	36
2.23	Predicados	36
2.24	Predicados Aritméticos	36
2.25	Operadores Lógicos	37
2.26	Predicados com número variável de argumentos	37
2.27	Predicados sobre Cadeias de Caracteres	38
2.28	Reconhecedores	38
2.29	Reconhecedores Universais	39
2.30	Exercícios	40
2.31	Selecção	41
2.32	Selecção Múltipla—A Forma <i>cond</i>	43

3 Modelação Geométrica	44
3.1 Coordenadas	44
3.2 Listas	45
3.3 Operações com Coordenadas	46
3.4 Abstracção de Dados	47
3.5 Tipos Abstractos	50
3.6 Coordenadas Bidimensionais	52
3.7 Coordenadas em AutoCad	53
3.8 Coordenadas Polares	54
3.9 A função command	56
3.10 Variantes de Comandos	62
3.11 Ângulos em Comandos	63
3.12 Efeitos Secundários	65
3.13 Sequenciação	65
3.14 A Ordem Dórica	68
3.15 Parametrização de Figuras Geométricas	72
3.16 Documentação	75
3.17 Depuração	79
3.17.1 Erros Sintáticos	79
3.17.2 Erros Semânticos	81
3.18 Modelação Tridimensional	83
3.18.1 Sólidos Tridimensionais Pré-Definidos	83
3.18.2 Coordenadas Cilíndricas	88
3.18.3 Coordenadas Esféricas	90
3.18.4 Modelação de Colunas Dóricas	92
4 Funções	94
4.1 Variáveis Locais	94
4.1.1 Declaração	96
4.1.2 Atribuição	96
4.2 Variáveis Globais	97
4.3 Variáveis Indefinidas	99
4.4 Proporções de Vitrúvio	99
4.5 Recursão	106
4.6 Recursão em Arquitectura	111
4.7 Depuração de Programas Recursivos	115
4.7.1 <i>Trace</i>	116
4.8 Templos Dóricos	120
4.9 A Ordem Jónica	131
4.10 Recursão na Natureza	138

5 Atribuição	143
5.1 Aleatoriedade	143
5.2 Números Aleatórios	144
5.3 Estado	147
5.4 Estado Local e Estado Global	148
5.5 Escolhas Aleatórias	149
5.5.1 Números Aleatórios Fraccionários	152
5.5.2 Números Aleatórios num Intervalo	153
5.6 Planeamento Urbano	157
6 Listas	165
6.1 Pares	167
6.2 Representação Gráfica de Pares	168
6.3 Tipos Recursivos	169
6.4 Recursão em Listas	170
6.5 Predicados sobre Listas	174
6.6 Enumerações	176
6.7 Comparação de Listas	178
6.8 Comparação de Símbolos	180
6.9 Comparação Estrutural de Listas	181
7 Carregamento de Ficheiros	182
7.1 Carregamento de Dependências	183
7.2 Módulos	184
7.3 Carregamento Automático	189
7.4 Interpretação e Compilação	189
8 Listas de Coordenadas	191
8.1 Polígonos	191
8.1.1 Estrelas Regulares	192
8.1.2 Polígonos Regulares	196
8.2 Iteração em Listas	197
8.3 Linhas Poligonais e <i>Splines</i>	200
8.4 Treliças	204
8.4.1 Desenho de Treliças	205
8.4.2 Geração de Coordenadas	212
8.4.3 Treliças Espaciais	216
9 Manipulação de Entidades	221
9.1 Listas de Associações	223
9.2 A função <code>assoc</code>	225
9.3 Modificação de Entidades	226
9.4 Criação de Entidades	226
9.5 Listas de Propriedades	229

9.6	Leitura e Modificação de Entidades	236
9.7	Eliminação de Entidades	237
9.8	Última Entidade	237
9.9	Iteração de Entidades	238
9.10	Entidades Compostas	239
9.11	Redesenho de Entidades	241
9.12	Criação de Linhas Poligonais “Leves”	241
9.13	Manipulação de Linhas Poligonais “Leves”	244
9.14	Selecção Interactiva de Entidades	245
9.15	Definição de Comandos AutoCad	246
9.16	Obtenção de Informação	247
9.17	Áreas de Polígonos	247
9.18	Entidades com Informação Adicional	257
9.19	Escadas	260
9.19.1	Dimensionamento de Escadas	266
9.19.2	Dimensionamento de Lanços	276
9.19.3	“El Castillo”	277
9.20	Caixas de Escada	279

1 Introdução

A transmissão de conhecimento é um dos problemas que desde cedo preocupou a humanidade. Sendo o homem capaz de acumular conhecimento ao longo de toda a sua vida, é com desânimo que enfrenta a ideia de que, com a morte, todo esse conhecimento se perca.

Para evitar esta perda, a humanidade inventou toda uma série de mecanismos de transmissão de conhecimento. O primeiro, a transmissão oral, consiste na transmissão do conhecimento de uma pessoa para um grupo reduzido de outras pessoas, de certa forma transferindo o problema da perda de conhecimento para a geração seguinte. O segundo, a transmissão escrita, consiste em registar em documentos o conhecimento que se pretende transmitir. Esta forma tem a grande vantagem de, por um lado, poder chegar a muitas mais pessoas e, por outro, de reduzir significativamente o risco de se perder o conhecimento por problemas de transmissão. De facto, a palavra escrita permite preservar por muito tempo e sem qualquer tipo de adulteração o conhecimento que o autor pretendeu transmitir.

É graças à palavra escrita que hoje conseguimos compreender e acumular um vastíssimo conjunto de conhecimentos, muitos deles registados há milhares de anos atrás.

Infelizmente, nem sempre a palavra escrita conseguiu transmitir com rigor aquilo que o autor pretendia. A língua natural tem inúmeras ambiguidades e evolui substancialmente com o tempo, o que leva a que a interpretação dos textos seja sempre uma tarefa subjectiva. Quer quando escrevemos um texto, quer quando o lemos e o interpretamos, existem omissões, imprecisões, incorrecções e ambiguidades que podem tornar a transmissão de conhecimento falível. Se o conhecimento que se está a transmitir for simples, o receptor da informação, em geral, consegue ter a cultura e imaginação suficientes para conseguir ultrapassar os obstáculos. No caso da transmissão de conhecimentos mais complexos já isso poderá ser muito mais difícil.

Quando se exige rigor na transmissão de conhecimento, fazer depender a compreensão desse conhecimento da capacidade de interpretação de quem o recebe pode ter consequências desastrosas e, de facto, a história da humanidade está repleta de acontecimentos catastróficos cuja causa é, tão somente, uma insuficiente ou errónea transmissão de conhecimento, ou uma deficiente compreensão do conhecimento transmitido.

Para evitar estes problemas, inventaram-se linguagens mais rigorosas. A matemática, em particular, tem-se obsessivamente preocupado ao longo dos últimos milénios com a construção de uma linguagem onde o rigor seja absoluto. Isto permite que a transmissão do conhecimento matemático seja muito mais rigorosa que nas outras áreas, reduzindo ao mínimo essencial a capacidade de imaginação necessária de quem está a absorver esse conhecimento.

Para melhor percebermos do que estamos a falar, consideremos um caso concreto de transmissão de conhecimento, por exemplo, o cálculo do factorial de um número. Se assumirmos, como ponto de partida, que a pessoa a quem queremos transmitir esse conhecimento já sabe de antemão o que são os números e as operações aritméticas, podemos dizer-lhe que *para calcular o factorial de um número qualquer, terá de multiplicar todos os números desde a unidade até esse número*. Infelizmente, esta descrição é demasiado extensa e, pior, é pouco rigorosa, pois não dá ao ouvinte a informação de que os números que ele tem de multiplicar são apenas os números inteiros. Para evitar estas imprecisões e, simultaneamente, tornar mais compacta a informação a transmitir, a Matemática inventou todo um conjunto de símbolos e conceitos cujo significado deve ser compreendido por todos. Por exemplo, para indicar a sequência de números inteiros entre 1 e 9, a Matemática permite-nos escrever $1, 2, 3, \dots, 9$. Do mesmo modo, para evitarmos falar de “*um número qualquer*,” a Matemática inventou o conceito de *variável*: um nome que designa qualquer “coisa” e que pode ser reutilizado em várias partes de uma afirmação matemática com o significado óbvio de representar sempre essa mesma “coisa.” Deste modo, a linguagem Matemática permite-nos formular a mesma afirmação sobre o cálculo do factorial nos seguintes termos:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

Será a definição anterior suficientemente rigorosa? Será possível interpretá-la sem necessitar de imaginar a intenção do autor? Aparentemente, sim mas, na verdade, há um detalhe da definição que *exige* imaginação: as reticências. Aquelas reticências indicam ao leitor que ele terá de imaginar o que deveria estar no lugar delas. Embora a maioria dos leitores imagine correctamente que o autor pretendia a multiplicação dos sucessores dos números anteriores, leitores haverá cuja imaginação delirante poderá levá-los a tentar substituir aquelas reticências por outra coisa qualquer.

Mesmo que excluamos do nosso público-alvo as pessoas de imaginação delirante, há ainda outros problemas com a definição anterior. Pensem, por exemplo, no factorial de 2. Qual será o seu valor? Se substituirmos na fórmula, para $n = 2$ obtemos:

$$2! = 1 \times 2 \times 3 \times \cdots \times 2$$

Neste caso, o cálculo deixa de fazer sentido, o que mostra que, na verdade, a imaginação necessária para a interpretação da fórmula não se resstringe apenas às reticências mas sim a toda a fórmula: o número de termos a considerar depende do número do qual queremos saber o factorial.

Admitindo que o nosso leitor teria tido a imaginação suficiente para descobrir esse detalhe, ele conseguiria tranquilamente calcular que $2! = 1 \times 2 = 2$. Ainda assim, casos haverá que o deixarão significativamente menos tranquilo. Por exemplo, qual é o factorial de zero? A resposta não

parece óbvia. E quanto ao factorial de -1 ? Novamente, não está claro. E quanto ao factorial de 4.5 ? Mais uma vez, a fórmula nada diz e a nossa imaginação também não consegue adivinhar.

Será possível encontrar uma forma de transmitir o conhecimento da função factorial que minimize as imprecisões, lacunas e ambiguidades? Experimentemos a seguinte variante da definição da função factorial:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{se } n \in \mathbb{N}. \end{cases}$$

Teremos agora atingido o rigor suficiente que dispensa imaginação por parte do leitor? A resposta estará na análise dos casos que causaram problemas à definição anterior. Em primeiro lugar, não existem reticências, o que é positivo. Em segundo lugar, para o factorial de 2 temos, pela definição:

$$2! = 2 \times 1! = 2 \times (1 \times 0!) = 2 \times (1 \times 1) = 2 \times 1 = 2$$

ou seja, não há qualquer ambiguidade. Finalmente, vemos que não faz sentido determinar o factorial de -1 ou de 4.5 pois a definição apresentada apenas se aplica aos membros de \mathbb{N}_0 .

Este exemplo mostra que, mesmo na matemática, há diferentes graus de rigor nas diferentes formas como se expõe o conhecimento. Algumas dessas formas exigem um pouco mais de imaginação e outras um pouco menos mas, em geral, qualquer delas tem sido suficiente para que a humanidade tenha conseguido preservar o conhecimento adquirido ao longo da sua história.

Acontece que, actualmente, a humanidade conta com um parceiro que tem dado uma contribuição gigantesca para o seu progresso: o *computador*. Esta máquina tem a extraordinária capacidade de poder ser instruída de forma a saber executar um conjunto complexo de tarefas. A actividade da *programação* consiste, precisamente, na transmissão, a um computador, do conhecimento necessário para resolver um determinado problema. Esse conhecimento é denominado um *programa*. Por serem *programáveis*, os computadores têm sido usados para os mais variados fins e, sobretudo nas últimas décadas, têm transformado radicalmente a forma como trabalhamos. Infelizmente, a extraordinária capacidade de “aprendizagem” dos computadores vem acompanhada duma igualmente extraordinária *incapacidade* de imaginação. O computador não imagina, apenas interpreta rigorosamente o conhecimento que lhe transmitimos sob a forma de um programa.

Uma vez que não tem imaginação, o computador depende criticamente do modo como lhe apresentamos o conhecimento que queremos transmitir: esse conhecimento tem de estar descrito numa linguagem tal que não possa haver margem para qualquer ambiguidade, lacuna ou imprecisão. Uma linguagem com estas características denomina-se, genericamente, de *linguagem de programação*.

1.1 Linguagens de Programação

Para que um computador possa resolver um problema é necessário que consigamos fazer uma descrição do processo de resolução desse problema numa linguagem que o computador entenda. Infelizmente, a linguagem que os computadores entendem de forma “inata” é extraordinariamente pobre, levando a que a qualquer problema não trivial acabe por exigir uma exaustiva, entediante e extremamente complexa descrição do processo de resolução. As inúmeras linguagens de programação que têm sido inventadas visam precisamente aliviar o programador desse fardo, introduzindo elementos linguísticos capazes de simplificar enormemente essas descrições. Por exemplo, os conceitos de *função*, *matriz*, *somatório*, ou *número racional* não existem nativamente nos computadores, mas muitas linguagens de programação permitem a sua utilização de modo a facilitar a descrição de cálculos científicos. Isto implica, naturalmente, que tem de existir um processo capaz de transformar as descrições que o programador faz em descrições que o computador entenda. Embora este processo seja relativamente complexo, o que nos importa saber é que ele permite termos linguagens de programação mais próximas das capacidades do pensamento humano do que das capacidades de pensamento do computador.

Este último facto tem uma importância crucial pois permite que passemos a usar linguagens de programação, não só para instruir um computador sobre uma forma de resolver um problema, mas também para explicar rigorosamente esse processo a outros seres humanos. A linguagem de programação torna-se assim um meio de transmissão de conhecimento, tal como a linguagem da matemática o foi nos últimos milhares de anos.

Existe uma enorme diversidade de linguagens de programação, umas mais apetrechadas para a resolução de um determinado tipo de problemas, outras mais apetrechadas para a resolução de outro. A escolha de uma linguagem de programação deve estar condicionada, naturalmente, pelo tipo de problemas que queremos resolver, mas não deve ser um comprometimento total. Para quem programa é muito mais importante compreender os fundamentos e técnicas da programação do que dominar esta ou aquela linguagem. No entanto, para mais rigorosamente se explicar aqueles fundamentos e técnicas, convém exemplificá-los numa linguagem de programação concreta.

Uma vez que este texto se irá focar na programação aplicada à Arquitetura, vamos empregar uma linguagem de programação que esteja particularmente vocacionada para resolver problemas geométricos. Várias linguagens existem com esta vocação, em geral associadas a ferramentas de desenho assistido por computador. O ArchiCAD, por exemplo, disponibiliza uma linguagem de programação denominada GDL—acrónimo de *Geometric Description Language*—que permite ao utilizador programar as várias formas geométricas pretendidas. Já no caso do AutoCad, a linguagem de

programação empregue é o Auto Lisp, um dialecto de uma famosa linguagem de programação denominada Lisp.

Apesar de a linguagem GDL parecer ser muito diferente da linguagem Auto Lisp, os conceitos fundamentais são muito semelhantes. É sobre estes conceitos fundamentais da programação que nos iremos debruçar, embora, por motivos pedagógicos, seja conveniente particularizá-los numa única linguagem. Neste texto iremos explicar os fundamentos da programação através da utilização da linguagem Auto Lisp, não só pela sua facilidade de aprendizagem, mas também pela sua aplicabilidade prática. No entanto, uma vez apreendidos esses fundamentos, o leitor deverá ser capaz de os traduzir para qualquer outra linguagem de programação.

1.2 Lisp

Lisp é uma linguagem de programação extremamente simples mas com enorme flexibilidade. Embora tenha sido inicialmente inventada como uma linguagem matemática, cedo se procedeu à sua implementação em diversos computadores, o que os tornou em executores dos processos descritos nos programas Lisp.

Desde a sua génesis que a linguagem Lisp prima pela enorme facilidade de extensão. Isto permite ao programador ampliar a linguagem, dotando-a de novos meios para resolver problemas. Esta capacidade permitiu ao Lisp sobreviver à evolução da informática. Embora outras linguagens se tenham tornado obsoletas, a linguagem Lisp continua activa e é a segunda mais antiga linguagem ainda em utilização, sendo apenas ultrapassada pela linguagem Fortran.

A facilidade de utilização, adaptação e extensão da linguagem fez com que surgissem dezenas de diferentes dialectos: FranzLisp, ZetaLisp, Le-Lisp, MacLisp, InterLisp, Scheme, T, Nil, XLisp e AutoLISP são apenas alguns dos exemplos mais relevantes. Neste livro iremos debruçarmo-nos apenas sobre o dialecto AutoLISP.

A linguagem AutoLISP derivou da linguagem XLisp e foi incorporada no AutoCAD em 1986, tendo sido intensivamente usada desde então. Em 1999, o AutoCAD passou a disponibilizar uma versão melhorada do AutoLisp, denominada Visual LISP, que, entre outras diferenças, possui um compilador para uma execução mais eficiente e um depurador de programas para facilitar a detecção e correcção de erros. A influência do AutoCAD é tão grande que vários outros vendedores decidiram incluir uma implementação de AutoLISP nos seus próprios produtos, de modo a facilitar a transição de utilizadores.

Um aspecto característico do Lisp é ser uma linguagem interactiva. Cada porção de um programa pode ser desenvolvida, testada e corrigida independentemente das restantes. Deste modo, o Lisp permite que o desenvolvimento, teste e correcção de programas seja feito de uma forma incremen-

tal, o que facilita muito a tarefa do programador.

1.3 Exercícios

Exercicio 1.1 A exponenciação b^n é uma operação entre dois números b e n designados base e expoente, respectivamente. Quando n é um inteiro positivo, a exponenciação define-se como uma multiplicação repetida:

$$b^n = \underbrace{b \times b \times \cdots \times b}_n$$

Para um leitor que nunca tenha utilizado o operador de exponenciação, a definição anterior levanta várias questões cujas respostas poderão não lhe ser evidentes. Quantas multiplicações são realmente feitas? Serão n multiplicações? Serão $n - 1$ multiplicações? O que fazer no caso b^1 ? E no caso b^0 ?

Proponha uma definição matemática de exponenciação que não levante estas questões.

Exercicio 1.2 O que é uma linguagem de programação? Para que serve?

2 A Linguagem Auto Lisp

Nesta secção vamos descrever a linguagem de programação Lisp que iremos usar ao longo deste texto. Antes, porém, vamos discutir alguns aspectos que são comuns a todas as linguagens.

2.1 Sintaxe, Semântica e Pragmática

Todas as linguagens possuem *sintaxe, semântica* e *pragmática*.

Em termos muito simples, podemos descrever a sintaxe de uma linguagem como o conjunto de regras que determinam as frases que se podem construir nessa linguagem. Sem sintaxe qualquer concatenação arbitrária de palavras constituiria uma frase. Por exemplo, dadas as palavras “João,” “o,” “comeu,” e “bolo,” as regras da sintaxe da língua Portuguesa dizem-nos que “o João comeu o bolo” é uma frase e que “o comeu João bolo bolo” *não* é uma frase. Note-se que, de acordo com a sintaxe do Português, “o bolo comeu o João” é também uma frase sintaticamente correcta.

A sintaxe regula a construção das frases mas nada diz acerca do seu significado. É a semântica de uma linguagem que nos permite atribuir significado às frases da linguagem e que nos permite perceber que a frase “o bolo comeu o João” não faz sentido.

Finalmente, a pragmática diz respeito à forma usual como se escrevem as frases da linguagem. Para uma mesma linguagem, a pragmática varia de contexto para contexto: a forma como dois amigos íntimos falam entre si é diferente da forma usada por duas pessoas que mal se conhecem.

Estes três aspectos das linguagens estão também presentes quando discutimos linguagens de programação. Contrariamente ao que acontece com as linguagens *naturais* que empregamos para comunicarmos uns com os

outros, as linguagens de programação caracterizam-se por serem *formais*, obedecendo a um conjunto de regras muito mais simples e rígido que permite que sejam analisadas e processadas *mecanicamente*.

Neste texto iremos descrever a sintaxe e semântica da linguagem Auto Lisp. Embora existam formalismos matemáticos que permitem descrever rigorosamente aqueles dois aspectos das linguagens, eles exigem uma sofisticação matemática que, neste trabalho, é desapropriada. Por este motivo, iremos apenas empregar descrições informais.

Quanto à pragmática, será explicada à medida que formos introduzindo os elementos da linguagem. A linguagem Auto Lisp promove uma pragmática que, nalguns aspectos, difere significativamente do que é habitual nos restantes dialectos de Lisp. Uma vez que a pragmática não influencia a correcção dos nossos programas, mas apenas o seu *estilo*, iremos empregar uma pragmática que, embora ligeiramente diferente da usual em Auto Lisp, tem a vantagem de ser visualmente mais compacta.

2.2 Sintaxe e Semântica do Lisp

Quando comparada com a grande maioria das outras linguagens de programação, a linguagem Lisp possui uma sintaxe extraordinariamente simples baseada no conceito de *expressão*.¹

Uma expressão, em Lisp, pode ser construída empregando elementos primitivos como, por exemplo, os números; ou combinando outras expressões entre si como, por exemplo, quando somamos dois números. Como iremos ver, esta simples definição permite-nos construir expressões de complexidade arbitrária. No entanto, convém relembrar que a sintaxe restringe aquilo que podemos escrever: o facto de podermos combinar expressões para produzir outras expressões mais complexas não nos autoriza a escrever *qualquer* combinação de subexpressões. As combinações obedecem a regras sintáticas que iremos descrever ao longo do texto.

À semelhança do que acontece com a sintaxe, também a semântica da linguagem Lisp é, em geral, substancialmente mais simples que a de outras linguagens de programação. Como iremos ver, a semântica é determinada pelos *operadores* que as nossas expressões irão empregar. Um operador de soma, por exemplo, serve para somar dois números. Uma combinação que junte este operador e, por exemplo, os números 3 e 4 tem, como significado, a soma de 3 com 4, i.e., 7. No caso das linguagens de programação, a semântica de uma expressão é dada pelo computador que a vai *avaliar*.

¹Originalmente, Lisp dizia-se baseado em *S-expression*, uma abreviatura para expressões simbólicas. Esta caracterização ainda é aplicável mas, por agora, não a vamos empregar.

2.3 O Avaliador

Em Lisp, qualquer expressão tem um valor. Este conceito é de tal modo importante que todas as implementações da linguagem Lisp apresentam um avaliador, i.e., um programa destinado a interagir com o utilizador de modo a avaliar expressões por este fornecidas.

No caso do AutoCad, o avaliador está contido num ambiente interactivo de desenvolvimento, denominado Visual Lisp, e pode ser acedido através de menus (menu “Tools”, submenu “AutoLISP”, item “Visual LISP Editor”) ou através do comando `vlisp`. Uma vez acedido o Visual Lisp, o utilizador encontrará uma janela com o título “Visual LISP Console” e que se destina precisamente à interacção com o avaliador de Auto Lisp.

Assim, quando o utilizador começa a trabalhar com o Lisp, é-lhe apresentado um sinal (denominado “*prompt*”) e o Lisp fica à espera que o utilizador lhe forneça uma expressão.

_\\$

O texto “_\\$” é a “*prompt*” do Lisp, à frente da qual vão aparecer as expressões que o utilizador escrever. O Lisp interacciona com o utilizador executando um ciclo em que lê uma expressão, determina o seu valor e escreve o resultado. Este ciclo de acções designa-se, tradicionalmente, de *read-eval-print-loop* (e abrevia-se para REPL).

Quando o Lisp lê uma expressão, constroi internamente um objecto que a representa. Esse é o papel da fase de leitura. Na fase de avaliação, o objecto construído é analisado de modo a produzir um valor. Esta análise é feita empregando as regras da linguagem que determinam, para cada caso, qual o valor do objecto construído. Finalmente, o valor produzido é apresentado ao utilizador na fase de escrita através de uma representação textual desse valor.

Dada a existência do *read-eval-print-loop*, em Lisp não é necessário instruir o computador a escrever explicitamente os resultados de um cálculo, o que permite que o teste e correcção de erros seja bastante facilitada. A vantagem de Lisp ser uma linguagem interactiva está na rapidez com que se desenvolvem protótipos de programas, escrevendo, testando e corrigindo pequenos fragmentos de cada vez.

Exercicio 2.1 O que é o REPL?

2.4 Elementos da Linguagem

Em qualquer linguagem de programação precisamos de lidar com duas espécies de objectos: dados e procedimentos. Os dados são as entidades que pretendemos manipular. Os procedimentos são descrições das regras para manipular esses dados.

Se considerarmos a linguagem da matemática, podemos identificar os números como dados e as operações algébricas como procedimentos. As operações algébricas permitem-nos *combinar* os números entre si. Por exemplo, 2×2 é uma combinação. Uma outra combinação envolvendo mais dados será $2 \times 2 \times 2$ e, usando ainda mais dados, $2 \times 2 \times 2 \times 2$. No entanto, a menos que pretendamos ficar eternamente a resolver problemas de aritmética elementar, convém considerar operações mais elaboradas que representem padrões de cálculos. Na sequência de combinações que apresentámos é evidente que o padrão que está a emergir é o da operação de potenciação, i.e., multiplicação sucessiva, tendo esta operação sido definida na matemática há já muito tempo. A potenciação é, portanto, uma abstracção de uma sucessão de multiplicações.

Tal como a linguagem da matemática, uma linguagem de programação deve possuir dados e procedimentos primitivos, deve ser capaz de combinar quer os dados quer os procedimentos para produzir dados e procedimentos mais complexos e deve ser capaz de abstrair padrões de cálculo de modo a permitir tratá-los como operações simples, definindo novas operações que representem esses padrões de cálculo.

Mais à frente iremos ver como é possível definir essas abstracções em Lisp. Por agora, vamos debruçar-nos sobre os elementos mais básicos, os chamados *elementos primitivos* da linguagem, as entidades mais simples com que a linguagem lida. Os elementos primitivos podem ser divididos em dados primitivos e procedimentos primitivos. Um número, por exemplo, é um dado primitivo. Já a operação de adição de números é um procedimento primitivo.

2.5 Números

Como dissemos anteriormente, o Lisp executa um ciclo *read-eval-print*. Isto implica que tudo o que escrevemos no Lisp tem de ser avaliado, i.e., tem de ter um valor, valor esse que o Lisp escreve no *écran*.

Assim, se dermos um número ao avaliador, ele devolve-nos o valor desse número. Quanto vale um número? O melhor que podemos dizer é que ele vale por ele próprio. Por exemplo, o número 1 vale 1.

```
_\$ 1
1
_\$ 12345
12345
_\$ 4.5
4.5
```

Como se vê no exemplo, em Lisp, os números podem ser *inteiros* ou *reais*. Os reais são números com um ponto decimal e, no caso do Auto Lisp, tem de haver pelo menos um dígito antes do ponto decimal:

```

_§ 0.1
0.1
_§ .1
; error: misplaced dot on input

```

Note-se, no exemplo anterior, que o Auto Lisp produziu um *erro*. Um erro é uma situação anómala que impede o Auto Lisp de prosseguir o que estava a fazer. Neste caso, é apresentada uma *mensagem de erro* e o Auto Lisp volta ao princípio do ciclo *read-eval-print*.

Exercicio 2.2 Descubra qual é o maior real e o maior inteiro que o seu Lisp aceita.

Os reais podem ser escritos em notação decimal ou científica consoante o seu tamanho.

2.6 Combinações

Uma combinação é uma expressão que descreve a aplicação de um operador aos seus operandos. Por exemplo, na matemática, os números podem ser combinados usando operações como a soma ou o produto. Como exemplo de combinações matemáticas, temos $1+2$ e $1+2\times 3$. A soma e o produto de números são exemplos de operações extremamente elementares consideradas procedimentos primitivos.

Em Lisp, cria-se uma combinação escrevendo uma sequência de expressões entre um par de parênteses. Uma expressão é um elemento primitivo ou uma outra combinação. A expressão $(+ 1 2)$ é uma combinação dos elementos primitivos 1 e 2 através do procedimento primitivo `+`. Já no caso $(+ 1 (* 2 3))$ a combinação é entre 1 e $(* 2 3)$, sendo esta última expressão uma outra combinação. Note-se que cada expressão deve ser separada das restantes por um ou mais espaços. Assim, embora a combinação $(* 2 3)$ possua as três expressões `*`, 2 e 3, a combinação $(* 2 3)$ só possui duas—`*` 2 e 3—sendo que a primeira delas não tem qualquer significado pré-definido.

Por agora, as únicas combinações com utilidade são aquelas em que as expressões correspondem a operadores e operandos. Por convenção, o Lisp considera que o primeiro elemento de uma combinação é um operador e os restantes são os operandos.

A notação que o Lisp utiliza para construir expressões (operador primeiro e operandos a seguir) é designada por notação prefixa por o operador ocorrer previamente aos operandos. Esta notação costuma causar alguma perplexidade a quem inicia o estudo da linguagem, que espera uma notação mais próxima da que aprendeu em aritmética e que é usada habitualmente nas outras linguagens de programação. Nestas, a expressão $(+ 1 (* 2 3))$ é usualmente escrita na forma $1 + 2 * 3$ (designada notação infixa, por o operador ocorrer *entre* os operandos) que, normalmente,

é mais simples de ler por um ser humano. No entanto, a notação prefixa usada pelo Lisp tem vantagens sobre a notação infixa:

- É muito fácil usar operadores que têm um número variável de operandos,² como por exemplo $(+ 1 2 3)$ ou $(+ 1 2 3 4 5 6 7 8 9 10)$. Na maioria das outras linguagens de programação apenas existem operadores unários ou binários e é necessário explicitar os operador binários entre cada dois operandos: $1 + 2 + 3$ ou $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$. Se se pretender um operador ternário (ou outro) já não se consegue escrever do mesmo modo.
- Não existe precedência entre os operadores. Nas linguagens de notação infixa, a expressão $1 + 2 * 3$ tem de ser calculada como se se tivesse escrito $1 + (2 * 3)$, e não $(1 + 2) * 3$, i.e., foi criada uma precedência para eliminar as ambiguidades. Essa precedência pode ser alterada através do emprego de parênteses. Em Lisp, as expressões seriam necessariamente distintas, $(+ 1 (* 2 3))$ ou $(* (+ 1 2) 3)$, não podendo haver qualquer ambiguidade.
- Os operadores que nós definimos usam-se exactamente da mesma maneira que os operadores da linguagem: operador primeiro e operandos a seguir. Na maioria das outras linguagens, os operadores são infixos, i.e., estão entre os operandos, e os procedimentos por nós definidos são prefixos, i.e., primeiro vem o nome do procedimento e depois os operandos. Isto impede a extensão da linguagem de uma forma coerente.

Para exemplificar este último aspecto, consideremos a operação de exponenciação numa linguagem com operadores infixos. Para ser coerente com o resto da linguagem, deveria existir um operador, por exemplo $\star\star$ que permitisse escrever $3 \star\star 4$ para indicar a quarta potência de 3. Como, em geral, esse operador não existe, somos obrigados a criar um procedimento que o implemente mas, neste caso, a sintaxe muda radicalmente pois, em geral, o utilizador não pode definir novos operadores infixos. A consequência é que o novo operador, por não se poder usar de forma infixa e ter de se usar de forma prefixa, não fica coerente com as restantes operações da linguagem, como a soma e a multiplicação. Em Lisp, pelo contrário, tanto podemos escrever $(* 3 3 3 3)$ como definir uma função que permita escrever $(\star\star 3 4)$.

Para além das notações infixa e prefixa existe ainda a notação posfixa em que o operador se escreve após os operandos. Esta notação tem as mesmas propriedades da notação prefixa mas é, em geral, mais difícil de ler

²Estes operadores dizem-se *variádicos*.

pelo facto de ser necessário ler todos os operandos antes de se perceber o que se vai fazer com eles.

2.7 Indentação

A desvantagem da notação prefixa está na escrita de combinações complexas. Por exemplo, a expressão $1+2*3-4/5*6$ lê-se com relativa facilidade mas, quando escrita na sintaxe do Lisp, $(- (+ 1 (* 2 3)) (* (/ 4 5) 6))$, fica com uma forma que, para quem não está ainda habituado, pode ser mais difícil de perceber devido à acumulação de parênteses.

Para tornarmos estas expressões mais fáceis de entender, podemos (e devemos) empregar a técnica da *indentação*. Esta técnica baseia-se no uso de diferentes alinhamentos na disposição textual dos programas de modo a facilitar a sua leitura. Assim, ao invés de escrevermos as nossas expressões todas numa linha ou com arranjos arbitrários entre linhas, escrevemo-las ao longo de várias linhas e com um alinhamento entre linhas que mostre o relacionamento das várias subexpressões com a expressão que as contém.

A regra para indentação de combinações Lisp é extremamente simples: numa linha coloca-se o operador e o primeiro operando; os restantes operandos vêm alinhados por debaixo do primeiro, usando-se um número suficiente de espaços em branco à esquerda para que os operandos fiquem correctamente arrumados. No caso de uma expressão ser curta, podemos escrevê-la numa só linha, com os operandos logo a seguir ao operador, com um espaço em branco entre cada um. Usando estas duas regras, podemos reescrever a expressão anterior na seguinte forma, onde usamos linhas verticais imaginárias para salientar a indentação:

```
(- |(+ |1  
|  |(* |2 |3 |))  
|  |(*) |(/ |4 |5 |)  
|  |  |6 |))
```

Note-se que o facto de se dispor uma combinação ao longo de várias linhas em nada afecta a forma como o Lisp a lê. A correcta delimitação dos elementos da combinação é feita apenas pela sua separação visual e pela utilização de parênteses correctamente emparelhados.

Quando a regra de indentação não é suficiente para produzir uma disposição de linhas de texto que seja esteticamente agradável, usam-se pequenas variações, como seja colocar o operador numa linha e os operandos por debaixo, por exemplo:

```
(um-operador-com-um-nome-muito-grande  
1 | 2 | 3 | 4)
```

No caso geral, podemos ter de empregar as várias regras em simultâneo:

```
(um-operador | (um-operador-com-um-nome-muito-grande
    1 2 3 4)
    (outro-operador | 5
        6
        (e-ainda-outro | 7
            8))
    (e-o-ultimo-operador 9 10))
```

Alguns operadores da linguagem possuem uma regra de indentação própria. Essas regras serão explicadas à medida que formos introduzindo esses operadores.

A indentação é fundamental em Lisp pois é muito fácil escrever código complexo. A grande maioria dos editores preparados para Lisp formatam automaticamente os programas à medida que os escrevemos e mostram o emparelhamento de parênteses. Desta forma, após algum tempo de prática, torna-se muito fácil escrever e ler os programas, por mais complexa que possa parecer a sua estrutura.

Exercicio 2.3 Qual é a diferença entre a notação prefixa e a notação infixa?

Exercicio 2.4 Em matemática, é usual empregar simultaneamente as diferentes notações prefixa, infixa e posfixa. Escreva exemplos de expressões matemáticas que utilizem estas diferentes notações.

Exercicio 2.5 Converta as seguintes expressões da notação infixa da aritmética para a notação prefixa do Lisp:

1. $1 + 2 - 3$
2. $1 - 2 \times 3$
3. $1 \times 2 - 3$
4. $1 \times 2 \times 3$
5. $(1 - 2) \times 3$
6. $(1 - 2) + 3$
7. $1 - (2 + 3)$
8. $2 \times 2 + 3 \times 3 \times 3$

Exercicio 2.6 Converta as seguintes expressões da notação prefixa do Lisp para a notação infixa da aritmética:

1. $(\ast (/ 1 2) 3)$
2. $(\ast 1 (- 2 3))$
3. $(/ (+ 1 2) 3)$
4. $(/ (/ 1 2) 3)$
5. $(/ 1 (/ 2 3))$
6. $(- (- 1 2) 3)$
7. $(- 1 2 3)$

Exercicio 2.7 Indente a seguinte expressão de modo a ter um único operando por linha.

```
(\ast (+ (/ 3 2) (- (\ast (/ 5 2) 3) 1) (- 3 2))) 2)
```

2.8 Avaliação de Combinações

Como já vimos, o Lisp considera que o primeiro elemento de uma combinação é um operador e os restantes são os operandos.

O avaliador determina o valor de uma combinação como o resultado de aplicar o procedimento especificado pelo operador ao valor dos operandos. O valor de cada operando é designado de argumento do procedimento. Assim, o valor da combinação `(+ 1 (* 2 3))` é o resultado de somar o valor de 1 com o valor de `(* 2 3)`. Como já se viu, 1 vale 1 e `(* 2 3)` é uma combinação cujo valor é o resultado de multiplicar o valor de 2 pelo valor de 3, o que dá 6. Finalmente, somando 1 a 6 obtemos 7.

```
_\$ (* 2 3)
6
_\$ (+ 1 (* 2 3))
7
```

Uma diferença importante entre o Auto Lisp e a aritmética ocorre na operação de divisão. Matematicamente falando, $7/2$ é uma fração, i.e., não é um número inteiro. Infelizmente, o Auto Lisp não sabe lidar com frações e, consequentemente, emprega uma definição ligeiramente diferente para o operador de divisão: em Auto Lisp, o símbolo `/` representa a divisão *inteira*, i.e., o número que multiplicado pelo *divisor* e somado ao *resto* iguala o *dividendo*. Assim, `(/ 7 2)` avalia para 3. No entanto, quando está a lidar com números reais, o Auto Lisp já faz a divisão segundo as regras da matemática mas, possivelmente, envolvendo perdas de precisão. Assim, `(/ 7.0 2)` produz 3.5.

Exercicio 2.8 Calcule o valor das seguintes expressões Lisp:

1. `(* (/ 1 2) 3)`
2. `(* 1 (- 2 3))`
3. `(/ (+ 1 2) 3)`
4. `(- (- 1 2) 3)`
5. `(- 1 2 3)`
6. `(- 1)`

2.9 Cadeias de Caracteres

As *cadeias de caracteres* (também denominadas *strings*) são outro tipo de dados primitivo. Um carácter é uma letra, um dígito ou qualquer outro símbolo gráfico, incluindo os símbolos gráficos não visíveis como o espaço, a tabulação e outros. Uma cadeia de caracteres é especificada através de uma sequência de caracteres delimitada por aspas. Tal como com os números, o valor de uma cadeia de caracteres é a própria cadeia de caracteres.

Sequência	Resultado
\ \	o carácter \ (<i>backslash</i>)
\ "	o carácter " (aspas)
\e	o carácter <i>escape</i>
\n	o carácter de mudança de linha (<i>newline</i>)
\r	o carácter de mudança de linha (<i>carriage return</i>)
\t	o carácter de tabulação (<i>tab</i>)
\nnn	o carácter cujo código octal é nnn

Tabela 1: Caracteres de *escape* válidos em Auto Lisp

```
_\$ "Ola"
"Ola"
_\$ "Eu sou o meu valor"
"Eu sou o meu valor"
```

Sendo uma *string* delimitada por aspas fica a dúvida sobre como criar uma *string* que contém aspas. Para isso, e para outros caracteres especiais, existe um carácter que o Auto Lisp interpreta de forma distinta: quando, na criação de uma *string*, aparece o carácter \ ele assinala que o próximo carácter tem de ser processado de forma especial. Por exemplo, para se a criar a *string* correspondente à frase

O Manuel disse “Bom dia!” ao Pedro.

temos de escrever:

```
"O Manuel disse \"Bom dia!\" ao Pedro."
```

O carácter \ é denominado um carácter de *escape* e permite a inclusão em *strings* de caracteres que, de outra forma, seriam difíceis de inserir. A Tabela 1 mostra as várias possibilidades.

Para além dos números e das *strings*, o Auto Lisp possui ainda outros tipos de elementos primitivos que iremos explicar mais tarde.

2.10 Operadores de *Strings*

Para além dos operadores que já vimos para números, existem operadores para *strings*. Por exemplo, para concatenar várias *strings*, existe o operador strcat. A concatenação de várias *strings* produz uma só *string* com todos os caracteres dessas várias *strings* e pela mesma ordem:

```

_ $ (strcat "1" "2")
"12"
_ $ (strcat "um" "dois" "tres" "quatro")
"umdoistresquatro"
_ $ (strcat "eu" " " "sou" " " "uma" " " "string")
"eu sou uma string"
_ $ (strcat "E eu" " sou " "outra")
"E eu sou outra"

```

Para saber o número de caracteres de uma *string* existe o operador `strlen`:

```

_ $ (strlen (strcat "eu" " " "sou" " " "uma" " " "string"))
17
_ $ (strlen "")
0

```

Note-se que as aspas são os delimitadores de *strings* e não contam como caracteres.

A função `substr` providencia uma terceira operação útil com *strings*: ela permite obter parte de uma *string*. A parte a obter é especificada através da posição do primeiro carácter e do número de caracteres a considerar. Numa *string*, a posição de um carácter é também denominada de *índice* do carácter. O Auto Lisp considera que o primeiro carácter de uma *string* tem o índice 1. A função `substr` recebe, como argumentos, uma *string*, um índice e, opcionalmente, um número de caracteres e devolve a parte da string que começa no índice dado e que tem o número de caracteres dado no parâmetro opcional ou todos os restantes caracteres no caso de esse número não ter sido dado. Assim, temos:

```

_ $ (substr "abcd" 1)
"abcd"
_ $ (substr "abcd" 2)
"bcd"
_ $ (substr "abcd" 2 2)
"bc"

```

Exercicio 2.9 Qual é o resultado das seguintes avaliações?

1. `(strcat "a" "vista" "da" " baixa" "da" " banheira")`
2. `(substr (strcat "Bom dia") 1 3)`
3. `(substr (strcat "Bom dia") 5)`

2.11 Definição de Funções

Para além das operações básicas aritméticas, a matemática disponibiliza-nos um vastíssimo conjunto de outras operações que se definem à custa daquelas. Por exemplo, o *quadrado* de um número é uma operação (também designada por *função*) que, dado um número, produz o resultado de multiplicar esse número por ele próprio. Matematicamente falando, define-se o quadrado de um número pela função $x^2 = x \cdot x$.

Tal como em matemática, pode-se definir numa linguagem de programação a função que obtém o quadrado de um número. Em Lisp, para obtermos o quadrado de um número qualquer, por exemplo, 5, escrevemos a combinação $(\ast 5 5)$. No caso geral, dado um número qualquer x , sabemos que obtemos o seu quadrado escrevendo $(\ast x x)$. Falta apenas associar um nome que indique que, dado um número x , obtemos o seu quadrado avaliando $(\ast x x)$. Lisp permite-nos fazer isso através da operação `defun` (abreviatura de `define function`):

```
_\$ (defun quadrado (x) (\* x x))
QUADRADO
```

Note-se que o Auto Lisp, ao avaliar a definição da função `quadrado` devolve como valor o nome da função definida. Note-se ainda que o Auto Lisp escreve esse nome em maiúsculas. Na realidade, o que se passa é que, por motivos históricos, todos os nomes que escrevemos no Auto Lisp são traduzidos para maiúsculas assim que são lidos. Por este motivo, `quadrado`, `QUADRADO`, `Quadrado`, `qUaDrAdo`, ou qualquer outra combinação de maiúsculas e minúsculas designa o mesmo nome em Auto Lisp: `QUADRADO`. Apesar da conversão para maiúsculas, que é feita na leitura, é pragmática usual escrevermos os nossos programas em letras minúsculas.

Como se pode ver pela definição da função `quadrado`, para se definirem novos procedimentos em Lisp é necessário criar uma combinação de quatro elementos. O primeiro elemento desta combinação é a palavra `defun`, que informa o avaliador que estamos a definir uma função. O segundo elemento é o nome da função que queremos definir, o terceiro elemento é uma combinação com os parâmetros da função e o quarto elemento é a expressão que determina o valor da função para aqueles parâmetros. De modo genérico podemos indicar que a definição de funções é feita usando a seguinte forma:

```
(defun nome (parâmetro1 ... parâmetron)
  corpo)
```

Quando se dá uma expressão desta forma ao avaliador, ele acrescenta a função ao conjunto de funções da linguagem, associando-a ao nome que lhe demos e devolve como valor da definição o nome da função definida. Se já existir uma função com o mesmo nome, a definição anterior é simplesmente esquecida. Os parâmetros de um procedimento são designados *parâmetros formais* e são os nomes usados no corpo de expressões para nos referirmos aos argumentos correspondentes. Quando escrevemos no avaliador de Lisp `(quadrado 5)`, 5 é o *argumento* do procedimento. Durante o cálculo da função este argumento está associado ao parâmetro formal x . Os argumentos de uma função são também designados por *parâmetros actuais*.

No caso da função `quadrado`, a sua definição diz que para se determinar o quadrado de um número x devemos multiplicar esse número por ele

próprio $(\star x x)$. Esta definição associa a palavra quadrado a um procedimento, i.e., a uma descrição do modo de produzir o resultado pretendido. Note-se que este procedimento possui parâmetros, permitindo o seu uso com diferentes argumentos. Para se utilizar este procedimento, podemos avaliar as seguinte expressões:

```
_\$ (quadrado 5)
25
_\$ (quadrado 6)
36
```

A regra de avaliação de combinações que descrevemos anteriormente é também válida para as funções por nós definidas. Assim, a avaliação da expressão $(\text{quadrado } (+ 1 2))$ passa pela avaliação do operando $(+ 1 2)$. Este operando tem como valor 3, valor esse que é usado pela função no lugar do parâmetro x . O corpo da função é então avaliado mas substituindo todas as ocorrências de x pelo valor 3, i.e., o valor final será o da combinação $(\star 3 3)$.

Para se perceber a avaliação de combinações que o Lisp emprega, é útil decompor essa avaliação nas suas etapas mais elementares. No seguinte exemplo mostramos o processo de avaliação para a expressão

```
(quadrado (quadrado (+ 1 2)))
```

Em cada passo, uma das sub-expressões foi avaliada.

```
(quadrado (quadrado (+ 1 2)))
      ↓
  (quadrado (quadrado 3))
      ↓
  (quadrado (* 3 3))
      ↓
  (quadrado 9)
      ↓
  (* 9 9)
      ↓
  81
```

Em resumo, para se invocar uma função, é necessário construir uma combinação cujo primeiro elemento seja uma expressão que avalia para a função que se pretende invocar e cujos restantes elementos são expressões que avaliam para os argumentos que se pretende passar à função. O resultado da avaliação da combinação é o valor calculado pela função para aqueles argumentos.

A avaliação de uma combinação deste género processa-se nos seguintes passos:

1. Todos os elementos da combinação são avaliados, sendo que o valor do primeiro elemento é necessariamente uma função.
2. Associam-se os parâmetros formais dessa função aos argumentos, i.e., aos valores dos restantes elementos da combinação. Cada parâmetro é associado a um argumento, de acordo com a ordem dos parâmetros e argumentos. É gerado um erro sempre que o número de parâmetros não é igual ao número de argumentos.
3. Avalia-se o corpo da função tendo em conta estas associações entre os parâmetros e os argumentos.

Como dissemos, a definição de funções permite-nos associar um procedimento a um nome. Isto implica que o Lisp tem de possuir uma memória onde possa guardar a função e a sua associação ao nome dado. Esta memória do Lisp designa-se *ambiente*.

Note-se que este ambiente apenas existe enquanto estamos a trabalhar com a linguagem. Quando terminamos, perde-se todo o ambiente. Para evitar perdermos as definições de procedimentos que tenhamos feito, convém registá-las num suporte persistente, como seja um ficheiro. Para facilitar a utilização, o Lisp permite que se avaliem as diversas definições a partir de um ficheiro. Por este motivo, o processo usual de trabalho com Lisp consiste em escrever as várias definições em ficheiros, embora se continue a usar o avaliador de Lisp para experimentar e testar o correcto funcionamento das nossas definições.

Exercicio 2.10 Defina a função `dobro` que, dado um número, calcula o seu dobro.

2.12 Símbolos

A definição de funções em Lisp passa pela utilização de nomes: nomes para as funções e nomes para os parâmetros das funções.

Internamente, o Lisp representa os nomes através de *símbolos*. Cada nome diferente que é lido pelo Lisp acaba por ter um símbolo associado e é este símbolo que é usado para determinar o significado do nome, por exemplo, qual a função ou qual o parâmetro a que ele diz respeito.

Em Lisp, quase não existem limitações para a escrita de nomes. Um nome como `quadrado` é tão válido como `+ ou como 1+2*3`, pois o que separa um nome dos outros elementos de uma combinação são apenas parênteses e espaço em branco.³

No caso do Auto Lisp, os únicos caracteres que não se podem utilizar nos nomes dos símbolos são o abrir e fechar parênteses (e), a plica (também chamada *apóstrofo*) ', as aspas ", o ponto . e o ponto e vírgula ;.

³Dentro do conceito de “espaço em branco” consideram-se os caracteres de espaço, de tabulação e de mudança de linha.

Todos os restantes caracteres se podem usar para nomes de funções mas, na prática, a criação de nomes segue algumas regras que convém ter presentes:

- Apenas se devem usar as letras do alfabeto, os dígitos, os símbolos aritméticos e alguns caracteres de pontuação, como o ponto de exclamação e o ponto de interrogação. Por motivos de portabilidade, convém evitar o uso de caracteres acentuados.
- Se o nome da função é composto por várias palavras, deve-se separar as palavras com traços (-). Por exemplo, uma função que calcula a área de um círculo poderá ter como nome o símbolo `area-circulo`. Já os nomes `areacirculo`, `area_circulo` e `area+circulo` serão menos apropriados.
- Se o nome corresponde a uma interrogação, deve-se terminar o nome com um ponto de interrogação (?) ou, na tradição Lisp, com a letra p.⁴ Por exemplo, uma função que indica se um número é par poderá ter o nome `par?`.
- Se a função faz uma conversão entre dois tipos de valores, o nome poderá ser obtido a partir dos nomes dos tipos com uma seta entre eles a indicar a direcção da conversão. Por exemplo, uma função que converte euros para libras poderá ter como nome `euros->libras` ou, ainda melhor, `libras<-euros`.

Exercicio 2.11 Indique um nome apropriado para cada uma das seguintes funções:

1. Função que calcula o volume de uma esfera.
2. Função que indica se um número é primo.
3. Função que converte uma medida em centímetros para polegadas.

Exercicio 2.12 Defina a função `radianos<-graus` que recebe uma quantidade angular em graus e calcula o valor correspondente em radianos. Note que 180 graus correspondem a π radianos.

Exercicio 2.13 Defina a função `graus<-radianos` que recebe uma quantidade angular em radianos e calcula o valor correspondente em graus.

Exercicio 2.14 Defina a função que calcula o perímetro de uma circunferência de raio r .

⁴Esta letra é a abreviatura de *predicado*, um tipo de função que iremos estudar mais à frente.

2.13 Funções de Múltiplos Parâmetros

Todas as funções que definimos até este momento possuem um único parâmetro. Vamos agora considerar funções com mais do que um parâmetro.

Por exemplo, a área A de um triângulo de base b e altura c define-se matematicamente por

$$A(b, c) = \frac{b \cdot c}{2}$$

Em Lisp, teremos:

```
(defun A (b c) (/ (* b c) 2))
```

Como se pode ver, a definição da função em Lisp é idêntica à definição correspondente em Matemática, com a diferença de os operadores se usarem de forma prefixa e o símbolo de definição matemática $=$ se escrever defun. No entanto, contrariamente ao que é usual ocorrer em Matemática, os nomes que empregamos em Lisp devem ter um significado claro. Assim, ao invés de se escrever A é preferível escrever `area-triangulo`. Do mesmo modo, ao invés de se escrever b e c , seria preferível escrever `base` e `altura`. Tendo estes aspectos em conta, podemos apresentar uma definição mais legível:

```
(defun area-triangulo (base altura)
  (/ (* base altura) 2))
```

Quando o número de definições aumenta torna-se particularmente importante para quem as lê que se perceba rapidamente o seu significado e, por isso, é de crucial importância que se faça uma boa escolha de nomes.

Exercício 2.15 Defina uma função que calcula o volume de um paralelipípedo a partir do seu comprimento, altura e largura. Empregue nomes suficientemente claros.

Exercício 2.16 Defina a função `media` que calcula o valor médio entre dois outros valores. Por exemplo $(\text{media} \ 2 \ 3) \rightarrow 2.5$.

2.14 Encadeamento de Funções

Todas funções por nós definidas são consideradas pelo avaliador de Lisp em pé de igualdade com todas as outras definições. Isto permite que elas possam ser usadas para definir ainda outras funções. Por exemplo, após termos definido a função `quadrado`, podemos definir a função que calcula a área de um círculo de raio r através da fórmula $\pi \cdot r^2$.

```
(defun area-circulo (raio)
  (* pi (quadrado raio)))
```

Durante a avaliação de uma expressão destinada a computar a área de um círculo, a função `quadrado` acabará por ser invocada. Isso é visível na seguinte sequência de passos de avaliação:

```
(area-circulo 2)
(* pi (quadrado 2))
(* 3.14159 (quadrado 2))
(* 3.14159 (* 2 2))
(* 3.14159 4)
12.5664
```

Exercicio 2.17 Defina a função que calcula o volume de um cilindro com um determinado raio e comprimento. Esse volume corresponde ao produto da área da base pelo comprimento do cilindro.

2.15 Funções Pré-Definidas

A possibilidade de se definirem novas funções é fundamental para aumentarmos a flexibilidade da linguagem e a sua capacidade de se adaptar aos problemas que pretendemos resolver. As novas funções, contudo, precisam de ser definidas à custa de outras que, ou foram também por nós definidas ou, no limite, já estavam pré-definidas na linguagem.

Isto mesmo se verifica no caso da função `area-circulo` que definimos acima: ela está definida à custa da função `quadrado` (que foi também por nós definida) e à custa da operação de multiplicação. No caso da função `quadrado`, ela foi definida com base na operação de multiplicação. A operação de multiplicação que, em última análise, é a base do funcionamento da função `area-circulo` é, na realidade, uma função pré-definida do Lisp.

Como iremos ver, o Lisp providencia um conjunto razoavelmente grande de funções pré-definidas. Em muitos casos, são suficientes para o que pretendemos, mas não nos devemos coibir de definir novas funções sempre que acharmos necessário.

A Tabela 2 apresenta uma seleção de funções matemáticas pré-definidas do Auto Lisp. Note-se que, devido às limitações sintáticas do Auto Lisp (e que são comuns a todas as outras linguagens de programação), há vários casos em que uma função em Auto Lisp emprega uma notação diferente daquela que é usual em matemática. Por exemplo, a função raiz quadrada \sqrt{x} escreve-se como `(sqrt x)`. O nome `sqrt` é uma contracção do Inglês *square root* e contracções semelhantes são empregues para várias outras funções. Por exemplo, a função valor absoluto $|x|$ escreve-se `(abs x)` (de *absolute value*), a função parte inteira $[x]$ escreve-se `(fix x)` (de *fixed point*) e a função potência x^y escreve-se `(expt x y)` (de *exponential*). A Tabela 3 mostra as equivalências mais relevantes entre invocações de funções em Auto Lisp e as correspondentes invocações na Matemática.

A Tabela 4 apresenta uma seleção de funções sobre *strings* pré-definidas do Auto Lisp. À medida que formos apresentando novos tópicos do Auto Lisp iremos explicando outras funções pré-definidas que sejam relevantes para o assunto.

Função	Argumentos	Resultado
+	Vários números	A adição de todos os argumentos. Sem argumentos, zero.
-	Vários números	Com apenas um argumento, o seu simétrico. Com mais de um argumento, a subtração ao primeiro de todos os restantes. Sem argumentos, zero.
*	Vários números	A multiplicação de todos os argumentos. Sem argumentos, zero.
/	Vários números	A divisão do primeiro argumento por todos os restantes. Sem argumentos, zero.
1+	Um número	A soma do argumento com um.
1-	Um número	A substracção do argumento com um.
abs	Um número	O valor absoluto do argumento.
sin	Um número	O seno do argumento (em radianos).
cos	Um número	O cosseno do argumento (em radianos).
atan	Um ou dois números	Com um argumento, o arco tangente do argumento (em radianos). Com dois argumentos, o arco tangente da divisão do primeiro pelo segundo (em radianos). O sinal dos argumentos é usado para determinar o quadrante.
sqrt	Um número	A raiz quadrada do argumento.
exp	Um número	A exponencial de base e .
expt	Dois números	O primeiro argumento elevado ao segundo argumento.
log	Um número	O logaritmo natural do argumento.
max	Vários números	O maior dos argumentos.
min	Vários números	O menor dos argumentos.
rem	Dois ou mais números	Com dois argumentos, o resto da divisão do primeiro pelo segundo. Com mais argumentos, o resto da divisão do resultado anterior pelo argumento seguinte.
fix	Um número	O argumento sem a parte fraccionária.
float	Um número	O argumento convertido em número real.

Tabela 2: Funções matemáticas pré-definidas do Auto Lisp.

Exercicio 2.18 Como pode verificar pelas Tabelas 2 e 3, em Auto Lisp uma multiplicação sem argumentos produz o valor zero, tal como acontece no caso de uma divisão sem argumentos. No entanto, os matemáticos contestam este resultado, afirmando que está errado e que o valor correcto deveria ser outro. Concorda? Em caso afirmativo, qual deveria ser o valor correcto?

Auto Lisp	Matemática
(+ x ₀ x ₁ ... x _n)	$x_0 + x_1 + \dots + x_n$
(+ x)	x
(+)	0
(- x ₀ x ₁ ... x _n)	$x_0 - x_1 - \dots - x_n$
(- x)	$-x$
(-)	0
(* x ₀ x ₁ ... x _n)	$x_0 \times x_1 \times \dots \times x_n$
(* x)	x
(*)	0
(/ x ₀ x ₁ ... x _n)	$x_0/x_1/\dots/x_n$
(/ x)	x
(/)	0
(1+ x)	$x + 1$
(1- x)	$x - 1$
(abs x)	$ x $
(sin x)	$\sin x$
(cos x)	$\cos x$
(atan x)	$\tan x$
(atan x y)	$\tan \frac{x}{y}$
(sqrt x)	\sqrt{x}
(exp x)	e^x
(expt x y)	x^y
(log x)	$\log x$
(fix x)	$\lfloor x \rfloor$

Tabela 3: Funções matemáticas pré-definidas do Auto Lisp.

Exercício 2.19 Embora o seno (`sin`) e o cosseno (`cos`) sejam funções pré-definidas em Auto Lisp, a tangente (`tan`) não é. Defina-a a partir da fórmula $\tan x = \frac{\sin x}{\cos x}$.

Exercício 2.20 Do conjunto das funções trigonométricas *inversas*, o Auto Lisp apenas provê a arco tangente (`atan`). Defina também as funções arco-seno (`asin`) e arco-cosseno (`acos`), cujas definições matemáticas são:

$$\text{asin } x = \text{atan} \frac{x}{\sqrt{1-x^2}}$$

$$\text{acos } x = \text{atan} \frac{\sqrt{1-x^2}}{x}$$

Exercício 2.21 A função `fix` permite *truncar* um número real, produzindo o número inteiro que se obtém pela eliminação da parte fracionária desse real. Defina a função `round` que permite *arredondar* um número real, i.e., produzir o inteiro que é mais próximo desse número real.

Exercício 2.22 Para além da função `round` que arredonda para o inteiro mais próximo e da

Função	Argumentos	Resultado
strcat	Várias <i>strings</i>	A concatenação de todos os argumentos. Sem argumentos, a <i>string</i> vazia .
strlen	Várias <i>strings</i>	O número de caracteres da concatenação das <i>strings</i> . Sem argumentos, devolve zero.
substr	Uma <i>string</i> , um inteiro (o índice) e, opcionalmente, outro inteiro (o número de caracteres)	A parte da <i>string</i> que começa no índice dado e que tem o número de caracteres dado ou todos os restantes caracteres no caso de esse número não ter sido dado.
strcase	Uma <i>string</i> e um booleano opcional	Com um argumento ou segundo argumento <i>nil</i> , a conversão para maiúsculas do argumento, caso contrário, conversão para minúsculas.
atof	Uma <i>string</i>	O número real cuja representação textual é o argumento.
atoi	Uma <i>string</i>	O número, sem a parte fraccionária, cuja representação textual é o argumento.
itoa	Um número	A representação textual do argumento.
rtos	Um, dois ou três números	A representação textual do argumento, de acordo com o modo especificado no segundo argumento e com a precisão especificada no terceiro argumento.

Tabela 4: Operações pré-definidas envolvendo *strings*.

função *fix* que arredonda “para baixo,” é usual considerar ainda a função *ceiling* que arredonda “para cima.” Defina esta função.

Exercicio 2.23 Traduza as seguintes expressões matemáticas para Auto Lisp:

1. $\sqrt{\frac{1}{\log_2(3 - 9 \log 25)}}$
2. $\frac{\cos^4 \frac{2}{\sqrt{5}}}{\operatorname{atan} 3}$
3. $\frac{1}{2} + \sqrt{3} + \sin^{\frac{5}{2}} 2$

Exercicio 2.24 Traduza as seguintes expressões Auto Lisp para a notação matemática:

1. $(\log (\sin (+ (\operatorname{expt} 2 4) (/ (\operatorname{fix} (\operatorname{atan} \pi)) (\operatorname{sqrt} 5)))))$
2. $(\operatorname{expt} (\cos (\cos (\cos 0.5))) 5)$
3. $(\sin (/ (\cos (/ (\sin (/ \pi 3)) 3)) 3))$

Exercicio 2.25 Defina a função *impar?* que, dado um número, testa se ele é ímpar, i.e., se o resto da divisão desse número por dois é um. Para calcular o resto da divisão de um número por outro, utilize a função pré-definida *rem*.

Exercicio 2.26 Em várias actividades é usual empregarem-se expressões padronizadas. Por exemplo, se o aluno Passos Dias Aguiar pretende obter uma certidão de matrícula da universidade que frequenta, terá de escrever uma frase da forma:

Passos Dias Aguiar vem respeitosamente pedir a V. Exa. que se digne passar uma certidão de matrícula.

Por outro lado, se a aluna Maria Gustava dos Anjos pretende um certificado de habilitações, deverá escrever:

Maria Gustava dos Anjos vem respeitosamente pedir a V. Exa. que se digne passar um certificado de habilitações.

Para simplificar a vida destas pessoas, pretende-se que implemente uma função denominada `requerimento` que, convenientemente parametrizada, gera uma *string* com a frase apropriada para qualquer um dos fins anteriormente exemplificados e também para outros do mesmo género que possam vir a ser necessários. Teste a sua função para garantir que consegue reproduzir os dois exemplos anteriores passando o mínimo de informação nos argumentos da função.

Exercicio 2.27 Também na actividade da Arquitectura é usual os projectos necessitarem de informações textuais escritas numa forma padronizada. Por exemplo, considere as seguintes três frases contidas nos Cadernos de Encargos de três diferentes projectos:

Toda a caixilharia será metálica, de alumínio anodizado, cor natural, construída com os perfis utilizados pela Serralharia do Corvo (ou semelhante), sujeitos a aprovação da Fiscalização.

Toda a caixilharia será metálica, de alumínio lacado, cor branca, construída com os perfis utilizados pela Technal (ou semelhante), sujeitos a aprovação da Fiscalização.

Toda a caixilharia será metálica, de aço zíncado, preparado com primários de aderência, primário epoxídico de protecção e acabamento com esmalte SMP, cor 1050-B90G (NCS), construída com os perfis standard existentes no mercado, sujeitos a aprovação da Fiscalização.

Defina uma função que, convenientemente parametrizada, gera a *string* adequada para os três projectos anteriores, tendo o cuidado de a generalizar para qualquer outra situação do mesmo género.

2.16 Aritmética de Inteiros em Auto Lisp

Vimos que o Auto Lisp disponibiliza, para além dos operadores aritméticos usuais, um conjunto de funções matemáticas. No entanto, há que ter em conta que existem diferenças substanciais entre o significado matemático destas operações e a sua implementação no Auto Lisp.

Um primeiro problema importante tem a ver com a gama de inteiros. Em Auto Lisp, os inteiros são representados com apenas 32 bits de informação.⁵ Isto implica que apenas se conseguem representar inteiros desde

⁵Em AutoCad, a gama de inteiros é ainda mais pequena pois a sua representação só usa 16 bits, permitindo apenas representar os inteiros de -32768 a 32767. Isto não afecta o Auto Lisp, excepto quando este tem de passar inteiros para o AutoCad.

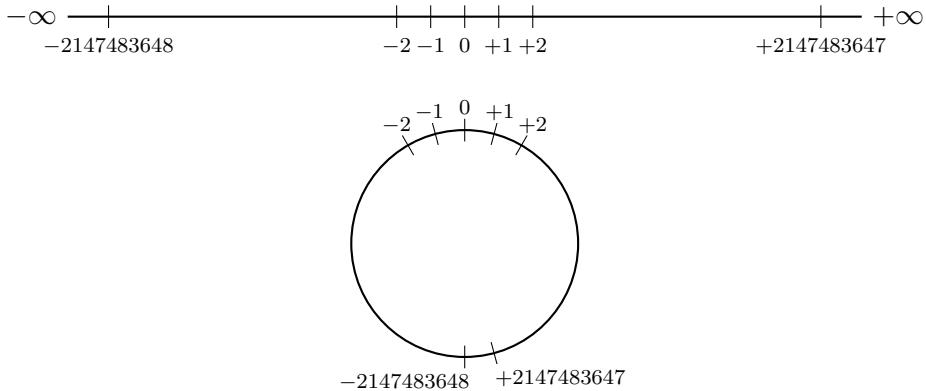


Figura 1: A recta infinita dos inteiros empregue em aritmética e o círculo dos inteiros *modulares* empregues em Auto Lisp.

-2147483647 até 2147483647 . Para tornar a situação ainda mais problemática, embora exista esta limitação, quando ela não é respeitada o Auto Lisp não emite qualquer aviso. Este comportamento é usual na maioria das linguagens de programação e está relacionado com questões de *performance*.⁶ Infelizmente, a performance tem como preço o poder originar resultados aparentemente bizarros:

```
_\$ (+ 2147483647 1)
-2147483648
```

Este resultado surge porque no Auto Lisp, na realidade, as operações aritméticas com inteiros são *modulares*. Isto implica que a sequência dos inteiros não corresponde a uma linha infinita nas duas direcção, mas antes a um círculo em que a seguir ao maior número inteiro positivo aparece o menor número inteiro negativo. Este comportamento encontra-se explicado na Figura 1.

Um segundo problema, que já referimos anteriormente, está no facto de o Auto Lisp não saber representar fracções. Isto implica que uma divisão de dois inteiros não corresponde à divisão matemática mas sim a uma operação substancialmente diferente, denominada *divisão inteira*: uma divisão em que a parte fraccional do resultado é eliminada, i.e., apenas se considera o *quociente* da divisão, eliminando-se o *resto*. Assim, $(/ 7 2)$ é 3 e não $7/2$ ou 3.5. Obviamente, isto inviabiliza algumas equivalências matemáticas óbvias. Por exemplo, embora $\frac{1}{3} \times 3 = 1$, em Auto Lisp temos:

⁶Curiosamente, a maioria dos dialectos de Lisp (mas não o Auto Lisp) *não* apresenta este comportamento, preferindo representar números inteiros com dimensão tão grande quanto for necessário.

```
_\$ (* (/ 1 3) 3)  
0
```

Finalmente, um terceiro problema que ocorre com os inteiros é que a *leitura* de um número inteiro que excede a gama dos inteiros implica a sua conversão automática para o tipo real.⁷ Neste caso, o avaliador do Auto Lisp nunca chega a “ver” o número inteiro pois ele foi logo convertido para real na leitura. Esse comportamento é visível no seguinte exemplo:

```
_\$ 1234567890  
1234567890  
_\$ 12345678901  
1.23457e+010
```

Também este aspecto do Auto Lisp pode ser fonte de comportamentos bizarros, tal como é demonstrado pelo seguinte exemplo:

```
_\$ (+ 2147483646 1)  
2147483647  
_\$ (+ 2147483647 1)  
-2147483648  
_\$ (+ 2147483648 1)  
2.14748e+009
```

É importante salientar que a conversão de inteiro para real, que é visível na última interacção, é feita logo na leitura.

Exercicio 2.28 Calcule o valor das seguintes expressões Auto Lisp:

1. $(- 1)$
2. $(- 1 1)$
3. $(- 1 1 1)$
4. $(/ 1 1)$
5. $(/ 1 2)$
6. $(/ (* 3 2) 2)$
7. $(* (/ 3 2) 2)$
8. $(/ (/ (/ 8 2) 2) 2)$
9. $(/ 8 (/ 2 (/ 2 2)))$
10. $(+ (/ 1 2) (/ 1 2))$

Exercicio 2.29 Explique o seguinte comportamento do Auto Lisp:⁸

⁷Note-se que estamos apenas a falar da leitura de números. Como já vimos, as operações aritméticas usuais não apresentam este comportamento.

⁸Este exemplo mostra que, em Auto Lisp, há um inteiro que não pode ser lido mas que pode ser produzido como resultado de operações aritméticas. É importante salientar que este comportamento verdadeiramente bizarro é exclusivo do Auto Lisp e não se verifica em mais nenhum dialecto de Lisp.

```

_§ (- -2147483647 1)
-2147483648
_§ (- -2147483648 1)
-2.14748e+009
_§ (- -2147483647 2)
2147483647

```

Exercicio 2.30 A área A de um pentágono regular inscrito num círculo de raio r é dada pela fórmula

$$A = \frac{5}{8}r^2\sqrt{10 + 2\sqrt{5}}$$

Defina uma função Auto Lisp que calcule essa área.

Exercicio 2.31 Defina uma função que calcula o volume de um elipsóide de semi-eixos a , b e c . Esse volume pode ser obtido pela fórmula $V = \frac{4}{3}\pi abc$.

2.17 Aritmética de Reais em Auto Lisp

Em relação aos reais, o seu comportamento é mais próximo do que se considera matematicamente correcto mas, ainda assim, há vários problemas com que é preciso lidar.

A gama dos reais vai desde $-4.94066 \cdot 10^{-324}$ até $1.79769 \cdot 10^{308}$. Se o Auto Lisp tentar ler um número real que excede esta gama ele é imediatamente convertido para um número especial que representa o infinito:

```

_§ 2e400
1.#INF
_§ -2e400
-1.#INF

```

Note-se que `1.#INF` ou `-1.#INF` é a forma do Auto Lisp indicar um valor que excede a capacidade de representação de reais do computador. Não é *infinito*, como se poderia pensar, mas apenas um valor excessivamente grande para as capacidades do Auto Lisp. Do mesmo modo, quando alguma operação aritmética produz um número que excede a gama dos reais, é simplesmente gerada a representação do infinito.

```

_§ 1e300
1.0e+300
_§ 1e100
1.0e+100
_§ (* 1e300 1e100)
1.#INF

```

Nestes casos, em que o resultado de uma operação é um número que excede as capacidades da máquina, dizemos que ocorreu um *overflow*.

As operações com números reais têm a vantagem de a maioria dos computadores actuais conseguirem detectar o *overflow* de reais e reagirem em conformidade (gerando um erro ou simplesmente produzindo uma representação do infinito). Como vimos anteriormente, se se tivessem usado

números inteiros, então o *overflow* não seria sequer detectado, produzindo comportamentos aparentemente bizarros, como, por exemplo, o produto de dois números positivos ser um número negativo.

Há ainda dois outros problemas importantes relacionados com reais: erros de arredondamento e redução de precisão na escrita. A título de exemplo, consideremos a óbvia igualdade matemática $(\frac{4}{3} - 1) \cdot 3 - 1 = 0$ e comparemos os resultados que se obtêm usando inteiros ou reais:

```
_\$ (- (* (- (/ 4 3) 1) 3) 1)
-1
_\$ (- (* (- (/ 4.0 3.0) 1.0) 3.0) 1.0)
-2.22045e-016
```

Como se pode ver, nem usando inteiros, nem usando reais, se consegue obter o resultado correcto. No caso dos inteiros, o problema é causado pela divisão inteira de 4 por 3 que produz 1. No caso dos reais, o problema é causado por erros de arredondamento: $4/3$ não é representável com um número finito de dígitos. Este erro de arredondamento é então propagado nas restantes operações produzindo um valor que, embora não seja zero, está muito próximo.

Obviamente, como o resultado da avaliação com reais é suficientemente pequeno, podemos convertê-lo para o tipo inteiro (aplicando-lhe uma truncatura com a função `fix`) e obtemos o resultado correcto:

```
_\$ (fix (- (* (- (/ 4.0 3.0) 1.0) 3.0) 1.0))
0
```

No entanto, esta “solução” também tem problemas. Consideremos a divisão $0.6/0.2 = 3$:

```
_\$ (/ 0.6 0.2)
3.0
_\$ (fix (/ 0.6 0.2))
2
```

O problema ocorre porque os computadores usam o sistema binário de representação e, neste sistema, não é possível representar os números 0.6 e 0.2 com um número finito de dígitos binários e, consequentemente, ocorrem erros de arredondamento. Por este motivo, o resultado da divisão, em vez de ser 3, é 2.999999999999955910790149937, embora o Auto Lisp o apresente como 3.0. No entanto, ao eliminarmos a parte fraccionária com a função `fix`, o que fica é apenas o inteiro 2.

Igualmente perturbante é o facto de, embora $0.4 \cdot 7 = 0.7 \cdot 4 = 2.8$, em Auto Lisp, $(\star 0.4 7) \neq (\star 0.7 4)$: ambas as expressões têm um valor que é escrito pelo Auto Lisp como 2.8 mas há diferentes erros de arredondamento cometidos nas duas expressões que tornam os resultados diferentes.

Note-se que, mesmo quando não há erros de arredondamento nas operações, a reduzida precisão com que o Auto Lisp escreve os resultados pode induzir-nos em erro. Por exemplo, embora internamente o Auto Lisp saiba que `(/ 1.000001 10)` produz `0.100001` como resultado, apenas escreve `0.1`.

Exercicio 2.32 Traduza a seguinte definição para Auto Lisp:

$$f(x) = x - 0.1 \cdot (10 \cdot x - 10)$$

Exercicio 2.33 Usando a função definida na questão anterior, calcule em Auto Lisp o valor das seguintes expressões e explique os resultados:

```
(f 5.1)
(f 51367.7)
(f 176498634.7)
(f 1209983553611.9)
(f 19843566622234755.9)
(f 553774558711019983333.9)
```

Exercicio 2.34 Pretende-se criar um lanço de escada com n espelhos capaz de vencer uma determinada altura a em metros. Admitindo que cada degrau tem uma altura do espelho h e uma largura do cobertor d que verificam a proporção

$$2h + d = 0.64$$

defina uma função que, a partir da altura a vencer e do número de espelhos, calcula o comprimento do lanço de escada.

2.18 Avaliação de Símbolos

Vimos que todos os outros elementos primitivos que apresentámos até agora, nomeadamente, os números e as cadeias de caracteres avaliavam para eles próprios, i.e., o valor de uma expressão composta apenas por um elemento primitivo é o próprio elemento primitivo. No caso dos símbolos, isso já não é verdade.

Lisp atribui um significado muito especial aos símbolos. Reparemos que, quando definimos uma função, o nome da função é um símbolo. Os parâmetros formais da função também são símbolos. Quando escrevemos uma combinação, o avaliador de Lisp usa a definição de função que foi associada ao símbolo que constitui o primeiro elemento da combinação. Isto quer dizer que o valor do primeiro símbolo de uma combinação é a função que lhe está associada. Admitindo que tínhamos definido a função quadrado da forma que ilustrámos na secção 2.11, podemos verificar este comportamento experimentando as seguintes expressões:

```
_\$ (quadrado 3)
9
_\$ quadrado
#<USUBR @1d4c66f4 QUADRADO>
```

Como se pode ver pelo exemplo anterior, o valor do símbolo quadrado é uma entidade que o Lisp descreve usando uma notação especial. A en-

tidade em questão é, como vimos, uma função. O mesmo comportamento ocorre para qualquer outra função pré-definida na linguagem:⁹

```
_\$ +
#<SUBR @0a87c5a8 +>
_\$ *
#<SUBR @0a87c588 *>
```

Como vimos com o `+` e o `*`, alguns dos símbolos estão pré-definidos na linguagem. Por exemplo, o símbolo `PI` também existe pré-definido com uma aproximação do valor de π .

```
_\$ pi
3.14159
```

No entanto, quando o avaliador está a avaliar o corpo de uma função, o valor de um símbolo especificado nos parâmetros da função é o argumento correspondente na invocação da função. Assim, na combinação (`quadrado 3`), depois de o avaliador saber que o valor de `quadrado` é a função por nós definida atrás e que o valor de `3` é `3`, o avaliador passa a avaliar o corpo da função `quadrado` mas assumindo que, durante essa avaliação, o nome `x`, sempre que for necessário, irá ter como valor precisamente o mesmo `3` que foi associado ao parâmetro `x`.

2.19 Símbolos como Valores

Vimos que os símbolos avaliam para aquilo a que estiverem associados no momento da avaliação. Por este motivo, os símbolos são usados para dar *nomes* aos *valores*. O que é mais interessante é que os símbolos são, eles próprios, *valores*!

Para percebermos esta característica dos símbolos vamos apresentar uma função pré-definida do Auto Lisp que nos permite saber o tipo de uma entidade qualquer: `type`. Consideremos a seguinte interacção:

```
_\$ (type 1)
INT
_\$ (type "Bom dia!")
STR
_\$ (type pi)
REAL
_\$ (type quadrado)
USUBR
_\$ (type +)
SUBR
```

Como podemos ver, a função `type` devolve-nos o tipo do seu argu-

⁹Um olhar mais atento encontrará uma pequena diferença: no caso do `quadrado`, a função associada é do tipo `USUBR` enquanto que nos casos do `+` e `*` as funções associadas são do tipo `SUBR`. A diferença entre estes dois tipos está relacionada com o facto de as `SUBRs` estarem *compiladas*, permitindo a sua invocação de forma mais eficiente. O termo `SUBR` é uma abreviatura de *subroutine*.

mento: INT para inteiros, STR para *strings*, REAL para reais, etc.

Mas o que são estes resultados—INT, STR, REAL, etc—que foram devolvidos pela função type? Que tipo de valores são? Para responder à questão, o melhor é usar a mesmíssima função type:

```
_\$ (type pi)
REAL
_\$ (type (type pi))
SYM
```

SYM é a abreviatura de *symbol*, indicando que o valor REAL que foi devolvido pela função type é um símbolo. Note-se que, contrariamente ao que acontece com o símbolo pi que está associado ao número 3.14159..., os símbolos REAL, INT, STR, etc., não estão associados a nada, eles apenas são usados como representação do nome de um determinado tipo de dados.

Se os valores devolvidos pela função type são objectos do tipo símbolo, então deverá ser possível designá-los, tal como designamos os números ou as *strings*. Mas qual será o modo de o fazermos? Uma hipótese (errada) seria escrevê-lo tal como escrevemos números ou *strings*. Acontece que isto é possível no caso dos números e *strings* pois eles avaliam para eles próprios. Já no caso dos símbolos, sabemos que não avaliam para eles próprios, antes avaliando para as entidades a que estão associados naquele momento. Assim, se quisermos designar o símbolo pi, não bastará escrevê-lo numa expressão, pois o que irá resultar após a sua avaliação não será um símbolo mas sim o número 3.14159... que é o valor desse símbolo.

Para ultrapassar este problema precisamos de, por momentos, alterar a semântica habitual que o Lisp atribui aos símbolos. Essa semântica, recordemos, é a de que o valor de um símbolo é a entidade a que esse símbolo está associado nesse momento e esse valor surge sempre que o Lisp avalia o símbolo. Para alterarmos essa semântica, precisamos de indicar ao Lisp que *não* queremos que avalie um determinado símbolo, i.e., queremos que trate o símbolo como ele é, sem o avaliar. Para isso, o Lisp disponibiliza a forma quote. Esta forma, que recebe um único argumento, tem uma semântica simplicíssima: devolve o argumento sem este ter sido avaliado. Reparemos na seguinte interacção:

```
_\$ pi
3.14159
_\$ (quote pi)
PI
_\$ (+ 1 2 3)
6
_\$ (quote (+ 1 2 3))
(+ 1 2 3)
```

Como se vê, qualquer que seja o argumento α da expressão (quote α), o valor da expressão é o próprio α sem ter sido avaliado.

A razão de ser do `quote` está associada à distinção que existe entre as frases “Escreve o teu nome” e “Escreve ‘o teu nome’”. No primeiro caso a frase tem de ser completamente interpretada para que o leitor possa dizer qual é o seu próprio nome. No segundo caso, as plicas estão lá para indicar ao leitor que ele não deve interpretar o que está entre plicas e deve limitar-se a escrever a frase “o teu nome”. As plicas servem, pois, para distinguir o que deve ser tomado como é e o que deve ser interpretado.

Para simplificar a escrita de formas que empregam o `quote`, o Lisp disponibiliza uma abreviatura que tem exactamente o mesmo significado: a plica (‘). Quando o Lisp está a fazer a *leitura* de uma expressão e encontra algo da forma ‘ α , aquilo que é lido é, na realidade, (`quote` α). Assim, temos:

```
_\$ 'pi
PI
_\$ '(+ 1 2 3)
(+ 1 2 3)
```

Exercicio 2.35 Qual é o significado da expressão ‘‘pi’?

2.20 Expressões Condicionais

Existem muitas operações cujo resultado depende da realização de um determinado teste. Por exemplo, a função matemática $|x|$, que calcula o valor absoluto do número x , equivale ao simétrico do número, se este for negativo, ou ao próprio número, caso contrário. Usando a moderna notação da matemática, temos:

$$|x| = \begin{cases} -x, & \text{se } x < 0 \\ x, & \text{caso contrário.} \end{cases}$$

Esta função terá, portanto, de *testar* se o seu argumento é negativo e escolher uma de duas alternativas: ou avalia para o próprio argumento, ou avalia para o seu simétrico.

Estas expressões, cujo valor depende de um ou mais testes a realizar previamente, são designadas *expressões condicionais*.

2.21 Expressões Lógicas

Uma expressão condicional assume a forma de “se *expressão* então … caso contrário …”. A *expressão*, cujo valor é usado para decidir se devemos usar o ramo “então” ou o ramo “caso contrário,” denomina-se *expressão lógica* e caracteriza-se por o seu valor ser interpretado como *verdade* ou *falso*. Por exemplo, a expressão lógica ($< x 0$) testa se o valor de x é menor que zero. Se for, a expressão avalia para verdade, caso contrário avalia para falso.

2.22 Valores Lógicos

Algumas linguagens de programação consideram a verdade e o falso como dois elementos de um tipo especial de dados denominado *lógico* ou *booliano*.¹⁰ Outras linguagens, como o Lisp, entendem que o facto de se considerar um valor como verdadeiro ou falso não implica necessariamente que o valor tenha de ser de um tipo de dados especial, mas apenas que a expressão condicional considera alguns dos valores como representando o verdadeiro e os restantes como o falso.

Em Lisp, as expressões condicionais consideram como falso um único valor. Esse valor é representado por `nil`, o mesmo valor que usámos anteriormente para representar uma lista vazia. Qualquer outro valor diferente de `nil` é considerado como verdadeiro. Assim, do ponto de vista de uma expressão condicional, o número 123, por exemplo, é um valor verdadeiro. Contudo, não faz muito sentido para o utilizador humano considerar um número como verdadeiro ou falso, pelo que se introduziu uma constante na linguagem para representar verdade. Essa constante representa-se por `t`. Logicamente, se `t` é diferente de `nil` e se `nil` é o único valor que representa a falsidade, então `t` representa necessariamente a verdade.

2.23 Predicados

No caso mais usual, uma expressão lógica é uma invocação de função com determinados argumentos. Nesta situação, a função usada como teste é denominada *predicado* e o valor do teste é interpretado como sendo verdadeiro ou falso. O predicado é, consequentemente, uma função que devolve apenas verdade ou falso.

Apesar da adopção dos símbolos `t` e `nil`, convém alertar que nem todos os predicados devolvem `t` ou `nil` exclusivamente. Alguns há que, quando querem indicar verdade, devolvem valores diferentes de `t` (e de `nil`, obviamente).

2.24 Predicados Aritméticos

Os *operadores relacionais* matemáticos $<$, $>$, $=$, \leq , \geq e \neq são um dos exemplos mais simples de predicados. Estes operadores compararam números entre si e permitem saber se um número é menor que outro. O seu uso em Lisp segue as regras da notação prefixa e escrevem-se, respectivamente, `<`, `>`, `=`, `<=`, `>=` e `/=`. Eis alguns exemplos:

¹⁰De George Boole, matemático inglês e inventor da álgebra da verdade e da falsidade.

```

_§ (> 4 3)
t
_§ (< 4 3)
nil
_§ (<= (+ 2 3) (- 6 1))
t

```

2.25 Operadores Lógicos

Para se poder combinar expressões lógicas entre si existem os operadores `and`, `or` e `not`. O `and` e o `or` recebem qualquer número de argumentos. O `not` só recebe um. O valor das combinações que empregam estes operadores lógicos é determinado do seguinte modo:

- O `and` avalia os seus argumentos da esquerda para a direita até que um deles seja falso, devolvendo este valor. Se nenhum for falso, o `and` devolve verdade.
- O `or` avalia os seus argumentos da esquerda para a direita até que um deles seja verdade, devolvendo este valor. Se nenhum for verdade, o `or` devolve falso.
- O `not` avalia para verdade se o seu argumento for falso e para falso em caso contrário.

Note-se que embora o significado de falso seja claro, pois corresponde necessariamente ao valor `nil`, o significado de verdade já não é tão claro, pois, desde que seja diferente de `nil`, é considerado verdade.

Exercício 2.36 Qual o valor das seguintes expressões?

1. `(and (or (> 2 3) (not (= 2 3))) (< 2 3))`
2. `(not (or (= 1 2) (= 2 3)))`
3. `(or (< 1 2) (= 1 2) (> 1 2))`
4. `(and 1 2 3)`
5. `(or 1 2 3)`
6. `(and nil 2 3)`
7. `(or nil nil 3)`

2.26 Predicados com número variável de argumentos

Uma propriedade importante dos predicados aritméticos `<`, `>`, `=`, `<=`, `>=` e `/=` é aceitarem qualquer número de argumentos. No caso em que há mais do que um argumento, o predícoado é aplicado sequencialmente aos pares de argumentos. Assim, $(< e_1 e_2 e_3 \dots e_{n-1} e_n)$ é equivalente a escrever $(\text{and} ((< e_1 e_2) (< e_2 e_3) \dots (< e_{n-1} e_n)))$. Este comportamento é visível nos seguintes exemplos.

```
_\$ (< 1 2 3)
T
_\$ (< 1 2 2)
nil
```

Um caso particular que convém ter em atenção é que, embora a expressão ($= e_1 e_2 \dots e_n$) teste se os elementos $e_1 e_2 \dots e_n$ são todos iguais, ($/= e_1 e_2 \dots e_n$) **não** testa se os elementos $e_1 e_2 \dots e_n$ são todos diferentes: uma vez que o teste é aplicado sucessivamente a pares de elementos, é perfeitamente possível que existam dois elementos iguais, desde que não sejam consecutivos. Esse comportamento é visível no seguinte exemplo:¹¹

```
_\$ (/= 1 2 3)
T
_\$ (/= 1 2 1)
T
```

2.27 Predicados sobre Cadeias de Caracteres

Na verdade, os operadores $<$, $>$, $=$, $<=$, $>=$ e $/=$ não se limitam a operarem sobre números: eles aceitam também cadeias de caracteres como argumentos, fazendo uma comparação *lexicográfica*: os argumentos são comparados carácter a carácter enquanto forem iguais. Quando são diferentes, a ordem léxicográfica do primeiro carácter diferente nas duas *strings* determina o valor lógico da relação. Se a primeira *string* “acabar” antes da segunda, considera-se que é “menor,” caso contrário, é “maior.”

```
_\$ (= "pois" "pois")
T
_\$ (= "pois" "poisar")
nil
_\$ (< "pois" "poisar")
T
_\$ (< "abcd" "abce")
T
```

2.28 Reconhecedores

Para além dos operadores relacionais, existem muitos outros predicados em Lisp, como por exemplo o `zerop`, que testa se um número é zero:

¹¹Outros dialectos de Lisp apresentam um comportamento diferente para esta operação. Em Common Lisp, por exemplo, o predicado `/=` testa se, de facto, os argumentos são todos diferentes.

```
_\$ (zerop 1)
nil
_\$ (zerop 0)
t
```

O facto de `zerop` terminar com a letra “p” deve-se a uma convenção adoptada nalguns dialectos de Lisp segundo a qual os predicados cujo nome seja uma ou mais palavras devem ser distinguidos das restantes funções através da concatenação da letra “p” (de *Predicate*) ao seu nome. Outros dialectos preferem terminar os nomes dos predicados com um ponto de interrogação precisamente porque, na prática, a invocação de um predicado corresponde a uma pergunta que fazemos. Por motivos históricos, nem todos os predicados pré-definidos no Auto Lisp seguem estas convenções. No entanto, nos predicados que definirmos devemos ter o cuidado de as seguir.

Note-se que o operador `zerop` serve para reconhecer um elemento em particular (o zero) de um tipo de dados (os números). Este género de predicados denominam-se de *reconhecedores*.

2.29 Reconhecedores Universais

Um outro conjunto importante de predicados são os denominados *reconhecedores universais*. Estes não reconhecem elementos particulares de um tipo de dados mas sim todos os elementos de um particular tipo de dados. Um reconhecedor universal aceita qualquer tipo de valor como argumento e devolve verdade se o valor é do tipo pretendido.

Por exemplo, para sabermos se uma determinada entidade é um número, podemos empregar o predicado `numberp`:

```
_\$ (numberp 1)
T
_\$ (numberp nil)
nil
_\$ (numberp "Dois")
nil
```

Para a maioria dos tipos, o Auto Lisp não providencia nenhum reconhecedor universal, antes preferindo usar a função genérica `type` que devolve o nome do tipo como símbolo. Como vimos na seccção 2.19, temos:

```

_§ (type pi)
REAL
_§ (type 1)
INT
_§ (type "Ola")
STR
_§ (type quadrado)
USUBR
_§ (type +)
SUBR

```

Obviamente, nada nos impede de definir os reconhecedores universais que pretendermos. À semelhança do predicado `numberp` podemos definir os seus subcasos `integerp` e `realp`:

```

(defun integerp (obj)
  (= (type obj) 'int))

(defun realp (obj)
  (= (type obj) 'real))

```

É igualmente trivial definir um reconhecedor universal de *strings* e outro para funções (pré-definidas ou não):

```

(defun stringp (obj)
  (= (type obj) 'str))

(defun functionp (obj)
  (or (= (type obj) 'subr)
      (= (type obj) 'usubr)))

```

2.30 Exercícios

Exercicio 2.37 O que é uma expressão condicional? O que é uma expressão lógica?

Exercicio 2.38 O que é um valor lógico? Quais são os valores lógicos empregues em Auto Lisp?

Exercicio 2.39 O que é um predicado? Dê exemplos de predicados em Auto Lisp.

Exercicio 2.40 O que é um operador relacional? Dê exemplos de operadores relacionais em Auto Lisp.

Exercicio 2.41 O que é um operador lógico? Quais são os operadores lógicos que conhece em Auto Lisp?

Exercicio 2.42 O que é um reconhecedor? O que é um reconhecedor universal? Dê exemplos em Auto Lisp.

Exercicio 2.43 Traduza para Lisp as seguintes expressões matemáticas:

1. $x < y$

2. $x \leq y$
3. $x < y \wedge y < z$
4. $x < y \wedge x < z$
5. $x \leq y \leq z$
6. $x \leq y < z$
7. $x < y \leq z$

2.31 Selecção

Se observarmos a definição matemática da função valor absoluto

$$|x| = \begin{cases} -x, & \text{se } x < 0 \\ x, & \text{caso contrário.} \end{cases}$$

constatamos que ela emprega uma expressão condicional da forma

$$\begin{cases} \text{expressão consequente,} & \text{se expressão lógica} \\ \text{expressão alternativa,} & \text{caso contrário.} \end{cases}$$

que, em linguagem natural, se traduz para “se *expressão lógica*, então *expressão consequente*, caso contrário, *expressão alternativa*.”

A avaliação de uma expressão condicional é feita através da avaliação da *expressão lógica* que, se produzir verdade, implica a avaliação da *expressão consequente* e, se produzir falso, implica a avaliação da *expressão alternativa*.

No caso da linguagem Lisp, a notação usada para descrever expressões condicionais é ainda mais simples do que na matemática pois baseia-se num simples operador, o `if`, denominado operador de *selecção* por permitir seleccionar entre duas alternativas. A sintaxe do operador `if` é a seguinte:

```
(if expressão lógica
    expressão consequente
    expressão alternativa)
```

O valor de uma expressão condicional que usa o operador `if` é obtido da seguinte forma:

1. A *expressão lógica* é avaliada.
2. Se o valor obtido da avaliação anterior é verdade, o valor da combinação é o valor da *expressão consequente*.
3. Caso contrário (ou seja, o valor obtido da avaliação anterior é falso), o valor da combinação é o valor da *expressão alternativa*.

Este comportamento, em tudo idêntico ao que teríamos em Matemática, pode ser confirmado pelos seguintes exemplos:

```

_\$ (if (> 3 2)
      1
      2)
1
_\$ (if (> 3 4)
      1
      2)
2

```

Empregando o `if` podemos agora definir a função valor absoluto por simples tradução da definição matemática para Auto Lisp:

```
(defun abs (x)
  (if (< x 0)
      (- x)
      x))
```

O objectivo fundamental do operador `if` é permitir definir funções cujo comportamento depende de uma ou mais condições. Por exemplo, consideremos a função `max` que recebe dois números como argumentos e devolve o maior deles. Para definirmos esta função apenas precisamos de testar se o primeiro argumento é maior que o segundo. Se for, a função devolve o primeiro argumento, caso contrário devolve o segundo. Com base neste raciocínio, podemos escrever:

```
(defun max (x y)
  (if (> x y)
      x
      y))
```

Um outro exemplo mais interessante ocorre com a função matemática *sinal* `sgn`, também conhecida como função *signum* (“sinal,” em Latim). Esta função pode ser vista como a função *dual* da função valor absoluto pois tem-se sempre $x = \text{sgn}(x)|x|$. A função sinal é definida por

$$\text{sgn } x = \begin{cases} -1 & \text{se } x < 0 \\ 0 & \text{se } x = 0 \\ 1 & \text{caso contrário} \end{cases}$$

Em linguagem natural, dizemos que se x for negativo, o valor de $\text{sgn } x$ é -1 , caso contrário, se x for 0 , o valor é 0 , caso contrário, o valor é 1 . Isto mostra que, na verdade, a definição anterior emprega duas expressões condicionais encadeadas, da forma:

$$\text{sgn } x = \begin{cases} -1 & \text{se } x < 0 \\ \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{caso contrário} \end{cases} & \text{caso contrário} \end{cases}$$

Assim sendo, para definirmos esta função em Lisp, temos de empregar dois `ifs`:

```
(defun signum (x)
  (if (< x 0)
      -1
      (if (= x 0)
          0
          1)))
```

2.32 Seleção Múltipla—A Forma cond

Quando uma expressão condicional necessita de vários ifs encadeados, é possível que o código comece a ficar mais difícil de ler. Neste caso, é preferível usar uma outra forma que torne a definição da função mais legível. Para evitar muitos ifs encadeados o Lisp providencia uma outra forma denominada `cond` cuja sintaxe é a seguinte:

```
(cond (expr0,0 expr0,1 ... expr0,n)
      (expr1,0 expr1,1 ... expr1,m)
      ...
      (exprk,0 exprk,1 ... exprk,p))
```

O `cond` aceita qualquer número de argumentos. Cada argumento é denominado *cláusula* e é constituído por uma lista de expressões. A semântica do `cond` consiste em avaliar sequencialmente a primeira expressão $expr_{i,0}$ de cada cláusula até encontrar uma cujo valor seja verdade. Nesse momento, o `cond` avalia todas as restantes expressões dessa cláusula e devolve o valor da última. Se nenhuma das cláusulas tiver uma primeira expressão que avalie para verdade, o `cond` devolve `nil`. Se a cláusula cuja primeira expressão é verdade não contiver mais expressões, o `cond` devolve o valor dessa primeira expressão.

É importante perceber que os parêntesis que envolvem as cláusulas não correspondem a nenhuma combinação: eles simplesmente fazem parte da sintaxe do `cond` e são necessários para separar as cláusulas umas das outras.

A pragmática usual para a escrita de um `cond` (em especial quando cada cláusula contém apenas duas expressões) consiste em alinhar as expressões umas debaixo das outras.

Usando o `cond`, a função *sinal* pode ser escrita de forma mais simples:

```
(defun signum (x)
  (cond ((< x 0)
         -1)
        ((= x 0)
         0)
        (t
         1)))
```

Note-se, no exemplo anterior, que a última cláusula do `cond` tem, como expressão lógica, o símbolo `t`. Como já vimos, este símbolo representa a

verdade, pelo que a sua presença garante que ela será avaliada no caso de nenhuma das cláusulas anteriores o ter sido. Neste sentido, cláusula da forma `(t ...)` representa um “em último caso”

Exercicio 2.44 Qual o significado de `(cond (expr1 expr2))`?

Exercicio 2.45 Qual o significado de `(cond (expr1) (expr2))`?

Exercicio 2.46 Defina uma função `soma-maiores` que recebe três números como argumento e determina a soma dos dois maiores.

Exercicio 2.47 Defina a função `max3` que recebe três números como argumento e calcula o maior entre eles.

Exercicio 2.48 Defina a função `segundo-maior` que recebe três números como argumento e devolve o segundo maior número, i.e., que está entre o maior e o menor.

3 Modelação Geométrica

Vimos, nas secções anteriores, alguns dos tipos de dados pré-definidos em Auto Lisp. Em muitos casos, esses tipos de dados são os necessários e suficientes para nos permitir criar os nossos programas. Noutros casos, será necessário introduzirmos novos tipos de dados. Nesta secção iremos estudar o modo de o fazermos e exemplificá-lo-emos com um tipo de dados que nos será particularmente útil para a modelação de entidades geométricas: coordenadas.

3.1 Coordenadas

A Arquitectura pressupõe a localização de elementos no espaço. Essa localização expressa-se em termos do que se designa por *coordenadas*: cada coordenada é um número e uma sequência de coordenadas identifica univocamente um ponto no espaço. A Figura 2 esquematiza uma possível sequência de coordenadas (x, y, z) que identificam o ponto P num espaço tridimensional. Diferentes sistemas de coordenadas são possíveis e, no caso da Figura 2, estamos a empregar aquele que se designa por sistema de coordenadas *rectangulares*, também conhecido por sistema de coordenadas *Cartesianas*, em honra do seu inventor: René Descartes.¹²

Se queremos trabalhar com coordenadas em Auto Lisp, temos de saber como formar sequências de números. Vimos, nas secções anteriores, que o Auto Lisp sabe trabalhar com números. Vamos agora mostrar que também sabe trabalhar com sequências. Para isso, vamos introduzir o conceito de *lista*.

¹²Descartes foi um importantíssimo filósofo Francês do século XVII, autor da famosa frase “*Cogito ergo sum*” (penso, logo existo) e de inúmeras contribuições na Matemática e na Física.

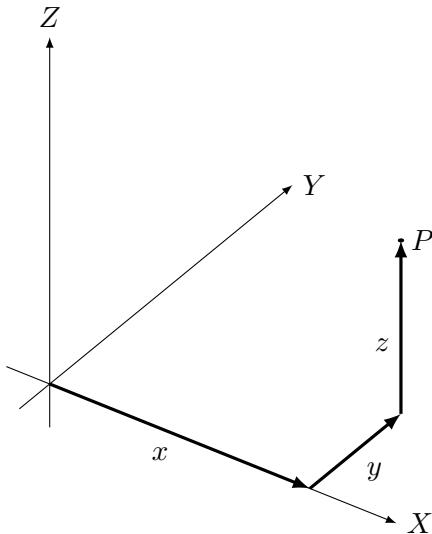


Figura 2: Coordenadas Cartesianas de um ponto no espaço.

3.2 Listas

Uma lista é uma sequência de elementos. Cada elemento é um valor de um qualquer tipo de dados, como seja um número, uma cadeia de caracteres, ou mesmo uma outra lista. Em Lisp, podemos criar listas usando a função pré-definida `list`. Eis um exemplo da criação de uma lista de números:

```
_\$ (list 1 2 3 4)
(1 2 3 4)
```

Note-se que a função `list`, à semelhança de outras funções como a adição e multiplicação, recebe qualquer número de argumentos, agrupando-os todos numa lista. Note-se ainda que quando o Lisp escreve a lista resultante, emprega a mesma sintaxe que é usada para a escrita de combinações, i.e., separando os elementos da lista com um espaço e envolvendo-os num par de parêntesis. Existe, contudo, uma exceção a esta regra: uma lista *vazia*, i.e., uma lista sem elementos, é escrita usando o símbolo `nil`, exatamente o mesmo que é usado para representar a falsidade. O nome `nil` é uma contracção da palavra *nihil* que, em latim, significa o vazio. É isso que acontece na seguinte interacção:

```
_\$ (list)
nil
```

Como se pode ver no exemplo anterior, para construirmos uma lista sem elementos basta invocar a função `list` *sem* argumentos. Quando necessário, podemos reconhecer a lista vazia empregando o predicado `null`.

A função `null` devolve verdade quando aplicada a uma lista vazia e falso em qualquer outro caso:

```
_\$ (null (list 1 2 3))  
nil  
_\$ (null (list))  
T
```

Uma vez que a construção de uma lista é feita usando uma função, a sua invocação segue as mesmas regras de avaliação de todas as outras invocações de funções, ou seja, as expressões que constituem os argumentos são avaliadas e são os resultados dessas avaliações que são usados como elementos da lista. Assim, temos:

```
_\$ (list (+ 1 2) (* 3 4))  
(3 12)  
_\$ (list 1 "dois" (+ 1 2))  
(1 "dois" 3)  
_\$ (list (area-circulo 10) (area-triangulo 20 30))  
(314.159 300)
```

A partir do momento em que temos uma lista de entidades, podemos estar interessados em saber quais são os elementos dessa lista. O Lisp disponibiliza várias formas de o fazermos. Uma das mais simples é empregar a função pré-definida `nth` (abreviatura de inglesa de *n-ésimo*) que recebe a posição do elemento a que queremos aceder e a lista que o contém.¹³ Para usarmos correctamente esta função é preciso termos em conta que o primeiro elemento da lista ocupa a posição zero. A seguinte interacção ilustra este comportamento:

```
_\$ (nth 0 (list (+ 1 2) (* 3 4)))  
3  
_\$ (nth 1 (list 1 "dois"))  
"dois"
```

Exercicio 3.1 Qual o resultado da avaliação da expressão `(list (list 1 2) 3)`?

Exercicio 3.2 Qual o resultado da avaliação da expressão `(nth 0 (list (list 1 2) 3))`?

Exercicio 3.3 Qual o resultado da avaliação da expressão `(nth 1 (list (list 1 2) 3))`?

3.3 Operações com Coordenadas

A partir do momento em que sabemos construir listas, podemos criar coordenadas e podemos definir operações sobre essas coordenadas. Para criarmos coordenadas tridimensionais podemos simplesmente juntar numa lista os três números das coordenadas (x, y, z). Por exemplo, o ponto do espaço cartesiano $(1, 2, 3)$ pode ser construído através de:

¹³Mais à frente iremos ver que existem formas mais eficientes de aceder aos elementos de uma lista. Por agora, e apenas por motivos pedagógicos, vamos limitar-nos à função `nth`.

```
_\$ (list 1 2 3)
(1 2 3)
```

Uma vez que estamos a fazer uma lista que contém, primeiro, a coordenada x , a seguir, a coordenada y e, por último, a coordenada z , podemos obter estes valores usando a função `nth` com os índices 0, 1, e 2, respectivamente.

Para melhor percebermos a utilização destas funções, imaginemos que pretendemos definir uma operação que mede a distância d entre os pontos $P_0 = (x_0, y_0, z_0)$ e $P_1 = (x_1, y_1, z_1)$. Essa distância é determinada pela fórmula

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

e a sua tradução para Auto Lisp será:

```
(defun distancia (p0 p1)
  (sqrt (+ (quadrado (- (nth 0 p1) (nth 0 p0)))
            (quadrado (- (nth 1 p1) (nth 1 p0)))
            (quadrado (- (nth 2 p1) (nth 2 p0))))))
```

Podemos agora experimentar a função com um caso concreto:

```
_\$ (distancia (list 1 2 3) (list 8 6 4))
8.12404
```

3.4 Abstracção de Dados

Embora tenhamos pensado na operação `distancia` como uma função que recebe as coordenadas de dois pontos, quer quando observamos a definição da função, quer quando observamos a sua utilização subsequente, o que vemos é a invocação das operações `list` e `nth` e, consequentemente, nada nos diz que a função esteja a lidar com coordenadas. Embora o conceito de coordenadas esteja presente nos nossos pensamentos, os nossos programas apenas mostram que estamos a construir e a manipular listas.

Esta diferença entre os conceitos que temos na nossa cabeça e os que empregamos na programação torna-se ainda mais evidente quando pensamos noutras entidades que, tal como as coordenadas, também agrupam elementos mais simples. Uma data, por exemplo, define-se como uma sequência de números inteiros: o dia, o mês, e o ano. À semelhança do que fizemos com as coordenadas, também poderíamos criar datas em Auto Lisp empregando listas. À medida que formos implementando em Auto Lisp as funções que manipulam estes conceitos, a utilização sistemática das funções `list` e `nth` fará com que seja cada vez mais difícil perceber qual ou quais os tipos de dados a que se destina uma determinada função. De facto, a partir apenas das funções `list` e `nth` não podemos saber se estamos a lidar com coordenadas, datas, ou qualquer outro tipo de dados que tenhamos implementado em termos de listas.

Para resolvemos este problema é necessário preservarmos no Auto Lisp o conceito original que pretendemos implementar. Para isso, temos de *abstrair* a utilização que fazemos das listas, “escondendo-a” no interior de funções que representem explicitamente os conceitos originais. Para exemplificar esta abordagem vamos reconsiderar o conceito de coordenadas cartesianas e vamos introduzir funções apropriadas para esconder a utilização das listas.

Assim, para construirmos as coordenadas (x, y, z) a partir dos seus componentes x, y , e z , ao invés de usarmos directamente a função `list` vamos definir uma nova função que vamos denominar de `xyz`:¹⁴

```
(defun xyz (x y z)
  (list x y z))
```

A construção de coordenadas por intermédio da função `xyz` é apenas o primeiro passo para abstrairmos a utilização das listas. O segundo passo é a criação de funções que acedem às componentes x, y , e z , das coordenadas (x, y, z) . Para isso, vamos definir, respectivamente, as funções `cx`, `cy`, e `cz`, como abreviaturas de, respectivamente, *coordenada x*, *coordenada y*, e *coordenada z*:

```
(defun cx (c)
  (nth 0 c))

(defun cy (c)
  (nth 1 c))

(defun cz (c)
  (nth 2 c))
```

As funções `xyz`, `cx`, `cy`, e `cz` constituem uma *abstracção* das coordenadas que nos permitem manipular coordenadas sem termos de pensar na sua implementação em termos de listas. Esse facto torna-se evidente quando reescrivemos a função `distancia` usando estas novas funções:

```
(defun distancia (p0 p1)
  (sqrt (+ (quadrado (- (cx p1) (cx p0)))
            (quadrado (- (cy p1) (cy p0))))
        (quadrado (- (cz p1) (cz p0))))))
```

Reparemos que, contrariamente ao que acontecia com a versão anterior, a leitura desta função dá agora uma ideia clara do que ela faz. Ao invés de termos de pensar em termos da posição dos elementos numa lista, vemos que a função lida com as coordenadas x, y , e z . A utilização da função também mostra claramente que o que ela está a manipular são coordenadas:

¹⁴Naturalmente, podemos considerar outro nome igualmente apropriado como, por exemplo, *coordenadas-cartesianas*. Contudo, como é de esperar que tenhamos frequentemente de criar coordenadas, é conveniente que adoptemos um nome suficientemente curto e, por isso, vamos adoptar o nome `xyz`.

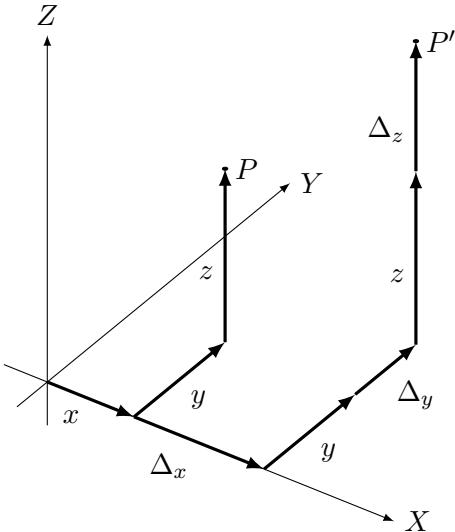


Figura 3: O ponto P' como resultado da translação do ponto $P = (x, y, z)$ de Δ_x no eixo do X , Δ_y no eixo do Y , e Δ_z no eixo do Z .

```
_\$ (distancia (xyz 1 2 3) (xyz 8 6 4))
8.12404
```

A introdução das operações de coordenadas `xyz`, `cx`, `cy`, e `cz` não só torna mais claro o significado dos nossos programas como facilita bastante a definição de novas funções. Agora, ao invés de termos de nos lembrar que as coordenadas são listas cujo primeiro elemento é a coordenada x , cujo segundo elemento é a coordenada y , e cujo terceiro elemento é a coordenada z , basta-nos pensar nas operações básicas de coordenadas e definir as funções que pretendemos em termos delas. Por exemplo, imaginemos que pretendemos definir uma operação que calcula a posição de um ponto após uma translacção, expressa em termos das componentes ortogonais Δ_x , Δ_y , e Δ_z , tal como está apresentado na Figura 3. Como se pode ver nesta Figura, sendo $P = (x, y, z)$, então teremos $P' = (x + \Delta_x, y + \Delta_y, z + \Delta_z)$. Para simplificar o uso desta função, vamos denominá-la `+xyz`. Naturalmente, ela precisa de receber, como parâmetros, o ponto de partida P e os incrementos Δ_x , Δ_y , e Δ_z , que iremos denominar `dx`, `dy`, e `dz`, respectivamente.

A definição da função fica então:

```
(defun +xyz (p dx dy dz)
  (xyz (+ (cx p) dx)
        (+ (cy p) dy)
        (+ (cz p) dz)))
```

Uma vez que esta função recebe coordenadas como argumento e produz coordenadas como resultado, ela constitui outra importante adição ao

conjunto de operações disponíveis para lidar com coordenadas. Naturalmente, podemos usar a função `+xyz` para definir novas funções como, por exemplo, os casos particulares de deslocamento horizontal e vertical que se seguem:

```
(defun +x (p dx)
  (+xyz p dx 0 0))

(defun +y (p dy)
  (+xyz p 0 dy 0))

(defun +z (p dz)
  (+xyz p 0 0 dz))
```

Igualmente útil é a definição de funções que operam deslocamentos “em diagonal” ao longo dos vários planos ortogonais XY , XZ , e YZ :

```
(defun +xy (p dx dy)
  (+xyz p dx dy 0))

(defun +xz (p dx dz)
  (+xyz p dx 0 dz))

(defun +yz (p dy dz)
  (+xyz p 0 dy dz))
```

O efeito destas funções é visível na Figura 4.

Exercício 3.4 Defina a função `ponto-medio` que calcula as coordenadas tridimensionais do ponto médio entre dois outros pontos P_0 e P_1 descritos também pelas suas coordenadas tridimensionais.

Exercício 3.5 Defina a função `=c` que compara as coordenadas de dois pontos e devolve verdade apenas quando são coincidentes. Note que dois pontos são coincidentes quando as coordenadas x , y e z dos dois pontos forem iguais.

3.5 Tipos Abstractos

A utilização das operações `xyz`, `cx`, `cy` e `cz` permite-nos definir um novo tipo de dados—as coordenadas cartesianas tridimensionais—e, mais importante, permite-nos *abstrair* a implementação desse tipo de dados, i.e., esquecer o modo como está implementado. Por este motivo, denomina-se este novo tipo de dados por *tipo abstracto*. É abstracto porque apenas existe no nosso pensamento. Para o Lisp, como vimos, as coordenadas são apenas listas de números. Essa particular combinação de dados denomina-se *representação* dos elementos do tipo.

Um tipo abstracto é caracterizado pelas suas operações e estas podem ser divididas em dois conjuntos fundamentais: os *construtores* que, a partir de argumentos de tipos apropriados, produzem elementos do tipo abs-

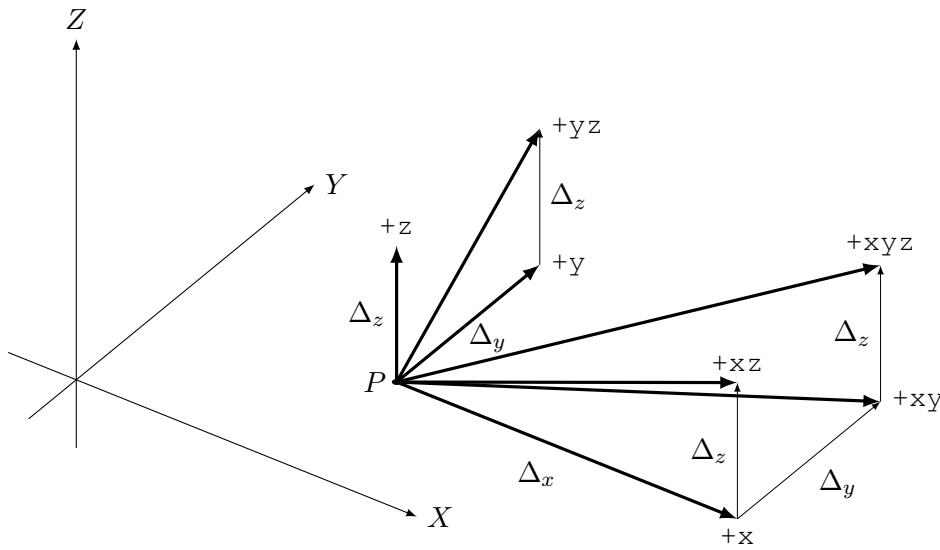


Figura 4: As translacções realizadas pelas funções $+x$, $+y$, $+z$, $+xy$, $+xz$, $+yz$, e $+xyz$ a partir de um ponto arbitrário P e dos deslocamentos Δ_x , Δ_y , e Δ_z .

tracto, e os *selectores* que, a partir de um elemento do tipo abstracto, produzem os seus constituintes. Existem ainda outras categorias de operações mas, por agora, iremos concentrarmo-nos apenas nestas duas.

No caso das coordenadas cartesianas tridimensionais, o conjunto dos construtores apenas contém a função xyz , enquanto que o conjunto dos selectores contém as funções cx , cy e cz . Para um tipo abstracto, a relação entre os construtores e os selectores é crucial pois eles têm de ser consistentes entre si. Matematicamente, essa consistência é assegurada por *equações* que, no caso presente, se podem escrever da seguinte forma:

$$\begin{aligned} (cx \ (xyz \ x \ y \ z)) &= x \\ (cy \ (xyz \ x \ y \ z)) &= y \\ (cz \ (xyz \ x \ y \ z)) &= z \end{aligned}$$

Estas equações asseguram que, se modificarmos a representação de coordenadas, mas mantivermos a consistência entre os construtores e os selectores, manter-se-á também o correcto funcionamento do tipo abstracto.

Exercício 3.6 O que é um *construtor* de um tipo abstracto?

Exercício 3.7 O que é um *selector* de um tipo abstracto?

Exercício 3.8 O que é a *representação* de um tipo abstracto?

3.6 Coordenadas Bidimensionais

Tal como as coordenadas tridimensionais localizam pontos no espaço, as coordenadas bidimensionais localizam pontos no plano. A questão que se coloca é: qual plano? Do ponto de vista matemático, a questão não é relevante pois é perfeitamente possível pensar em geometria no plano sem pensar no plano propriamente dito, mas quando tentamos visualizar essa geometria no AutoCad, que é um programa de modelação tridimensional, inevitavelmente temos de pensar na localização desse plano. Se nada for dito em contrário, o AutoCad, como aliás todos os outros programas de CAD tridimensionais, considera que o plano bidimensional corresponde ao plano XY , localizado na cota Z igual a zero. Assim sendo, é razoável assumirmos o mesmo e, consequentemente, vamos considerar a coordenada bidimensional (x, y) como uma notação simplificada da coordenada tridimensional $(x, y, 0)$.

Usando esta simplificação, podemos definir um construtor de coordenadas bidimensionais à custa do construtor de coordenadas tridimensionais:

```
(defun xy (x y)
  (xyz x y 0))
```

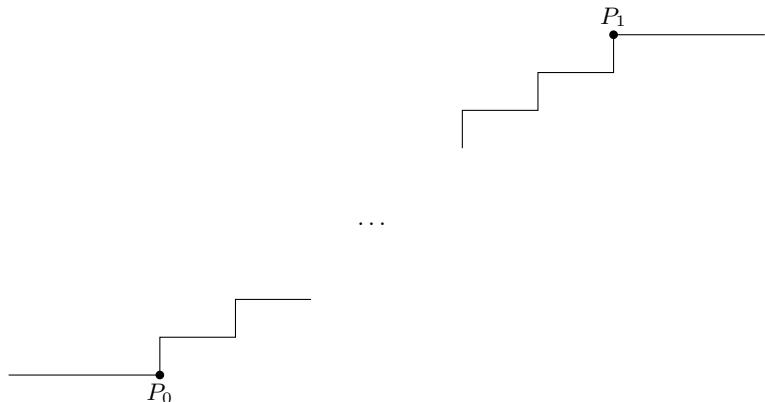
Uma das vantagens da definição de coordenadas bidimensionais como um caso particular das coordenadas tridimensionais é o facto de os selectores `cx`, `cy`, e `cz` serem automaticamente aplicáveis a coordenadas bidimensionais e, por arrasto, também o são as operações de coordenadas que definimos anteriormente, como a `distancia` e as translações `+x`, `+y`, `+z`, `+xy`, `+xz`, `+yz`, e `+xyz`.

Exercicio 3.9 Dado um ponto $P_0 = (x_0, y_0)$ e uma recta definida por dois pontos $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$, a distância mínima d do ponto P_0 à recta obtém-se pela fórmula:

$$d = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

Defina uma função denominada `distancia-ponto-recta` que, dadas as coordenadas dos pontos P_0 , P_1 e P_2 , devolve a distância mínima de P_0 à recta definida por P_1 e P_2 .

Exercicio 3.10 Sabendo que o espelho máximo admissível para um degrau é de 0.18m, defina uma função que calcula o número mínimo de espelhos que a escada representada no seguinte esquema necessita para fazer a ligação entre os pontos P_0 e P_1 .



3.7 Coordenadas em AutoCad

Como mostrámos nas secções anteriores, a representação de coordenadas (x, y, z) que adoptámos baseia-se na utilização de listas contendo os valores de x , y , e z dispostos por ordem. Para além de esta representação ser simples, tem ainda outra enorme vantagem: é totalmente compatível com a forma como o AutoCad espera receber coordenadas.

Seria então de esperar que já existissem pré-definidas em Auto Lisp as operações que lidam com coordenadas neste formato mas, na verdade, tal não acontece porque na altura em que o Auto Lisp foi criado a *teoria dos tipos abstractos* ainda não tinha sido efectivamente posta em prática. Na verdade, se fizermos uma análise dos programas Auto Lisp existentes, incluindo programas desenvolvidos recentemente, iremos constatar que a prática usual é a manipulação directa da representação, saltando por cima de qualquer definição de tipos abstractos. Em particular, a manipulação de coordenadas é geralmente feita usando directamente as operações do tipo lista.

Embora sejamos grandes defensores do respeito pela pragmática do Auto Lisp, neste caso particular vamos adoptar uma abordagem diferente: por motivos de clareza dos programas, de facilidade da sua compreensão e de facilidade da sua correcção, vamos empregar as operações do tipo abstracto coordenadas, nomeadamente o construtor `xyz` e os selectores `cx`, `cy`, e `cz` e vamos evitar aceder directamente à representação das coordenadas, i.e., vamos evitar usar as operações de listas para manipular coordenadas. Isso não impede que usemos essas funções para outros fins, em particular, para manipular listas. Uma vez que as coordenadas estão implementadas usando listas, esta distinção poderá parecer confusa mas, na verdade, não é: as coordenadas *não são* listas embora a *representação* das coordenadas seja uma lista.

Naturalmente, quando o leitor consultar programas escritos por outros programadores de Auto Lisp deverá ter em conta estas subtilezas e deverá

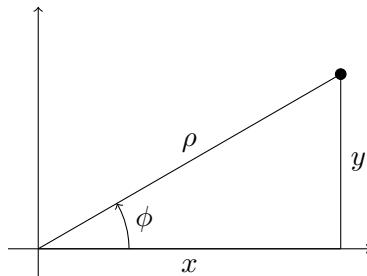


Figura 5: Coordenadas rectangulares e polares.

conseguir perceber se uma dada lista presente num programa significa coordenadas ou se significa outro tipo abstracto qualquer.

3.8 Coordenadas Polares

Apesar do AutoCad exigir que as coordenadas estejam descritas no sistema Cartesiano, nada nos impede de usar outros sistemas de coordenadas, desde que tenhamos o cuidado de, antes da passagem de coordenadas para o AutoCad, proceder à sua conversão para o sistema Cartesiano.

No caso bidimensional, um dos sistemas mais úteis é o de *coordenadas polares*.

Tal como representado da Figura 5, uma posição no plano bidimensional é descrita, em coordenadas rectangulares, pelos números x e y —significando, respectivamente, a *abcissa* e a *ordenada*—enquanto que a mesma posição em coordenadas polares é descrita pelos números ρ e ϕ —significando, respectivamente, o *raio vector* (também chamado *módulo*) e o *ângulo polar* (também chamado *argumento*). Com a ajuda da trigonometria e do teorema de Pitágoras conseguimos facilmente converter de coordenadas polares para coordenadas rectangulares:

$$\begin{cases} x = \rho \cos \phi \\ y = \rho \sin \phi \end{cases}$$

ou de coordenadas rectangulares para coordenadas polares:

$$\begin{cases} \rho = \sqrt{x^2 + y^2} \\ \phi = \arctan \frac{y}{x} \end{cases}$$

Com base nas equações anteriores, podemos definir o construtor de coordenadas polares `pol` (abreviatura de “polar”) que contrói coordenadas a partir da sua representação polar simplesmente convertendo-a para a representação rectangular equivalente.

```
(defun pol (ro fi)
  (xy (* ro (cos fi))
       (* ro (sin fi))))
```

Assim sendo, o tipo abstracto coordenadas polares fica representado em termos do tipo coordenadas rectangulares. Por este motivo, os selectores do tipo coordenadas polares—a função `pol-ro` que nos permite obter o módulo ρ e a função `pol-fi` que nos permite obter o argumento ϕ —terão de usar os selectores de coordenadas rectangulares, i.e., `cx` e `cy`:

```
(defun pol-ro (c)
  (sqrt (+ (quadrado (cx c)) (quadrado (cy c)))))

(defun pol-fi (c)
  (atan (cy c) (cx c)))
```

Eis alguns exemplos do uso destas funções:¹⁵

```
_\$ (pol 1 0)
(1.0 0.0 0)
_\$ (pol (sqrt 2) (/ pi 4))
(1.0 1.0 0)
_\$ (pol 1 (/ pi 2))
(6.12323e-017 1.0 0)
_\$ (pol 1 pi)
(-1.0 1.22465e-016 0)
```

Uma outra operação bastante útil é a que, dado um ponto $P = (x, y)$ e um “vector” com origem em P e descrito em coordenadas polares por uma distância ρ e um ângulo ϕ , devolve o ponto localizado na extremidade destino do vector, tal como é visualizado na Figura 6. A trigonometria permite-nos facilmente concluir que as coordenadas do ponto destino são dadas por $P' = (x + \rho \cos \phi, y + \rho \sin \phi)$.

A tradução directa da função para Lisp é:

```
(defun +pol (p ro fi)
  (+xy p
        (* ro (cos fi))
        (* ro (sin fi))))
```

Como exemplos de utilização, temos:

```
_\$ (+pol (xy 1 2) (sqrt 2) (/ pi 4))
(2.0 3.0 0)
_\$ (+pol (xy 1 2) 1 0)
(2.0 2.0 0)
_\$ (+pol (xy 1 2) 1 (/ pi 2))
(1.0 3.0 0)
```

¹⁵Note-se, nestes exemplos, que alguns valores das coordenadas não são zero, como seria expectável, mas sim valores muito próximos de zero, que resultam de erros de arredondamento. Note-se também que, como estamos a usar coordenadas bidimensionais, a coordenada z é sempre igual a zero.

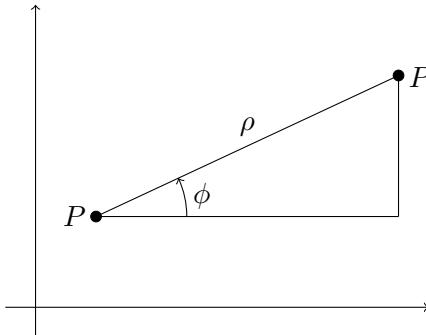


Figura 6: O deslocamento de um ponto em coordenadas polares.

Exercicio 3.11 A função `=c` definida no exercício 3.5 compara as coordenadas de dois pontos e devolve verdade apenas quando são coincidentes. No entanto, se tivermos em conta que as operações numéricicas podem produzir erros de arredondamento, é perfeitamente possível que duas coordenadas que, em teoria, deveriam ser iguais, na prática, sejam consideradas diferentes. Por exemplo, o ponto $(-1, 0)$ em coordenadas rectangulares pode ser expresso através das coordenadas polares $\rho = 1, \phi = \pi$ mas o Auto Lisp não as considera iguais, tal como é perfeitamente visível na seguinte interação:

```
_\$ (=c (xy -1 0) (pol 1 pi))
nil
_\$ (xy -1 0)
(-1 0 0)
_\$ (pol 1 pi)
(-1.0 1.22465e-016 0)
```

Como se pode ver, embora as coordenadas não sejam iguais, elas são bastante próximas, ou seja, a distância entre elas é muito próxima de zero. Proponha uma nova definição para a função `=c` baseada no conceito de distância entre as coordenadas.

3.9 A função `command`

A partir do momento em que sabemos construir coordenadas passamos a poder utilizar um conjunto muito vasto de operações gráficas do Auto Lisp. Existem diferentes formas de se invocar essas operações gráficas mas, por agora, vamos explorar apenas a mais simples: a função `command`.

A função `command` aceita um número arbitrário de expressões que vai avaliando e passando os valores obtidos para o AutoCad à medida que o AutoCad os vai requerendo.¹⁶ No caso de utilização mais comum, a função `command` recebe uma cadeia de caracteres que descreve o comando AutoCad que se pretende executar, seguido de qualquer número de argumentos que serão usados como os dados necessários para a execução desse comando.

¹⁶Este comportamento mostra que, na realidade, `command` não pertence ao conjunto das funções “normais” pois, ao contrário destas, só avalia os argumentos à medida que eles vão sendo necessários.

A vantagem da função `command` é permitir criar programas completos que criam entidades gráficas tal como um normal utilizador de AutoCad o poderia fazer de forma manual. A título de exemplo, consideremos a criação de um círculo. Para se criar um círculo em AutoCad pode-se usar o comando `circle` e fornecer-lhe as coordenadas do centro e o raio do círculo. Se pretendermos criar o mesmo círculo a partir do Auto Lisp, ao invés de invocarmos interactivamente o comando AutoCad `circle` e de lhe fornecermos os dados necessários, podemos simplesmente invocar a função `command` e passar-lhe todos os dados necessários como argumentos, começando pela *string "circle"*, seguida das coordenadas do centro do círculo e, por último, seguida do raio do círculo.

Para concretizarmos o exemplo, consideremos a criação de um círculo de raio $r = 1$ centrado na posição $(x, y) = (2, 3)$. A invocação Auto Lisp que cria este círculo é a seguinte:

```
_\$ (command "circle" (list 2 3) 1)
nil
```

Como se pode ver pelo exemplo, as coordenadas do centro do círculo foram especificadas através da criação de uma lista. No entanto, como referimos na secção 3.7, vamos evitar a especificação de coordenadas directamente em termos da sua representação como listas e vamos, em seu lugar, usar as operações do tipo. Assim, a expressão anterior fica mais correcta na seguinte forma:

```
_\$ (command "circle" (xy 2 3) 1)
nil
```

Como se pode ver também, a invocação da função `command` devolve `nil` como resultado.

Um outro exemplo da utilização desta função é na colocação de um texto na nossa área de desenho. Para isso, podemos usar novamente a função `command` mas, desta vez, os argumentos a passar serão outros, nomeadamente, a cadeia de caracteres "`text`" que designa o comando desejado, as coordenadas onde pretendemos colocar o canto inferior esquerdo do texto, um número que representa a altura do texto, um número que representa o ângulo (em radianos) que a base do texto deverá fazer com a horizontal e, finalmente, uma cadeia de caracteres com o texto a inserir. Eis um exemplo:

```
_\$ (command "text" (xy 1 1) 1 0 "AutoCAD")
nil
```

O resultado da avaliação das duas invocações anteriores é visível da Figura 7.

Finalmente, a função `command` pode ser usada para alterar os parâmetros de funcionamento do AutoCad. Por exemplo, para desligar os *Object Snaps* podemos fazer:



Figura 7: Um círculo com o texto "AutoCad".

```
_\$ (command "osnap" "off")  
nil
```

É importante memorizar esta última expressão pois, sem a sua invocação, todos os usos da função `command` estarão sob o efeito de *Object Snaps*, fazendo com que as coordenadas que indicamos para as nossas figuras geométricas não sejam estritamente respeitadas pelo AutoCad, que as “arredondará” de acordo com a malha do *Object Snaps*.

Como exemplo da utilização de comandos em Auto Lisp, a Figura 8 mostra o resultado da avaliação das seguintes expressões onde usamos coordenadas polares para desenhar círculos e texto:

```
(command "erase" "all" "")  
(command "circle" (pol 0 0) 4)  
(command "text" (+pol (pol 0 0) 5 0)  
           1 0 "Raio: 4")  
(command "circle" (pol 4 (/ pi 4)) 2)  
(command "text" (+pol (pol 4 (/ pi 4)) 2.5 0)  
           0.5 0 "Raio: 2")  
(command "circle" (pol 6 (/ pi 4)) 1)  
(command "text" (+pol (pol 6 (/ pi 4)) 1.25 0)  
           0.25 0 "Raio: 1")  
(command "zoom" "e")
```

Exercicio 3.12 Refaça o desenho apresentado na Figura 8 mas utilizando apenas coordenadas rectangulares.

Existem algumas particularidades da função `command` que é importante discutir. Para as compreendermos é preciso recordar que a função `command` “simula” a interacção do utilizador com o AutoCad. Consideremos então um exemplo de uma interacção: imaginemos que estamos em frente à interface do AutoCad e pretendemos apagar todo o conteúdo da nossa área de desenho no AutoCad. Para isso, podemos invocar o comando `erase` do AutoCad, o que podemos fazer escrevendo as letras `e-r-a-s-e` e

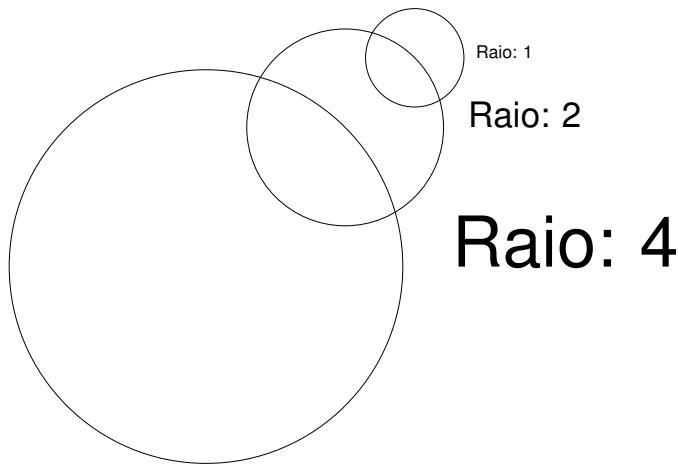


Figura 8: Círculos e textos especificados usando coordenadas polares.

premindo `enter` no final. Acontece que o comando não pode ser imediatamente executado pois o AutoCad precisa de saber mais informação, em particular, o que é que é para apagar. Para isso, o AutoCad pede-nos para seleccionar um ou mais objectos para apagar. Nessa altura, se respondermos com a palavra `all` e premirmos novamente a tecla `enter`, o AutoCad selecciona todos os objectos mas fica a aguardar que indiquemos que terminámos a selecção de objectos, o que se pode fazer premindo simplesmente a tecla `enter`.

Ora para que a função `command` possa simular a interacção que acabámos de descrever, ela tem de passar ao AutoCad todas as informações necessárias e tem de o fazer à medida que o AutoCad precisa delas. Assim, cada sequência de teclas que foi por nós dada ao AutoCad deverá ser agora dada na forma de *string* como argumento à função `command`. No entanto, como não faz muito sentido obrigar o utilizador a especificar, numa *string* o premir da tecla `enter`, a função `command` assume que, após passar todos os caracteres de uma *string* ao AutoCad, deve passar-lhe também o correspondente “premir da tecla `enter`.”

Tentemos agora, à luz desta explicação, construir a invocação da função `command` que simula a interacção anterior. O primeiro passo da avaliação da expressão é, como referimos, o passar da sequência de letras `e-r-a-s-e` seguidas do premir virtual da tecla `enter`, o que podemos fazer escrevendo:

```
(command "erase" ...)
```

Em seguida, sabemos que o AutoCad nos vai pedir para seleccionarmos os objectos a apagar, pelo que escrevemos a *string* com a palavra `all` que a função irá passar ao AutoCad, novamente terminando-a com o premir virtual da tecla `enter`:

```
(command "erase" "all" ...)
```

Finalmente, sabemos que depois desta interacção, o AutoCad vai continuar à espera que indiquemos que terminámos a selecção de objectos através do premir da tecla `enter`, o que implica que temos de indicar à função `command` para simplesmente passar o tal “premir virtual da tecla `enter`.” Ora já sabemos que a função faz isso automaticamente após passar os caracteres de uma *string*. Assim, se não queremos passar caracteres alguns, mas apenas a tecla `enter`, então, logicamente, temos de usar uma *string* vazia, ou seja:

```
(command "erase" "all" "")
```

Assim, a *string* vazia, quando usada como argumento de `command`, é equivalente a premir a tecla `enter` na linha de comandos do AutoCad.

É claro que, se na sequência de uma invocação da função `command`, o AutoCad ficar à espera de dados, novas invocações da função `command` poderão providenciar esses dados ou o utilizador poderá dar essa informação directamente no AutoCad. Isto quer dizer que a invocação anterior pode também ser feita na forma:

```
(command "erase")
(command "all")
(command "")
```

Como se pode constatar pelo exemplo anterior, a função `command` termina assim que conseguir passar todos os argumentos que tem para o AutoCad, independentemente de eles serem suficientes ou não para o AutoCad conseguir completar o pretendido. Se não forem, o AutoCad limita-se a continuar à espera que lhe forneçamos, via `command` ou directamente na interface, os dados que lhe faltam, o que nos permite implementar um processo de “colaboração” com o AutoCad: parte do que se pretende é feito no lado do Auto Lisp e a parte restante no lado do AutoCad.

Por exemplo, imaginemos que pretendemos criar um círculo num dado ponto mas queremos deixar ao utilizador a escolha do raio. Uma solução será *iniciar* a construção do círculo centrado num dado ponto através da função `command`, mas aguardar que o utilizador termine essa construção, indicando explicitamente no AutoCad (por exemplo, usando o rato) qual é o raio do círculo pretendido. Assim, iniciariamos o comando com:

```
(command "circle" (xy 0 0))
```

e o utilizador iria então ao AutoCad terminar a criação do círculo através da indicação do raio.

Um problema um pouco mais complicado será criar um círculo de raio fixo mas deixar ao utilizador a escolha do centro. A complicação deriva de

não conhecemos as coordenadas do centro mas querermos passar o raio: se nos falta o argumento do “meio,” ou seja, a posição do centro do círculo, não podemos passar o do “fim,” ou seja, o raio do círculo.

Há várias soluções para este problema mas, por agora, iremos explorar apenas uma. Para contornar o problema, o AutoCad permite a utilização do carácter especial “\” para indicar que se pretende suspender momentaneamente a passagem de argumentos para o AutoCad, para se retomar essa passagem assim que o AutoCad obtenha o valor correspondente ao argumento que não foi passado. Para facilitar a legibilidade dos programas, o símbolo `pause` designa precisamente a *string* que contém esse único carácter. A seguinte interacção mostra um exemplo da utilização deste símbolo para resolver o problema da criação de um círculo de raio igual a 3 mas centro especificado pelo utilizador. Note-se que, tal como referido na Tabela 1, o carácter “\” é um carácter de *escape* e é por isso que tem de ser escrito em duplicado.

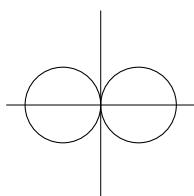
```
_S pause
"\\"
_S (command "circle" pause 3)
nil
```

Um outro aspecto importante da função `command` prende-se com a passagem de números e coordenadas (representadas por listas de números). Embora nos exemplos anteriores estes tipos de dados tenham sido directamente passadas à função `command`, na realidade também podem ser passados “simulando” o premir das teclas correspondentes. Isto quer dizer que é possível criar um círculo com centro no ponto (2, 3) e raio 1 através da expressão:

```
(command "circle" "2,3" "1")
```

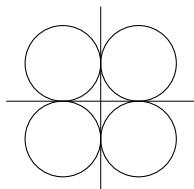
O que acontece, na prática, é que os argumentos da função `command`, para além de só serem avaliados quando são necessários, são também automaticamente convertidos para o formato pretendido pelo AutoCad. Isto permite-nos “trabalhar” as propriedades das entidades geométricas no formato que nos é mais conveniente (números, coordenadas, etc) e deixar à função `command` a responsabilidade de converter os valores para o formato apropriado para o AutoCad.

Exercicio 3.13 Pretendemos colocar duas circunferências de raio unitário em torno da origem de modo a que fiquem encostadas uma à outra, tal como se ilustra no seguinte desenho:



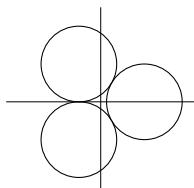
Escreva uma sequência de expressões que, quando avaliadas, produzem a figura anterior.

Exercicio 3.14 Pretendemos colocar quatro circunferências de raio unitário em torno da origem de modo a que fiquem encostadas umas às outras, tal como se ilustra no seguinte desenho:



Escreva uma sequência de expressões que, quando avaliadas, produzem a figura anterior.

Exercicio 3.15 Pretendemos colocar três circunferências de raio unitário em torno da origem de modo a que fiquem encostadas umas às outras, tal como se ilustra no seguinte desenho:



Escreva uma sequência de expressões que, quando avaliadas, produzem a figura anterior.

3.10 Variantes de Comandos

Para a correcta invocação dos comandos do AutoCad a partir do Auto Lisp é necessário ter em conta que o AutoCad permite ao utilizador, quer a redefinição de comandos, quer a sua eliminação. Isto implica que quando invocamos um comando como, por exemplo, `circle`, estamos na realidade a invocar a mais recente redefinição que foi feita para esse comando. Só no caso de não ter sido feita qualquer redefinição do comando e de este não ter sido eliminado é que iremos invocar a definição original. A consequência é que, não sabendo se foi feita uma redefinição ou se o comando não foi tornado como não definido, não podemos ter a certeza do que vai realmente ser executado. Assim, é perfeitamente possível que um programa Auto Lisp que funcionava correctamente num determinado AutoCad, não consiga funcionar noutro porque, nesse, um ou mais dos comandos usados pelo programa foram redefinidos ou eliminados.

Para evitar este problema, o AutoCad disponibiliza uma forma de aceder à definição original de um comando, independentemente de ele ter

sido redefinido ou eliminado: precedendo o nome de um comando com um ponto “.”, o AutoCad executa o comando originalmente definido com esse nome.

Uma outra característica do AutoCad que pode complicar a utilização da função `command` é a *internacionalização*: em diferentes países, o AutoCad utiliza os nomes dos comandos traduzidos para diferentes línguas. Por exemplo, o comando `circle` escreve-se `kreis` no AutoCad Alemão, `cercle` no Francês, `cerchio` no Italiano, `kružnice` no Checo e `circulo` no Espanhol. Assim, se tivermos feito um programa que invoca o comando `circle` e tentarmos executar esse comando num AutoCad preparado para outra língua, iremos obter um erro por o comando ser desconhecido nessa língua.

Para evitar este problema, o AutoCad admite ainda outra variante para todos os comandos: se o nome do comando começar com o carácter “_” então é feita a execução do comando na língua original do AutoCad, que é o Inglês.

A combinação destas duas variantes é possível. O nome `_circle` (ou `._circle`) indica a versão original do comando, independentemente de alterações linguísticas ou redefinições do comando.

Para evitar problemas, de agora em diante iremos sempre utilizar esta convenção de preceder com os caracteres “_.” todos os comandos que usarmos na operação `command`.

Exercicio 3.16 Defina a função `comando-pre-definido` que, dado o nome de um comando em inglês, devolve esse nome convenientemente modificado para permitir aceder ao comando correspondente pré-definido no AutoCad independentemente da língua em que este esteja.

3.11 Ângulos em Comandos

Alguns dos comandos do AutoCad necessitam de saber raios e ângulos em simultâneo. Por exemplo, para criar um polígono regular, o comando `polygon` necessita de saber quantos lados ele deverá ter, qual o centro do polígono, se ele vai estar inscrito numa circunferência ou circunscrito numa circunferência e, finalmente o raio dessa circunferência e o ângulo que o “primeiro” vértice do polígono faz com o eixo dos *x*. Este dois últimos parâmetros, contudo, não podem ser fornecidos separadamente, sendo necessário especificá-los de uma forma conjunta, através de uma *string* em que se junta o raio e o ângulo separados pelo carácter “<” (representando um ângulo) e precedidos do carácter “@” (representando um incremento relativamente ao último ponto dado, que era o centro da circunferência). Por exemplo, o raio 2 com um ângulo de 30° escreve-se “@2<30”. Note-se que o ângulo tem de estar em graus. Se, ao invés de fornecermos esta *string* fornecermos apenas um número, o AutoCad irá tratar esse número como o

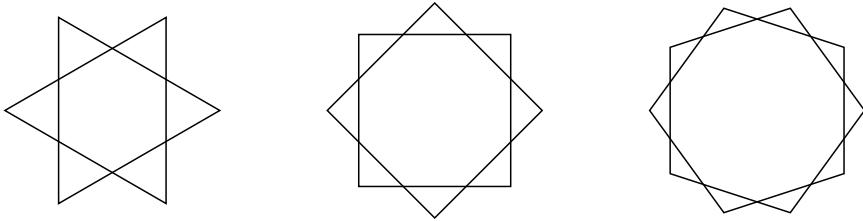


Figura 9: Triângulos, quadrados e pentágonos sobrepostos com diferentes ângulos de rotação.

raio e, mesmo que lhe tentemos fornecer outro número para o ângulo, irá assumir que o ângulo é zero.

Uma vez que, na maior parte dos casos, os raios e ângulos serão parâmetros das nossas funções e ainda que, do ponto de vista matemático, é preferível trabalhar em radianos, é conveniente definir uma função que, a partir do raio e ângulo em radianos, produz a *string* apropriada com o ângulo em graus. Para isso, é conveniente usarmos a função `strcat` para concatenarmos as *strings* parciais que serão produzidas através da função `rtos` que converte um número numa *string*. Assim, temos:

```
(defun raio&angulo (raio angulo)
  (strcat "@" (rtos raio) "<" (rtos (graus<-radianos angulo))))
```

Usando esta função é agora trivial criarmos polígonos com diferentes ângulos de rotação. Por exemplo, a seguinte sequência de comandos produz a imagem representada na Figura 9:

```
(command "_.polygon" 3
         (xy 0 0) "_Inscribed" (raio&angulo 1 0))
(command "_.polygon" 3
         (xy 0 0) "_Inscribed" (raio&angulo 1 (/ pi 3)))

(command "_.polygon" 4
         (xy 3 0) "_Inscribed" (raio&angulo 1 0))
(command "_.polygon" 4
         (xy 3 0) "_Inscribed" (raio&angulo 1 (/ pi 4)))

(command "_.polygon" 5
         (xy 6 0) "_Inscribed" (raio&angulo 1 0))
(command "_.polygon" 5
         (xy 6 0) "_Inscribed" (raio&angulo 1 (/ pi 5)))
```

3.12 Efeitos Secundários

Vimos anteriormente que qualquer expressão Lisp tem um valor. No entanto, aquilo que se pretende de uma expressão que use a função `command` não é saber qual o seu valor mas sim qual o *efeito* que é produzido num determinado desenho. De facto, a execução de um comando AutoCad produz, em geral, uma alteração do desenho actual, sendo irrelevante o seu valor.

Este comportamento da função `command` é fundamentalmente diferente do comportamento das funções que vimos até agora pois, anteriormente, as funções eram usadas para computar algo, i.e., para produzir um valor a partir da sua invocação com determinados argumentos e, agora, não é o valor que resulta da invocação do comando que interessa mas sim o *efeito secundário* (também chamado *efeito colateral*) que interessa.

Contudo, mesmo no caso em que apenas nos interessa o efeito secundário, é necessário continuar a respeitar a regra de que, em Lisp, qualquer expressão tem um valor e, por isso, também uma invocação de função tem de produzir um valor como resultado. É por este motivo que a invocação da função `command` devolve sempre `nil` como resultado. Obviamente, qualquer outro valor serviria (pois não é suposto ser usado) mas convenciona-se que, nos casos em que não há um valor mais relevante a devolver, deve-se devolver `nil` (que, em Latim, significa *nada*).

Um dos aspectos importantes da utilização de efeitos secundários está na possibilidade da sua *composição*. A composição de efeitos secundários faz-se através da sua *sequenciação*, i.e., da realização sequencial dos vários efeitos. Na secção seguinte iremos discutir a sequenciação de efeitos secundários.

3.13 Sequenciação

Até agora, temos combinado expressões matemáticas usando operadores matemáticos. Por exemplo, a partir das expressões $(\sin x)$ e $(\cos x)$, podemos calcular a sua divisão através de $(/ (\sin x) (\cos x))$. Isto é possível porque a avaliação das subexpressões $(\sin x)$ e $(\cos x)$ irá produzir dois valores que podem então ser usados para fazer a divisão.

Já no caso de expressões envolvendo a função `command`, a sua combinação tem de ser realizada de forma diferente pois, como vimos, a sua avaliação não produz valores relevantes mas apenas efeitos secundários. Como é precisamente a realização desses efeitos que nos interessa, o Auto Lisp providencia uma forma de os realizarmos sequencialmente, i.e., uns a seguir aos outros. Essa forma denomina-se `progn` e serve para fazermos a *sequenciação* de efeitos secundários. A título de exemplo, consideremos uma função que desenha um círculo de raio unitário com um quadrado inscrito ou circunscrito, dependendo de uma escolha do utilizador, tal como



Figura 10: Um círculo com um quadrado inscrito (à esquerda) e circunscrito (à direita).

apresentamos na Figura 10.

A definição desta função poderá começar por algo do género:

```
(defun circulo-quadrado (inscrito?)
  ...)
```

O desenho produzido pela função depende, naturalmente, do valor lógico do parâmetro `inscrito?`, ou seja, a definição da função deverá ser algo da forma:

```
(defun circulo-quadrado (inscrito?)
  (if inscrito?
      desenha um círculo e um rectângulo inscrito no círculo
      desenha um círculo e um rectângulo circunscrito ao círculo))
```

O problema que se coloca agora é que, em cada caso do `if`, a função necessita de realizar *dois* efeitos secundários, nomeadamente, desenhar um círculo e desenhar um rectângulo. Ora um `if` apenas admite, em cada caso, uma única expressão, o que nos leva a perguntar como é que podemos combinar os dois efeitos secundários numa só expressão. Para este fim, o Auto Lisp disponibiliza a forma `progn`. Este operador recebe várias expressões que avalia sequencialmente, i.e., uma a seguir à outra, devolvendo o valor da última. Logicamente, se apenas o valor da última expressão é utilizado, então os valores de todas as outras avaliações dessa sequência são descartados e estas apenas são relevantes pelos efeitos secundários que possam provocar.

Usando o operador `progn` já podemos detalhar um pouco mais a função:

```
(defun circulo-quadrado (inscrito?)
  (if inscrito?
      (progn
        desenha um círculo
        desenha um rectângulo inscrito no círculo)
      (progn
        desenha um círculo
        desenha um rectângulo circunscrito ao círculo))))
```

Basta-nos agora traduzir cada um dos desenhos elementares em invocações da função `command`. No caso do rectângulo inscrito, vamos utilizar coordenadas polares pois sabemos que os vértices terão de ficar sobre o círculo, um a $\pi/4$ e ou outro a $\pi + \pi/4 = 5\pi/4$. Assim, temos:

```
(defun circulo-quadrado (inscrito?)
  (if inscrito?
      (progn
        ((command ".circle" (xy 0 0) 1))
        ((command ".rectangle" (pol 1 (/ (* 5 pi) 4)) (pol 1 (/ pi 4))))
      (progn
        ((command ".circle" (xy 0 0) 1))
        ((command ".rectangle" (xy -1 -1) (xy 1 1))))))
```

Reparemos que, na óptica do operador `if`, quer o consequente, quer a alternativa, são uma única expressão, embora cada uma dessas expressões corresponda à sequenciação de duas expressões mais elementares. O operador `progn` pode assim ser visto como um mecanismo de agrupamento de expressões.

Apesar de a sequenciação de expressões implicar a utilização do operador `progn`, é normalmente possível simplificar essas expressões de forma a minimizar o seu uso. Por exemplo, uma observação atenta da função `circulo-quadrado` mostra que a criação do círculo é sempre feita, independentemente do rectângulo ser inscrito ou circunscrito. Isto permite-nos reescrever a definição da função de modo a que a criação do círculo seja realizada fora do `if`:

```
(defun circulo-quadrado (inscrito?)
  (progn
    ((command ".circle" (xy 0 0) 1))
    (if inscrito?
        (progn
          ((command ".rectangle" (pol 1 (/ (* 5 pi) 4)) (pol 1 (/ pi 4))))
        (progn
          ((command ".rectangle" (xy -1 -1) (xy 1 1)))))))
```

É claro que, agora, os dois `progs` dentro do `if` apenas têm uma expressão, pelo que o seu uso é perfeitamente dispensável. Assim, podemos simplificar a definição para:

```
(defun circulo-quadrado (inscrito?)
  (progn
    (command "_circle" (xy 0 0) 1)
    (if inscrito?
        (command "_rectangle" (pol 1 (/ (* 5 pi) 4)) (pol 1 (/ pi 4)))
        (command "_rectangle" (xy -1 -1) (xy 1 1))))
```

Embora o operador `progn` seja necessário sempre que pretendemos avaliar mais do que uma expressão, existem alguns locais onde o Auto Lisp assume um `progn` *implícito* como, por exemplo, no corpo de uma função. De facto, quando invocamos uma função, todas as expressões que colocamos no seu corpo são avaliadas sequencialmente, sendo o valor da função determinado apenas pela última das expressões. Esta característica permite-nos simplificar ainda mais a função `circulo-quadrado`, ficando apenas:

```
(defun circulo-quadrado (inscrito?)
  (command "_circle" (xy 0 0) 1)
  (if inscrito?
      (command "_rectangle" (pol 1 (/ (* 5 pi) 4)) (pol 1 (/ pi 4)))
      (command "_rectangle" (xy -1 -1) (xy 1 1))))
```

Exercicio 3.17 Defina uma função denominada `circulo-e-raio` que, dadas as coordenadas do centro do círculo e o raio desse círculo, cria o círculo especificado no AutoCad e, à semelhança do que se vê na Figura 8, coloca o texto a descrever o raio do círculo à direita do círculo. O texto deverá ter um tamanho proporcional ao raio do círculo.

Exercicio 3.18 Utilize a função `circulo-e-raio` definida na pergunta anterior para reconstituir a imagem apresentada na Figura 8.

3.14 A Ordem Dórica

Na Figura 11 apresentamos uma imagem do templo grego de Segesta. Este templo, que nunca chegou a ser acabado, foi construído no século quinto antes de Cristo e representa um excelente exemplo da *Ordem Dórica*, a mais antiga das três ordens da arquitectura Grega clássica. Nesta ordem, uma coluna caracteriza-se por ter um fuste, um coxim e um ábaco. O ábaco tem a forma de uma placa quadrada que assenta sobre o coxim, o coxim assemelha-se a um tronco de cone invertido e assenta sobre o fuste, e o fuste assemelha-se a um tronco de cone com vinte *caneluras* em seu redor. Estas caneluras assemelham-se a uns canais semi-circulares escavados ao longo da coluna.¹⁷ Quando os Romanos copiaram a Ordem Dórica introduziram-lhe um conjunto de alterações, em particular, nas caneluras que, muitas vezes, são simplesmente eliminadas.

¹⁷Estas colunas apresentam ainda uma deformação intencional denominada *entasis*. A entasis consiste em dar uma ligeira curvatura à coluna e, segundo alguns autores, destina-se a corrigir uma ilusão de óptica que faz as colunas direitas parecerem encurvadas.



Figura 11: O Templo Grego de Segesta, exemplificando alguns aspectos da Ordem Dórica. Este templo nunca chegou a ser terminado, sendo visível, por exemplo, a falta das caneluras nas colunas. Fotografia de Enzo De Martino.

Embora o ênfase desta obra seja na modelação tridimensional, por motivos pedagógicos vamos começar por esquematizar o alçado de uma coluna Dórica (sem caneluras). Nas secções seguintes iremos estender este processo para a criação de um modelo tridimensional.

Do mesmo modo que uma coluna Dórica se pode decompor nos seus componentes fundamentais—o fuste, o coxim e o ábaco—também o desenho da coluna se poderá decompor no desenho dos seus componentes. Assim, vamos definir funções para desenhar o fuste, o coxim e o ábaco. A Figura 12 apresenta um modelo de referência. Comecemos por definir uma função para o desenho do fuste:

```
(defun fuste ()
  (command "_.line"
    (xy -0.8 10)
    (xy -1 0)
    (xy 1 0)
    (xy 0.8 10)
    (xy -0.8 10)
    ""))

```

Neste exemplo, a função `command` executa o comando AutoCad `line` que, dada uma sequência de pontos, constrói uma linha poligonal com vértices nesses pontos. Para se indicar o fim da sequência de pontos usa-se uma *string* vazia. Naturalmente, a invocação da função `fuste` terá, como

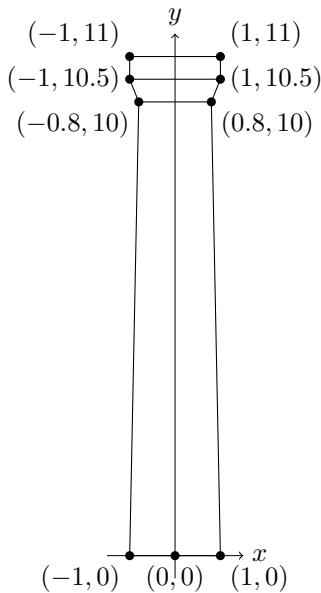


Figura 12: Uma coluna Dórica de referência.

efecto secundário, a criação do fuste da coluna na área de desenho do AutoCad. Uma outra possibilidade, eventualmente mais correcta, seria pedir ao AutoCad a criação de uma linha *fechada*, algo que podemos fazer com a opção "close" no lugar do último ponto, i.e.:

```
(defun fuste ()
  (command "_line"
    (xy -0.8 10)
    (xy -1 0)
    (xy 1 0)
    (xy 0.8 10)
    "close"))
```

Note-se que o resultado do comando `line` é a criação de um conjunto de segmentos de recta. Cada um destes segmentos de recta é uma entidade individual que pode ser seleccionada e modificada independentemente dos restantes. Para o caso de não pretendermos esse tratamento independente, o AutoCad disponibiliza *polilinhas* (também conhecidas por *plines*). Estas são criadas pelo comando `pline` que, neste contexto, difere do comando `line` no facto de ter como resultado a criação de uma única entidade composta pelos vários segmentos.¹⁸

Para completar a figura, é ainda necessário definir uma função para o coxim e outra para o ábaco. No caso do coxim, o raciocínio é semelhante:

¹⁸Outras diferenças incluem o facto de as *plines* poderem ter espessura e estarem limitadas a um plano.



Figura 13: Uma coluna dórica desenhada pelo AutoCad.

```
(defun coxim ()
  (command "_.line"
    (xy -0.8 10)
    (xy -1 10.5)
    (xy 1 10.5)
    (xy 0.8 10)
    "_close"))
```

No caso do ábaco, podemos empregar uma abordagem idêntica ou podemos explorar outro comando do AutoCad ainda mais simples destinado à construção de rectângulos. Este comando apenas necessita de dois pontos para definir completamente o rectângulo:

```
(defun abaco ()
  (command "_.rectangle"
    (xy -1 10.5)
    (xy 1 11)))
```

Finalmente, vamos definir uma função que desenha as três partes da coluna:

```
(defun coluna ()
  (fuste)
  (coxim)
  (abaco))
```

Repare-se, na função `coluna`, que ela invoca sequencialmente as funções `fuste`, `coxim` e, finalmente, `abaco`.

A Figura 13 mostra o resultado da invocação da função `coluna`.

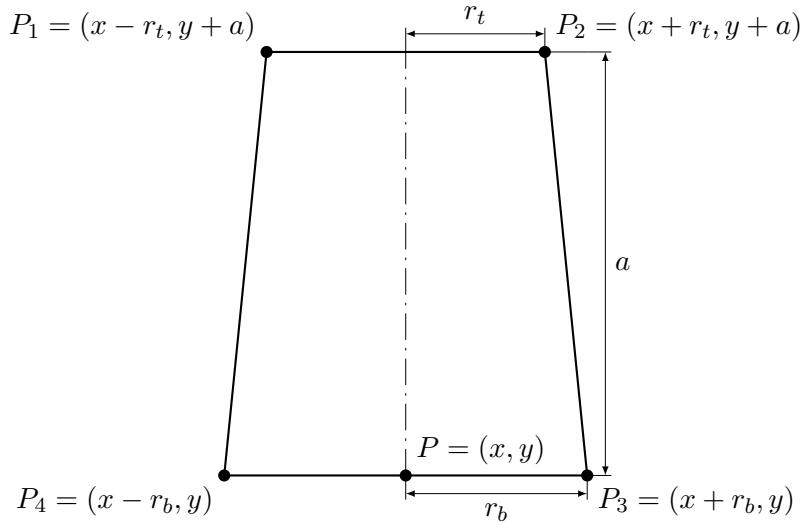


Figura 14: Esquema do desenho do fuste de uma coluna.

3.15 Parametrização de Figuras Geométricas

Infelizmente, a coluna que criámos na secção anterior tem todas as suas dimensões fixas, pelo que será difícil encontrarmos outras situações em que possamos reutilizar a função que definimos. Naturalmente, esta função seria mais útil se a criação da coluna fosse *parametrizável*, i.e., se a criação dependesse dos vários parâmetros que caracterizam a coluna como, por exemplo, as coordenadas da base da coluna, a altura do fuste, do coxim e do ábaco, os raios da base e do topo do fuste, etc.

Para se compreender a parametrização destas funções vamos começar por considerar o fuste representado esquematicamente na Figura 14.

O primeiro passo para parametrizarmos um desenho geométrico consiste na identificação dos parâmetros relevantes. No caso do fuste, um dos parâmetros óbvios é a localização espacial desse fuste, i.e., as coordenadas de um ponto de referência em relação ao qual fazemos o desenho do fuste. Assim, começemos por imaginar que o fuste irá ser colocado com o centro da base num imaginário ponto P de coordenadas arbitrárias (x, y) . Para além deste parâmetro, temos ainda de conhecer a altura do fuste a e os raios da base r_b e do topo r_t do fuste.

Para mais facilmente idealizarmos um processo de desenho, é conveniente assinalarmos no esquema alguns pontos de referência adicionais. No caso do fuste, uma vez que o seu desenho é, essencialmente, um trapézio, basta-nos idealizar o desenho deste trapézio através de uma sucessão de linhas rectas dispostas ao longo de uma sequência de pontos P_1 , P_2 , P_3 e P_4 , pontos esses que conseguimos calcular facilmente a partir do ponto P .

Desta forma, estamos em condições de definir a função que desenha

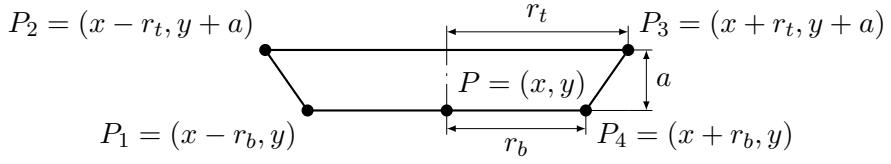


Figura 15: Esquema do desenho do coxim de uma coluna.

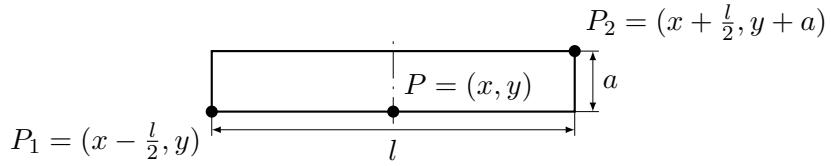


Figura 16: Esquema do desenho do ábaco de uma coluna.

o fuste. Para tornar mais claro o programa, vamos empregar os nomes *a-fuste*, *r-base* e *r-topo* para caracterizar a altura *a*, o raio da base *r_b* e o raio do topo *r_t*, respectivamente. A definição fica então:

```
(defun fuste (p a-fuste r-base r-topo)
  (command "_.line"
    (+xy p (- r-topo) a-fuste)
    (+xy p (- r-base) 0)
    (+xy p (+ r-base) 0)
    (+xy p (+ r-topo) a-fuste)
    "_close"))
```

Em seguida, temos de especificar o desenho do coxim. Mais uma vez, convém pensarmos num esquema geométrico, tal como apresentamos na Figura 15.

Tal como no caso do fuste, a partir de um ponto *P* correspondente ao centro da base do coxim, podemos computar as coordenadas dos pontos que estão nas extremidades dos segmentos de recta que delimitam o desenho do coxim. Usando estes pontos, a definição da função fica com a seguinte forma:

```
(defun coxim (p a-coxim r-base r-topo)
  (command "_.line"
    (+xy p (- r-base) 0)
    (+xy p (- r-topo) a-coxim)
    (+xy p (+ r-topo) a-coxim)
    (+xy p (+ r-base) 0)
    "_close"))
```

Terminado o fuste e o coxim, é preciso definir o desenho do ábaco. Para isso, fazemos um novo esquema que apresentamos na Figura 16.

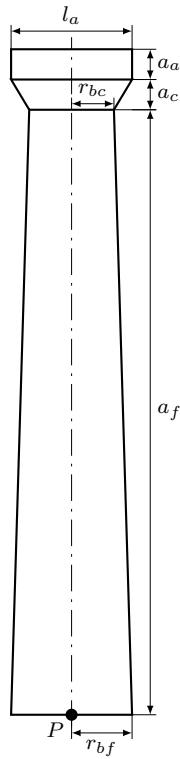


Figura 17: A composição do fuste, coxim e ábaco.

Mais uma vez, vamos considerar como ponto de partida o ponto P no centro da base do ábaco. A partir deste ponto, podemos facilmente calcular os pontos P_1 e P_2 que constituem os dois extremos do rectângulo que representa o alçado do ábaco. Assim, temos:

```
(defun abaco (p a-abaco l-abaco)
  (command "_rectangle"
    (+xy p (/ l-abaco -2.0) 0)
    (+xy p (/ l-abaco +2.0) a-abaco)))
```

Finalmente, para desenhar a coluna completa, temos de combinar os desenhos do fuste, do coxim e do ábaco. Apenas precisamos de ter em conta que, tal como a Figura 17 demonstra, o raio do topo do fuste coincide com o raio da base do coxim e o raio do topo do coxim é metade da largura do ábaco. A mesma Figura mostra também que as coordenadas da base do coxim correspondem a somar a altura do fuste às coordenadas da base do fuste e as coordenadas da base do ábaco correspondem a somar a altura do fuste e a altura do coxim às coordenadas da base do fuste.

Tal como fizemos anteriormente, vamos dar nomes mais claros aos parâmetros da Figura 17. Usando os nomes p , a -fuste, r -base-fuste,

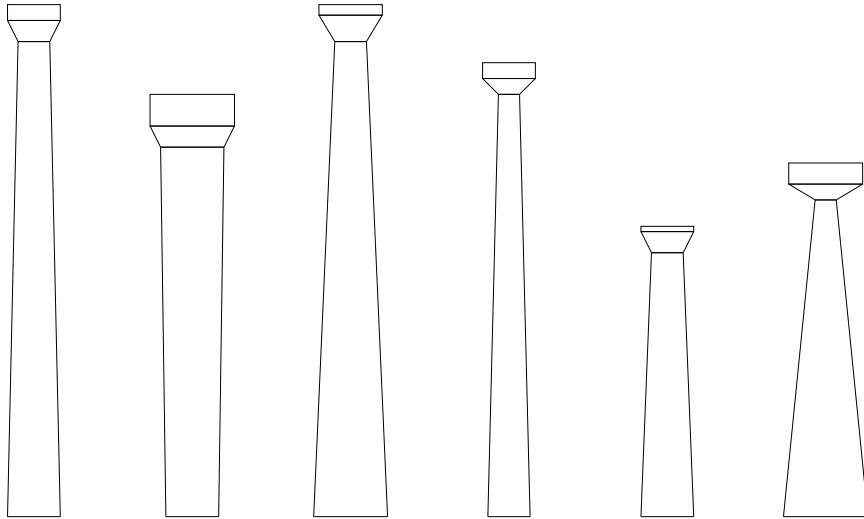


Figura 18: Variações de colunas dóricas.

a-coxim, r-base-coxim, a-abaco e l-abaco no lugar de, respectivamente, $P, a_f, r_{bf}, a_c, r_{bc}, a_a$ e l_a , temos:

```
(defun coluna (p
    a-fuste r-base-fuste
    a-coxim r-base-coxim
    a-abaco l-abaco)
  (fuste p a-fuste r-base-fuste r-base-coxim)
  (coxim (+xy p 0 a-fuste) a-coxim r-base-coxim (/ l-abaco 2.0))
  (abaco (+xy p 0 (+ a-fuste a-coxim)) a-abaco l-abaco))
```

Com base nestas funções, podemos agora facilmente experimentar variações de colunas. As seguintes invocações produzem o desenho apresentado na Figura 18.

```
(coluna (xy 0 0) 9 0.5 0.4 0.3 0.3 1.0)
(coluna (xy 3 0) 7 0.5 0.4 0.6 0.6 1.6)
(coluna (xy 6 0) 9 0.7 0.5 0.3 0.2 1.2)
(coluna (xy 9 0) 8 0.4 0.3 0.2 0.3 1.0)
(coluna (xy 12 0) 5 0.5 0.4 0.3 0.1 1.0)
(coluna (xy 15 0) 6 0.8 0.3 0.2 0.4 1.4)
```

Como é óbvio pela análise desta figura, nem todas as colunas desenhadas obedecem aos cânones da ordem Dórica. Mais à frente iremos ver que modificações serão necessárias para evitar este problema.

3.16 Documentação

Na função `coluna`, *a-fuste* é a altura do fuste, *r-base-fuste* é o raio da base do fuste, *r-topo-fuste* é o raio do topo do fuste, *a-coxim* é a

altura do coxim, a-abaco é a altura do ábaco e, finalmente, r-abaco é o raio do ábaco. Uma vez que a função já tem vários parâmetros e o seu significado poderá não ser óbvio para quem lê a definição da função pela primeira vez, é conveniente *documentar* a função. Para isso, a linguagem Lisp providencia uma sintaxe especial: sempre que surge o carácter ponto-e-vírgula ; , o processo de leitura do Lisp ignora tudo o que vem a seguir até ao fim da linha. Isto permite-nos escrever texto nos nossos programas sem correr o risco de o Lisp tentar perceber o que lá está escrito.

Usando documentação, o nosso programa completo para desenhar colunas dóricas fica com o aspecto ilustrado na Figura 19. Note-se que o exemplo destina-se a mostrar as diferentes formas de documentação usadas em Lisp e não a mostrar um exemplo típico de programa documentado.¹⁹

Quando um programa está documentado, a sua leitura permite ficar com uma ideia muito mais clara do que cada função faz sem obrigar ao estudo do corpo das funções. Como se vê pelo exemplo, é pragmática usual em Lisp usar um diferente número de caracteres ";" para indicar a relevância do comentário:

- ;;;; Devem começar na margem esquerda e servem para dividir o programa em secções e dar um título a cada secção.
- ;; Devem começar na margem esquerda e servem para fazer comentários gerais ao programa que aparece em seguida. Não se devem usar no interior das funções.
- ; Devem estar alinhadas com a parte do programa a que se vão aplicar, que aparece imediatamente em baixo.
- ; Devem aparecer alinhados numa mesma coluna à direita e comentam a parte do programa imediatamente à esquerda.

É importante que nos habituemos a documentar as nossas definições, mas convém salientar que a documentação em excesso também tem desvantagens:

- O código Lisp deve ser suficientemente claro para que um ser humano o consiga perceber. É sempre preferível perder mais tempo a tornar o código claro do que a escrever documentação que o explique.
- Documentação que não está de acordo com o programa é pior que não ter documentação.

¹⁹Na verdade, o programa é tão simples que não deveria necessitar de tanta documentação.

```

;;;;;Desenho de colunas doricas

;;;;O desenho de uma coluna dorica divide-se no desenho do
;;;;fuste, do coxim e do abaco. A cada uma destas partes
;;;;corresponde uma funcao independente.

;Desenha o fuste de uma coluna dorica.
;p: coordenadas do centro da base da coluna,
;a-fuste: altura do fuste,
;r-base: raio da base do fuste,
;r-topo: raio do topo do fuste.
(defun fuste (p a-fuste r-base r-topo)
  (command "_line"
    (+xy p (- r-topo) a-fuste) ;a criacao de linhas
    (+xy p (- r-base) 0) ;com a funcao command
    (+xy p (+ r-base) 0) ;tem de ser terminada
    (+xy p (+ r-topo) a-fuste) ;com a opcao "close"
    (+xy p (+ r-base) 0) ;para fechar a figura
    "close"))

;Desenha o coxim de uma coluna dorica.
;p: coordenadas do centro da base do coxim,
;a-coxim: altura do coxim,
;r-base: raio da base do coxim,
;r-topo: raio do topo do coxim.
(defun coxim (p a-coxim r-base r-topo)
  (command "_line"
    (+xy p (- r-base) 0)
    (+xy p (- r-topo) a-coxim)
    (+xy p (+ r-topo) a-coxim)
    (+xy p (+ r-base) 0)
    "close")) ;para fechar a figura

;Desenha o abaco de uma coluna dorica.
;p: coordenadas do centro da base da coluna,
;a-abaco: altura do abaco,
;l-abaco: largura do abaco.
(defun abaco (p a-abaco l-abaco)
  (command "_rectangle"
    (+xy p (/ l-abaco -2.0) 0)
    (+xy p (/ l-abaco +2.0) a-abaco)))

;Desenha uma coluna dorica composta por fuste, coxim e abaco.
;p: coordenadas do centro da base da coluna,
;a-fuste: altura do fuste,
;r-base-fuste: raio da base do fuste,
;r-base-coxim: raio da base do coxim = raio do topo do fuste,
;a-coxim: altura do coxim,
;a-abaco: altura do abaco,
;l-abaco: largura do abaco = 2*raio do topo do coxim.
(defun coluna (p
  a-fuste r-base-fuste
  a-coxim r-base-coxim
  a-abaco l-abaco)
  ;;desenhamos o fuste com a base em p
  (fuste p a-fuste r-base-fuste r-base-coxim)
  ;;colocamos o coxim por cima do fuste
  (coxim (+ p a-fuste) a-coxim r-base-coxim (/ l-abaco 2.0))
  ;;e o abaco por cima do coxim
  (abaco (+ p (+ a-fuste a-coxim)) a-abaco l-abaco))

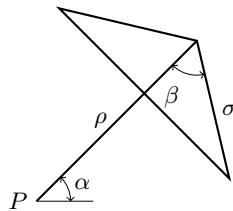
```

Figura 19: Um programa Auto Lisp documentado.

- É frequente termos de modificar os nossos programas para os adaptar a novos fins. Quanto mais documentação existir, mais documentação é necessário alterar para a pôr de acordo com as alterações que tivermos feito ao programa.

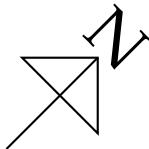
Por estes motivos, devemos esforçar-nos por escrever o código mais claro que nos for possível e, ao mesmo tempo, providenciar documentação sucinta e útil: a documentação não deve dizer aquilo que é óbvio a partir da leitura do programa.

Exercicio 3.19 Considere o desenho de uma seta com origem no ponto P , comprimento ρ , inclinação α , ângulo de abertura β e comprimento da “farpa” σ , tal como se representa em seguida:



Defina uma função denominada `seta` que, a partir dos parâmetros P , ρ , α , σ e β , constrói a seta correspondente.

Exercicio 3.20 Com base na solução do exercício anterior, defina uma função que, dados o ponto P , a distância ρ e o ângulo α , desenha “o norte” tal como se apresenta no esquema em baixo:



O desenho deve ainda obedecer às seguintes proporções:

- O ângulo β de abertura da seta é de 45° .
- O comprimento σ da “farpa” é de $\frac{\rho}{2}$.
- O centro da letra “N” deverá ser posicionado a uma distância de $\frac{\rho}{10}$ da extremidade da seta segundo a direcção da seta.
- O tamanho da letra “N” deverá ser metade da distância ρ .

Exercicio 3.21 Usando a função `seta`, defina uma nova função denominada `seta-de-para` que, dados dois pontos, cria uma seta que vai do primeiro para o segundo ponto. As farpas da seta deverão ter comprimento unitário e ângulo $\frac{\pi}{8}$.

Exercicio 3.22 Considere o desenho de uma habitação composta apenas por divisões rectangulares. Pretende-se que defina a função `divisao-rectangular` que recebe como parâmetros a posição do canto inferior esquerdo da divisão, o comprimento e a largura da

divisão e um texto a descrever a função dessa divisão na habitação. Com esses valores a função deverá construir o rectângulo correspondente e deve colocar no interior desse rectângulo duas linhas de texto, a primeira com a função da divisão e a segunda com a área da divisão. Por exemplo, a sequência de invocações

```
(divisao-rectangular (xy 0 0) 4 3 "cozinha")
(divisao-rectangular (xy 4 0) 2 3 "despensa")
(divisao-rectangular (xy 6 0) 5 3 "quarto")
(divisao-rectangular (xy 0 5) 5 4 "sala")
(divisao-rectangular (xy 5 5) 3 4 "i.s.")
(divisao-rectangular (xy 8 3) 3 6 "quarto")
```

produz, como resultado, a habitação que se apresenta de seguida:



3.17 Depuração

Como sabemos, *errare humanum est*. O erro faz parte do nosso dia-a-dia e, por isso, em geral, sabemos lidar com ele. Já o mesmo não se passa com as linguagens de programação. Qualquer erro num programa tem como consequência que o programa tem um comportamento diferente daquele que era esperado.

Uma vez que é fácil cometer erros, deve também ser fácil detectá-los e corrigi-los. À actividade de detecção e correcção de erros denomina-se *depuração*. Diferentes linguagens de programação providenciam diferentes mecanismos para essa actividade. Neste domínio, como iremos ver, o AutoCad está particularmente bem apetrechado.

Em termos gerais, os erros num programa podem classificar-se em *erros sintáticos* e *erros semânticos*.

3.17.1 Erros Sintáticos

Os erros sintáticos ocorrem quando escrevemos frases que não obedecem à gramática da linguagem. Como exemplo prático, imaginemos que pretendíamos definir uma função que criava uma única coluna dórica, que iremos

designar de coluna *standard* e que tem sempre as mesmas dimensões, não necessitando de quaisquer outros parâmetros. Uma possibilidade para a definição desta função será:

```
(defun coluna-standard  
  (coluna (xy 0 0) 9 0.5 0.4 0.3 0.3 0.5))
```

No entanto, se avaliarmos aquela definição, o Auto Lisp irá apresentar um erro, avisando-nos de que algo está errado:²⁰

```
; error: bad DEFUN syntax:  
(COLUNA-STANDARD (COLUNA (XY 0 0) 9 0.5 0.4 0.3 0.3 0.5))
```

O erro de que o Auto Lisp nos está a avisar é de que a forma `defun` que lhe demos para avaliar não obedece à sintaxe exigida para essa forma e, de facto, uma observação atenta da forma anterior mostra que não seguimos a sintaxe exigida para uma definição e que, tal como discutimos na secção 2.11, era a seguinte:

```
(defun nome (parâmetro1 ... parâmetron)  
  corpo)
```

O nosso erro é agora óbvio: esquecemo-nos da lista de parâmetros. Se a função não tem parâmetros, a lista de parâmetros é vazia mas tem de estar lá na mesma. Uma vez que não estava, o Auto Lisp detecta e reporta um erro sintático, uma “frase” que não obedece à sintaxe da linguagem.

Há vários outros tipos de erros sintáticos que o Auto Lisp é capaz de detectar e que serão apresentados à medida que os formos discutindo. O importante, no entanto, não é saber quais são os erros sintáticos detectáveis pelo Auto Lisp, mas antes saber que o Auto Lisp é capaz de verificar as expressões que escrevemos e fazer a detecção de erros sintáticos antes mesmo de as avaliar.

Para isso, o Visual Lisp disponibiliza na sua interface operações que fazem essa verificação para uma seleção ou para todo o ficheiro actual. Na versão Inglesa do AutoCad, essas operações denominam-se “Check Selection” ou “Check Text in Editor” e estão disponíveis no menu “Tools.” Como consequência da invocação destas operações, o AutoCad analiza a seleção ou o ficheiro actual e escreve, numa janela à parte, todos os erros sintáticos encontrados. Nessa janela, se clicarmos duas vezes sobre uma mensagem de erro, somos conduzidos ao local do nosso programa onde o erro ocorre.

²⁰Nem todos os dialectos e interpretadores de Lisp possuem exactamente este comportamento mas, variações à parte, os conceitos são os mesmos.

3.17.2 Erros Semânticos

Os erros semânticos são muito diferentes dos sintáticos. Um erro semântico não é um erro na escrita de uma “frase” da linguagem, mas sim um erro no significado dessa frase. Dito de outra forma, um erro semântico ocorre quando escrevemos uma frase que julgamos ter um significado e, na verdade, ela tem outro.

Em geral, os erros semânticos apenas são detectáveis durante a invocação das funções que os contêm. Parte dos erros semânticos é detectável pelo avaliador de Lisp, mas há inúmeros erros cuja detecção só pode ser feita pelo próprio programador.

Como exemplo de erro semântico consideremos uma operação sem significado, como seja a soma de um número com uma *string*:

```
_\$ (+ 1 "dois")
; error: bad argument type: numberp: "dois"
```

Como se pode ver, o erro é explicado na mensagem que indica que o segundo argumento devia ser um número. Neste exemplo, o erro é suficientemente óbvio para o conseguirmos detectar imediatamente. No entanto, no caso de programas mais complexos, isso já poderá não ser assim.

Na continuação do exemplo que apresentámos na discussão sobre erros sintáticos, consideremos a seguinte modificação à função `coluna-standard` que corrige a falta de parâmetros mas que introduz um outro erro:²¹

```
(defun coluna-standard ()
  (coluna (xy 0 0) 9 0.5 0.4 0.3 0.3 0.5))
```

Do ponto de vista sintático, a função está correcta. No entanto, quando a invocamos surge um erro:

```
_\$ (coluna-standard)
; error: bad argument type: numberp: nil
```

Nesta resposta o Auto Lisp protesta que um dos argumentos devia ser um número mas, em lugar disso, era `nil`. Uma vez que a função `coluna-standard` não tem quaisquer argumentos, a mensagem de erro poderá ser difícil de compreender. No entanto, ela torna-se comprehensível quando pensamos que o erro poderá não ter ocorrido na invocação da função `coluna-standard` mas sim em qualquer função que tenha sido invocada directa ou indirectamente por esta.

Para se perceber melhor onde está o erro, o Visual Lisp providencia algumas operações extremamente úteis. Uma delas dá pelo nome de “Error Trace,” está disponível no menu “View” e destina-se a mostrar a “cascata”

²¹Consegue detectá-lo?

de invocações que acabou por provocar o erro.²² Quando executamos essa operação, o Visual Lisp apresenta-nos, numa pequena janela, a seguinte informação:

```
<1> :ERROR-BREAK
[2] (+ nil -0.3)
[3] (+XY (nil 0 0) -0.3 9)
[4] (FUSTE (nil 0 0) 9 0.5 0.3)
[5] (COLUNA (nil 0 0) 9 0.5 0.4 0.3 0.3 0.5)
[6] (COLUNA-STANDARD)
...
```

Na informação acima, as reticências representam outras invocações que não são relevantes para o nosso problema e que correspondem às funções Auto Lisp cuja execução antecede a das nossas funções.²³ A listagem apresentada pelo Visual Lisp mostra, por ordem inversa, as invocações de funções que provocaram o erro. Para cada linha, o Visual Lisp disponibiliza um menu contextual que, entre outras operações, permite visualizarmos imediatamente no editor qual é a linha do programa em questão.

A leitura da listagem do *error trace* diz-nos que o erro foi provocado pela tentativa de somarmos `nil` a um número. Essa tentativa foi feita pela função `+xy`, que foi invocada pela função `fuste`, que foi invocada pela função `coluna`, que foi invocada pela função `coluna-standard`. Como podemos ver, à frente do nome de cada função, aparecem os argumentos com que a função foi invocada. Estes argumentos mostram que o erro de que o Auto Lisp se queixa na soma, na realidade, foi provocado muito antes disso, logo na invocação da função `coluna`. De facto, é visível que essa função é invocada com um ponto cuja coordenada x é `nil` e não um número como devia ser. Esta é a pista que nos permite identificar o erro: ele tem de estar na própria função `coluna-standard` e, mais especificamente, na expressão que cria as coordenadas que são passadas como argumento da função `coluna`. Se observarmos cuidadosamente essa expressão (que assinalámos a **negrito**) vemos que, de facto, está lá um muito subtil erro: o primeiro argumento da função `xy` não é zero mas sim a letra “ó” maiúsculo.

```
(defun coluna-standard ()
  (coluna (xy O 0) 9 0.5 0.4 0.3 0.3 0.5))
```

Acontece que, em Auto Lisp, qualquer nome que não tenha sido previamente definido tem o valor `nil` e, como o nome constituído pela letra `O` não está definido, a avaliação do primeiro argumento da função `xy` é, na verdade, `nil`. Esta função, como se limita a fazer uma lista com os seus

²²A expressão *error trace* pode ser traduzida para “Rastreio de erros.”

²³Essas funções que foram invocadas antes das nossas revelam que, na verdade, parte da funcionalidade do Visual Lisp está ela própria implementada em Auto Lisp.

argumentos, cria uma lista cujo primeiro elemento é `nil` e cujos restantes elementos são zero. A lista resultante é assim passada de função em função até chegarmos à função `+xy` que, ao tentar fazer a soma, acaba por provocar o erro.

Esta “sessão” de depuração mostra o procedimento habitual para detecção de erros. A partir do momento em que o erro é identificado, é geralmente fácil corrigi-lo mas convém ter presente que o processo de identificação de erros pode ser moroso e frustrante. É também um facto que quanto mais experiência tivermos na detecção de erros, mais rapidamente se detectam novos erros.

3.18 Modelação Tridimensional

Como vimos na secção anterior, o AutoCad disponibiliza um conjunto de operações de desenho (linhas, rectângulos, círculos, etc) que nos permitem facilmente criar representações bidimensionais de objectos, como sejam plantas, alçados e cortes.

Embora até este momento apenas tenhamos utilizado as capacidades de desenho bi-dimensional do AutoCad, é possível irmos mais longe, entrando no que se denomina por *modelação tridimensional*. Esta modelação visa a representação gráfica de linhas, superfícies e volumes no espaço tridimensional.

Nesta secção iremos estudar as operações do AutoCad que nos permitem modelar directamente os objectos tridimensionais.

3.18.1 Sólidos Tridimensionais Pré-Definidos

As versões mais recentes do AutoCad disponibilizam um conjunto de operações pré-definidas que constroem um sólido a partir das suas coordenadas tridimensionais. Embora as operações pré-definidas apenas permitam construir um conjunto muito limitado de sólidos, esse conjunto é suficiente para a elaboração de modelos sofisticados.

As operações pré-definidas para criação de sólidos permitem construir paralelipípedos (comando `box`), cunhas (comando `wedge`), cilindros (comando `cylinder`), cones (comando `cone`), esferas (comando `sphere`), toros (comando `torus`) e pirâmides (comando `pyramid`). Os comandos `cone` e `pyramid` permitem ainda a construção de troncos de cone e troncos de pirâmide através da utilização do parâmetro "Top". Cada um destes comandos aceita várias opções que permitem construir sólidos de diferentes maneiras.²⁴ A Figura 20 mostra um conjunto de sólidos construídos pela avaliação das seguintes expressões:²⁵

²⁴Para se conhecer as especificidades de cada comando recomenda-se a consulta da documentação que acompanha o AutoCad.

²⁵A construção de uma pirâmide exige a especificação simultânea do raio e ângulo da

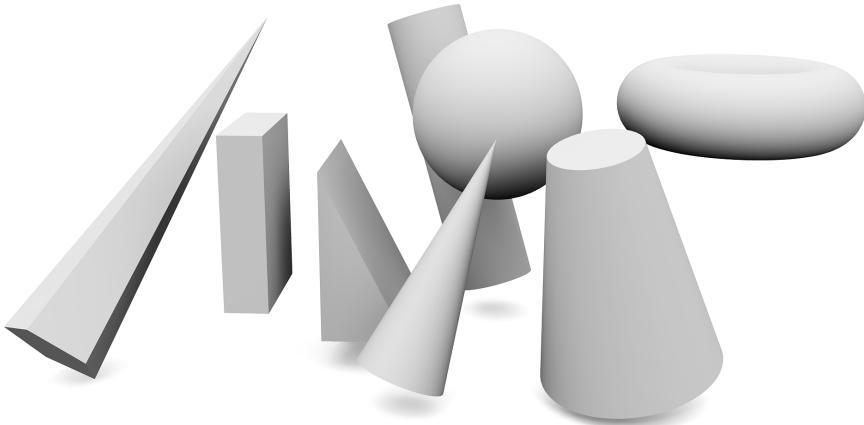


Figura 20: Sólidos primitivos em AutoCad.

```
(command ".box" (xyz 1 1 1) (xyz 3 4 5))
(command ".wedge" (xyz 4 2 0) (xyz 6 6 4))
(command ".cone" (xyz 6 0 0) 1
    "_Axis" (xyz 8 1 5))
(command ".cone" (xyz 11 1 0) 2 "_Top" 1
    "_Axis" (xyz 10 0 5))
(command ".sphere" (xyz 8 4 5) 2)
(command ".cylinder" (xyz 8 7 0) 1 "_Axis" (xyz 6 8 7))
(command ".pyramid" "_Sides" 5 (xyz 1 6 1) (raio&angulo 1 0)
    "_Axis" (xyz 2 7 9))
(command ".torus" (xyz 13 6 5) 2 1)
```

Note-se que alguns dos sólidos, nomeadamente o paralelipípedo e a cuña só podem ser construídos com a base paralela ao plano *XY*. Esta não é uma verdadeira limitação do AutoCad, pois é possível alterar a orientação do plano *XY* através da manipulação do sistema de coordenadas *UCS-user coordinate system*.

A partir dos comandos disponibilizados pelo AutoCad é possível definir funções Auto Lisp que simplifiquem a sua utilização. Embora o AutoCad permita vários modos diferentes de se criar um sólido, esses modos estão mais orientados para facilitar a vida ao utilizador do AutoCad do que propriamente para o programador de Auto Lisp. Para este último, é preferível dispor de uma função que, a partir de um conjunto de parâmetros simples, invoca o comando correspondente em AutoCad, especificando automaticamente as opções adequadas para a utilização desses parâmetros.

Para vermos um exemplo, consideremos a criação de um cilindro. Embora o AutoCad permita construir o cilindro de várias maneiras diferentes, cada uma empregando diferentes parâmetros, podemos considerar que os

base. Tal como explicado na secção 3.11, a função *raio&angulo* constrói essa especificação a partir dos valores do raio e do ângulo.

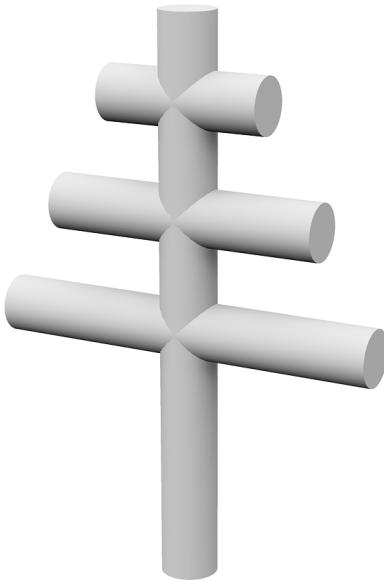


Figura 21: Uma cruz papal.

únicos parâmetros relevantes são o centro da base, o raio da base e o centro do topo. Estes parâmetros são suficientes para especificar completamente um cilindro, permitindo orientá-lo no espaço como muito bem entenderemos. Assim, para o programador de Auto Lisp, basta-lhe invocar o comando AutoCad `cylinder` seleccionando o modo de construção que facilita o uso deste parâmetros. É precisamente isso que faz a seguinte função:

```
(defun cilindro (centro-base raio centro-topo)
  (command "_cylinder" centro-base raio "_Axis" centro-topo))
```

Usando esta função é agora mais simples definir outras estruturas mais complexas. Por exemplo, uma *cruz papal* define-se pela união de três cilindros horizontais de comprimento progressivamente decrescente dispostos ao longo de um cilindro vertical, tal como se pode ver na Figura 21. É de salientar que os cilindros têm todos o mesmo raio e que o seu comprimento e posicionamento é função desse raio. Em termos de proporção, o cilindro vertical da cruz papal tem um comprimento igual a 20 raios, enquanto que os cilindros horizontais possuem comprimentos iguais a 14, 10 e 6 raios e o seu eixo está posicionado a uma altura igual a 9, 13 e 17 raios. Estas proporções são implementadas pela seguinte função:

```
(defun cruz-papal (p raio)
  (cilindro p
    raio
    (+xyz p 0 0 (* 20 raio)))
  (cilindro (+xyz p (* -7 raio) 0 (* 9 raio))
    raio
    (+xyz p (* +7 raio) 0 (* 9 raio)))
  (cilindro (+xyz p (* -5 raio) 0 (* 13 raio))
    raio
    (+xyz p (* +5 raio) 0 (* 13 raio)))
  (cilindro (+xyz p (* -3 raio) 0 (* 17 raio))
    raio
    (+xyz p (* +3 raio) 0 (* 17 raio))))
```

Um outro exemplo que vale a pena simplificar é o do tronco de cone. Do ponto de vista geométrico, é possível caracterizarmos um tronco de cone apenas a partir dos pontos correspondentes aos centros da base e do topo e dos raios dessa base e desse topo. Assim, podemos definir:

```
(defun tronco-cone (centro-base raio-base centro-topo raio-topo)
  (command "_.cone" centro-base raio-base
    "_Top" raio-topo
    "_Axis" centro-topo))
```

A título de exemplo, consideremos a cruzeta representada na Figura 22, constituída por seis troncos de cone idênticos, com a base centrada num mesmo ponto e com os topos posicionados ao longo dos eixos coordenados.

Para a modelação deste sólido vamos parametrizar o posicionamento p da base dos troncos de cone, bem como as dimensões de cada tronco de cone em termos dos raios da base r_b e do topo r_t e ainda do comprimento c . Assim, temos:

```
(defun cruzeta (p rb rt c)
  (tronco-cone p rb (+x p c) rt)
  (tronco-cone p rb (+y p c) rt)
  (tronco-cone p rb (+z p c) rt)
  (tronco-cone p rb (+x p (- c)) rt)
  (tronco-cone p rb (+y p (- c)) rt)
  (tronco-cone p rb (+z p (- c)) rt))
```

O caso do cone é semelhante ao do tronco de cone, mas em que evitamos especificar o raio do topo, i.e.:

```
(defun cone (centro-base raio-base centro-topo)
  (command "_.cone" centro-base raio-base "_Axis" centro-topo))
```

Em alternativa, se considerar o cone como um caso particular do tronco de cone em que o raio do topo é zero, podemos também definir:

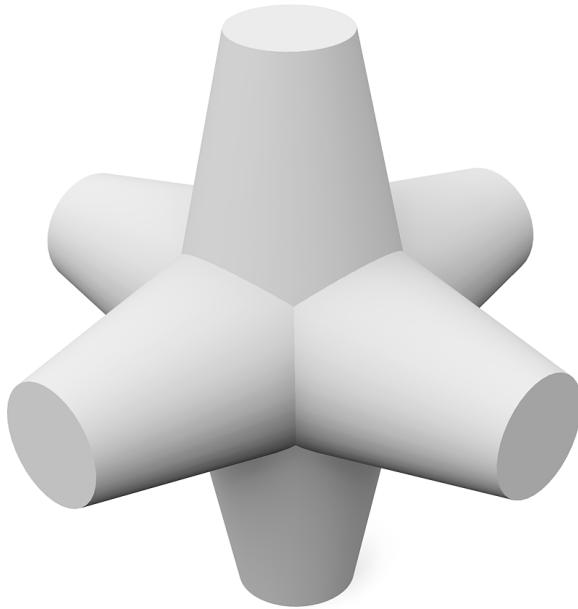


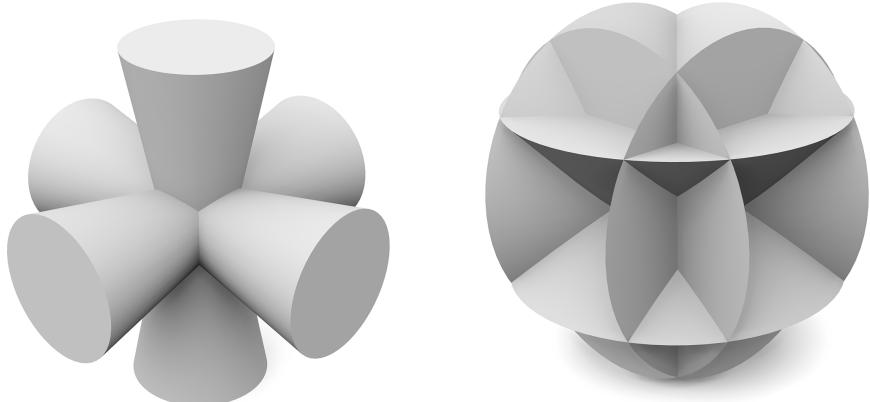
Figura 22: Uma sobreposição ortogonal de troncos de cone.

```
(defun cone (centro-base raio-base centro-topo)
  (tronco-cone centro-topo raio-base centro-topo 0))
```

O caso da esfera é ainda mais simples:

```
(defun esfera (centro raio)
  (command "_.sphere" centro raio))
```

Exercicio 3.23 Empregando a função `cruzeta` responsável pela criação do modelo apresentado na Figura 22, determine valores aproximados para os parâmetros de modo a conseguir reproduzir os seguintes modelos:

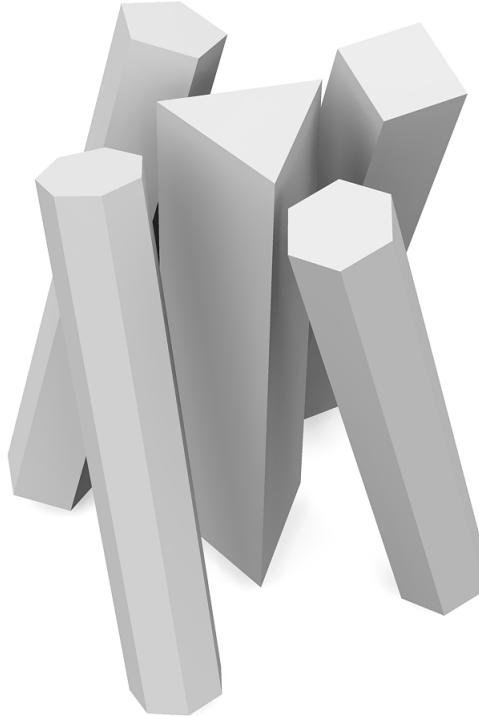


Exercício 3.24 Defina uma função denominada `prisma` que cria um sólido prismático regular. A função deverá receber o número de lados do prisma, as coordenadas tridimensionais do centro da base do prisma, a distância do centro da base a cada vértice, o ângulo de rotação da base do prisma e as coordenadas tridimensionais do centro do topo do prisma.

A título de exemplo, considere as expressões

```
(prisma 3 (xyz 0 0 0) 0.4 0 (xyz 0 0 5))
(prisma 5 (xyz -2 0 0) 0.4 0 (xyz -1 1 5))
(prisma 4 (xyz 0 2 0) 0.4 0 (xyz 1 1 5))
(prisma 6 (xyz 2 0 0) 0.4 0 (xyz 1 -1 5))
(prisma 7 (xyz 0 -2 0) 0.4 0 (xyz -1 -1 5))
```

cuja avaliação produz a imagem seguinte:



3.18.2 Coordenadas Cilíndricas

Vimos nas secções anteriores alguns exemplos da utilização dos sistemas de coordenadas rectangulares e polares. Ficou também claro que uma escolha judiciosa do sistema de coordenadas pode simplificar bastante a solução de um problema geométrico.

Para a modelação tridimensional, para além das coordenadas rectangulares e polares, é ainda usual empregarem-se dois outros sistemas de coordenadas que iremos ver de seguida: as *coordenadas cilíndricas* e as *coordenadas esféricas*.

Tal como podemos verificar na Figura 23, um ponto, em coordenadas cilíndricas, caracteriza-se por um raio ρ assente no plano $z = 0$, um ângulo ϕ que esse raio faz com o eixo x , e por uma cota z . É fácil de ver que o raio

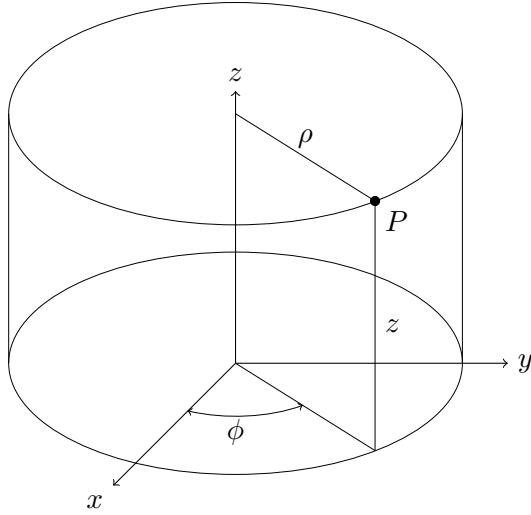


Figura 23: Coordenadas cilíndricas.

e o ângulo correspondem às coordenadas polares da projecção do ponto no plano $z = 0$.

A partir da Figura 23 é fácil vermos que, dado um ponto (ρ, ϕ, z) em coordenadas cilíndricas, o mesmo ponto em coordenadas rectangulares é

$$(\rho \cos \phi, \rho \sin \phi, z)$$

De igual modo, dado um ponto (x, y, z) em coordenadas rectangulares, o mesmo ponto em coordenadas cilíndricas é

$$(\sqrt{x^2 + y^2}, \text{atan} \frac{y}{x}, z)$$

Estas equivalências permitem-nos definir uma função que constrói pontos em coordenadas cilíndricas empregando as coordenadas rectangulares como representação canónica:

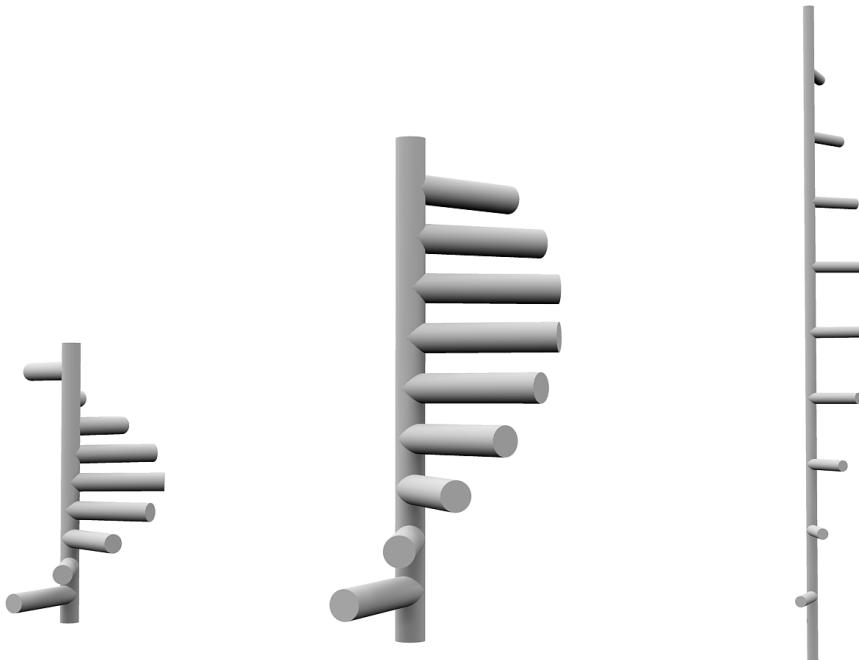
```
(defun cil (ro fi z)
  (xyz (* ro (cos fi))
        (* ro (sin fi))
        z))
```

No caso de pretendermos simplesmente somar a um ponto um deslocamento em coordenadas cilíndricas, podemos tirar partido da função `+xyz` e definir:

```
(defun +cil (p ro fi z)
  (+xyz p
        (* ro (cos fi))
        (* ro (sin fi))
        z))
```

Exercicio 3.25 Defina os selectores `cil-ro`, `cil-fi` e `cil-z` que devolvem, respectivamente, os componentes ρ , ϕ e z de um ponto construído pelo construtor `cil`.

Exercicio 3.26 Defina uma função `escadas` capaz de construir uma escada em hélice. A imagem seguinte mostra três exemplos destas escadas.



Como se pode ver pela imagem anterior, a escada é constituída por um cilindro central no qual se apoiam 10 degraus cilíndricos. Para facilitar a experimentação considere que o cilindro central é de raio r . Os degraus são iguais ao cilindro central, possuem 10 raios de comprimento e estão dispostos a alturas progressivamente crescentes. Cada degrau está a uma altura h em relação ao degrau anterior e, visto em planta, faz um ângulo α com o degrau anterior.

A função `escada` deverá receber as coordenadas do centro da base do cilindro central, o raio r , a altura h e o ângulo α . A título de exemplo, considere que as três escadas anteriores foram construídas pelas seguintes invocações:

```
(escada (xyz 0 0 0) 1.0 3 (/ pi 6))
(escada (xyz 0 40 0) 1.5 5 (/ pi 9))
(escada (xyz 0 80 0) 0.5 6 (/ pi 8))
```

3.18.3 Coordenadas Esféricas

Tal como podemos ver na Figura 24, um ponto, em coordenadas esféricas (também denominadas polares), caracteriza-se pelo comprimento ρ do raio vector, por um ângulo ϕ (denominado *longitude* ou *azimute*) que a projecção desse vector no plano $z = 0$ faz com o eixo x e por ângulo ψ (denominado *colatitude*²⁶, *zénite* ou *ângulo polar*) que o vector faz com o eixo z .

²⁶A colatitude é, como é óbvio, o ângulo complementar à latitude, i.e., a diferença entre o pólo ($\frac{\pi}{2}$) e a latitude.

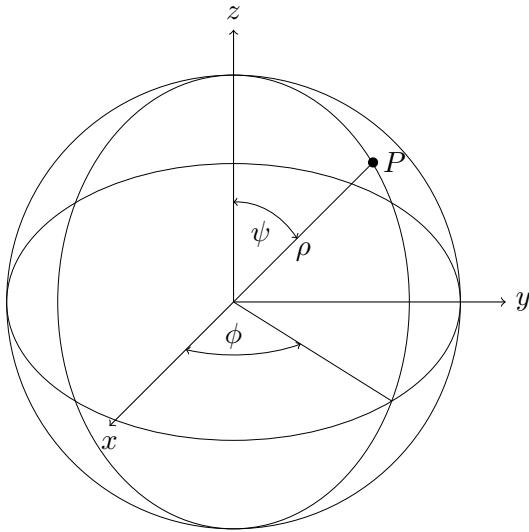


Figura 24: Coordenadas Esféricas

Dado um ponto (ρ, ϕ, ψ) em coordenadas esféricas, o mesmo ponto em coordenadas Cartesianas é

$$(\rho \sin \psi \cos \phi, \rho \sin \psi \sin \phi, \rho \cos \psi)$$

De igual modo, dado um ponto (x, y, z) em coordenadas Cartesianas, o mesmo ponto em coordenadas esféricas é

$$(\sqrt{x^2 + y^2 + z^2}, \text{atan} \frac{y}{x}, \text{atan} \frac{\sqrt{x^2 + y^2}}{z})$$

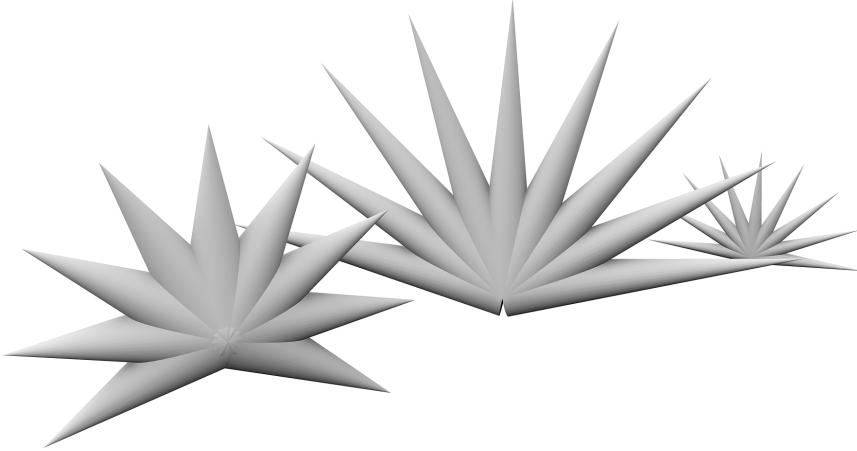
Tal como fizemos para as coordenadas cilíndricas, vamos definir o construtor de coordenadas esféricas e a soma de um ponto a um deslocamento em coordenadas esféricas:

```
(defun esf (ro fi psi)
  (xyz (* ro (sin psi) (cos fi))
        (* ro (sin psi) (sin fi))
        (* ro (cos psi)))))

(defun +esf (p ro fi psi)
  (+xyz p
    (* ro (sin psi) (cos fi))
    (* ro (sin psi) (sin fi))
    (* ro (cos psi))))
```

Exercício 3.27 Defina os selectores `esf-ro`, `esf-fi` e `esf-ps` que devolvem, respectivamente, os componentes ρ , ϕ e ψ de um ponto construído pelo construtor `esf`.

Exercicio 3.28 O corte de cabelo de estilo *Moicano* foi muito utilizado no período *punk*. Ele consiste em fixar os cabelos em bicos que se dispõem em forma de leque ou crista, tal como se esquematiza na seguinte imagem.



Defina a função `moicano`, de parâmetros P , r , c , ϕ e Δ_ψ , que constrói 9 cones de comprimento c e raio da base r , todos com a base centrada num mesmo ponto P e com os eixos inclinados uns em relação aos outros por um ângulo Δ_ψ , e dispostos ao longo de um plano que faz um ângulo ϕ com o plano XZ .

A título de exemplo, considere que a figura anterior foi produzida pelas seguintes invocações:

```
(moicano (xyz 0 0 0) 1.0 10 (/ pi 2) (/ pi 6))
(moicano (xyz 0 15 0) 0.5 15 (/ pi 3) (/ pi 9))
(moicano (xyz 0 30 0) 1.5 6 (/ pi 4) (/ pi 8))
```

3.18.4 Modelação de Colunas Dóricas

A modelação tri-dimensional tem a virtude de nos permitir criar entidades geométricas muito mais realistas do que meros aglomerados de linhas a representarem vistas dessas entidades. A título de exemplo, reconsideremos a coluna Dórica que apresentámos na secção 3.14. Nessa secção desenvolvemos um conjunto de funções cuja invocação criava uma vista frontal dos componentes da coluna Dórica. Apesar dessas vistas serem úteis, é ainda mais útil poder modelar directamente a coluna como uma entidade tri-dimensional.

Nesta secção vamos empregar algumas das operações mais relevantes para a modelação tri-dimensional de colunas, em particular, a criação de troncos de cone para modelar o fuste e o coxim e a criação de paralelipípedos para modelar o ábaco.

Anteriormente, as nossas “colunas” estavam dispostas no plano XY , com as colunas a “crescer” ao longo do eixo Y . Agora, será apenas a base das colunas que ficará assente no plano XY : o corpo das colunas irá

desenvolver-se ao longo do eixo dos Z . Embora fosse trivial empregar outro arranjo dos eixos do sistema de coordenadas, este é aquele que é mais próximo da realidade.

À semelhança de inúmeras outras operações do AutoCad, cada uma das operações de modelação de sólidos do AutoCad permite vários modos diferentes de invocação. No caso da operação de modelação de troncos de cone—cone—o modo que nos é mais conveniente é aquele em que a operação recebe as coordenadas do centro da base do cone e o raio dessa base, de seguida, o raio do topo do cone (com a opção Top radius) e, finalmente, a altura do tronco.

Tendo isto em conta, podemos redefinir a operação que constrói o fuste tal como se segue:

```
(defun fuste (p a-fuste r-base r-topo)
  (command "_.cone" p r-base
           "_t" r-topo a-fuste))
```

Do mesmo modo, a operação que constrói o coxim ficará com a forma:

```
(defun coxim (p a-coxim r-base r-topo)
  (command "_.cone" p r-base
           "_t" r-topo a-coxim))
```

Finalmente, no que diz respeito ao ábaco—o paralelipípedo que é colocado no topo da coluna—temos várias maneiras de o especificarmos. A mais directa consiste em indicar os dois cantos do paralelipípedo. Uma outra, menos directa, consiste em indicar o centro do paralelipípedo, seguido do tamanho dos seus lados. Por agora, vamos seguir a mais directa:

```
(defun abaco (p a-abaco l-abaco)
  (command "_.box"
           (+xyz p (/ l-abaco -2.0) (/ l-abaco -2.0) 0)
           (+xyz p (/ l-abaco +2.0) (/ l-abaco +2.0) a-abaco)))
```

Exercicio 3.29 Implemente a função abaco mas empregando a criação de um paralelipípedo centrado num ponto seguido da especificação do seu comprimento, largura e altura.

Finalmente, falta-nos implementar a função coluna que, à semelhança do que fazia no caso bi-dimensional, invoca sucessivamente as funções fuste, coxim e abaco mas, agora, elevando progressivamente a coordenada z :

```
(defun coluna (p
                a-fuste r-base-fuste
                a-coxim r-base-coxim
                a-abaco l-abaco)
  (fuste p a-fuste r-base-fuste r-base-coxim)
  (coxim (+z p a-fuste) a-coxim r-base-coxim (/ l-abaco 2.0))
  (abaco (+z p (+ a-fuste a-coxim)) a-abaco l-abaco))
```

Com estas redefinições, podemos agora repetir as colunas que desenhámos na secção 3.15 e que apresentámos na Figura 18, mas, agora, gerando

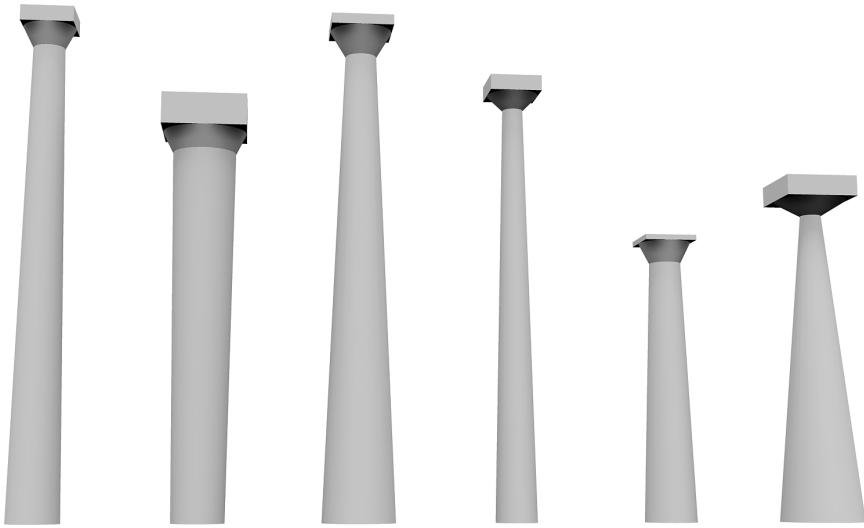


Figura 25: Modelação tri-dimensional das variações de colunas dóricas.

uma imagem tridimensional dessas mesmas colunas, tal como apresentamos na Figura 25:

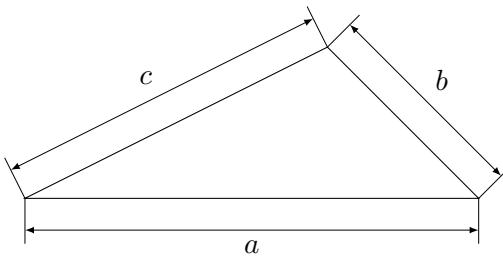
```
(coluna (xyz 0 0 0) 9 0.5 0.4 0.3 0.3 1.0)
(coluna (xyz 3 0 0) 7 0.5 0.4 0.6 0.6 1.6)
(coluna (xyz 6 0 0) 9 0.7 0.5 0.3 0.2 1.2)
(coluna (xyz 9 0 0) 8 0.4 0.3 0.2 0.3 1.0)
(coluna (xyz 12 0 0) 5 0.5 0.4 0.3 0.1 1.0)
(coluna (xyz 15 0 0) 6 0.8 0.3 0.2 0.4 1.4)
```

4 Funções

As funções são um dos componentes mais importantes do Auto Lisp e são a ferramenta fundamental para implementarmos a solução dos nossos problemas arquitecturais. Nesta secção iremos discutir algumas características adicionais das funções.

4.1 Variáveis Locais

Consideremos o seguinte triângulo



e tentemos definir uma função Auto Lisp que calcula a área do triângulo a partir dos parâmetros a , b e c .

Uma das formas de calcular a área do triângulo baseia-se na famosa fórmula de Heron:²⁷

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

em que é s o semi-perímetro do triângulo:

$$s = \frac{a+b+c}{2}$$

Ao tentarmos usar a fórmula de Heron para definir a correspondente função Auto Lisp, somos confrontados com uma dificuldade: a fórmula está escrita (também) em termos do semi-perímetro s , mas s não é um parâmetro, é um valor *derivado* dos parâmetros do triângulo.

Uma das maneira de resolvemos este problema consiste em eliminar a variável s , substituindo-a pelo seu significado:

$$A = \sqrt{\frac{a+b+c}{2} \left(\frac{a+b+c}{2} - a \right) \left(\frac{a+b+c}{2} - b \right) \left(\frac{a+b+c}{2} - c \right)}$$

A partir desta fórmula já é possível definir a função pretendida:

```
(defun area-triangulo (a b c)
  (sqrt (* (/ (+ a b c) 2)
            (- (/ (+ a b c) 2) a)
            (- (/ (+ a b c) 2) b)
            (- (/ (+ a b c) 2) c))))
```

Infelizmente, esta definição tem dois problemas. O primeiro é que se perdeu a correspondência entre a fórmula original e a definição da função, tornando mais difícil reconhecer que a função está a implementar a fórmula de Heron. O segundo problema é que a função é obrigada a repetidamente empregar a expressão $(/ (+ a b c) 2)$, o que não só representa um desperdício do nosso esforço, pois tivemos de a escrever quatro vezes, como

²⁷Heron de Alexandria foi um importante matemático e engenheiro Grego do primeiro século depois de Cristo a quem se atribuem inúmeras descobertas e invenções, incluindo a máquina a vapor e a seringa.

representa um desperdício de cálculo, pois a expressão é calculada quatro vezes, embora saibamos que ela produz sempre o mesmo resultado.

Para resolver estes problemas, o Auto Lisp permite a utilização de *variáveis locais*. Uma variável local é uma variável que apenas tem significado no contexto de uma dada função e que é usada para calcular valores intermediários como o do semi-perímetro s . Empregando uma variável local, podemos reescrever a função `area-triangulo` da seguinte forma:

```
(defun area-triangulo (a b c / s)
  (setq s (/ (+ a b c) 2))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

Na definição anterior há dois aspectos novos que vamos agora discutir: a *declaração* da variável local s e a *atribuição* dessa variável através do operador `setq`.

4.1.1 Declaração

Para se indicar que uma função vai usar variáveis locais, estas têm de ser *declaradas* conjuntamente com a lista de parâmetros da função. Para isso, vamos agora ampliar a sintaxe da definição de funções que apresentámos anteriormente (na secção 2.11) de modo a incorporar a declaração de variáveis locais:

```
(defun nome (parâmetro1 ... parâmetron
  / variável1 ... variávelm)
  corpo)
```

Note-se que as variáveis locais são separadas dos parâmetros por uma barra. É necessário ter o cuidado de não “colar” a barra nem ao último parâmetro nem à primeira variável, sob pena de o Auto Lisp considerar a barra como parte de um dos nomes (e, consequentemente, tratar as variáveis locais como se fossem parâmetros).

Embora seja raro, é perfeitamente possível termos funções sem parâmetros mas com variáveis locais. Neste caso, a sintaxe da função seria simplesmente:

```
(defun nome (/ variável1 ... variávelm)
  corpo)
```

4.1.2 Atribuição

Para além da declaração de uma variável local é ainda necessário atribuir-lhe um valor, i.e., realizar uma operação de *atribuição*. Para isso, o Auto Lisp disponibiliza o operador `setq`, cuja sintaxe é:

```
(setq variável1 expressão1
      ...
      variávelm expressãom)
```

A semântica deste operador consiste simplesmente em avaliar a expressão $expressão_1$ e associar o seu valor ao nome $variável_1$, repetindo este processo para todas as restantes variáveis e valores.

A utilização que fazemos deste operador na função `area-triangulo` é fácil de compreender:

```
(defun area-triangulo (a b c / s)
  (setq s (/ (+ a b c) 2))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

Ao invocarmos a função `area-triangulo`, passando-lhe os argumentos correspondentes aos parâmetros a , b e c , ela começa por introduzir um novo nome— s —que vai existir apenas durante a invocação da função. De seguida, atribui a esse nome o valor que resultar da avaliação da expressão $(/ (+ a b c) 2)$. Finalmente, avalia as restantes expressões do corpo da função. Na prática, é como se a função dissesse: “Sendo $s = \frac{a+b+c}{2}$, calculemos $\sqrt{s(s-a)(s-b)(s-c)}$.”

Há dois detalhes que é importante conhecer relativamente às variáveis locais. O primeiro detalhe, que iremos discutir mais em profundidade na próxima secção, é que se nos esquecermos de declarar a variável, ela será tratada como variável *global*. O segundo detalhe é que se nos esquecermos de atribuir a variável ela fica automaticamente com o valor `nil`. Logicamente, para que uma variável nos seja útil, devemos mudar o seu valor para algo que nos interesse.

4.2 Variáveis Globais

Qualquer referência a um nome que não está declarado localmente implica que esse nome seja tratado como uma *variável global*.²⁸ O nome `pi`, por exemplo, representa a variável $\pi = 3.14159\dots$ e pode ser usado em qualquer ponto dos nossos programas. Por esse motivo, o nome `pi` designa uma variável global.

A declaração de variáveis globais é mais simples que a das variáveis locais: basta uma primeira atribuição, em qualquer ponto do programa, para a variável ficar automaticamente declarada. Assim, se quisermos introduzir uma nova variável global, por exemplo, para definir a *razão de ouro*²⁹

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6180339887$$

²⁸Na realidade, o Auto Lisp permite ainda uma terceira categoria de nomes que não são nem locais, nem globais. Mais à frente discutiremos esse caso.

²⁹Também conhecida por *proporção divina* e *número de ouro*, entre outras designações, e abreviada por ϕ em homenagem a Fineas, escultor Grego responsável pela construção do Partenon onde, supostamente, usou esta proporção. A razão de ouro foi inicialmente in-

basta-nos escrever:

```
(setq razao-de-ouro (/ (+ 1 (sqrt 5)) 2))
```

A partir desse momento, o nome `razao-de-ouro` pode ser referenciado em qualquer ponto dos programas.

É importante referir que o uso de variáveis globais deve ser restrito, na medida do possível, à definição de *constantes*, i.e., variáveis cujo valor nunca muda como, por exemplo, `pi`. Outros exemplos que podem vir a ser úteis incluem `2*pi`, `pi/2`, `4*pi` e `pi/4`, bem como os seus simétricos, que se definem à custa de:

```
(setq 2*pi (* 2 pi))
(setq pi/2 (/ pi 2))
(setq 4*pi (* 4 pi))
(setq pi/4 (/ pi 4))
(setq -pi (- pi))
(setq -2*pi (- 2*pi))
(setq -pi/2 (- pi/2))
(setq -4*pi (- 4*pi))
(setq -pi/4 (- pi/4))
```

O facto de uma variável global ser uma constante implica que a variável é atribuída uma única vez, no momento da sua definição. Esse facto permite-nos usar a variável sabendo sempre qual o valor a que ela está associada. Infelizmente, por vezes é necessário usarmos variáveis globais que não são constantes, i.e., o seu valor muda durante a execução do programa, por acção de diferentes atribuições feitas em locais diferentes e em momentos diferentes. Quando temos variáveis globais que são atribuidas em diversos pontos do nosso programa, o comportamento deste pode tornar-se muito mais difícil de entender pois, na prática, pode ser necessário compreender o funcionamento de *todo* o programa em simultâneo e não apenas função a função, como temos feito. Por este motivo, devemos evitar o uso de variáveis globais que sejam atribuidas em vários pontos do programa.

troduzida por Euclides quando resolveu o problema de dividir um segmento de recta em duas partes de tal forma que a razão entre o segmento de recta e a parte maior fosse igual à razão entre a parte maior e a menor. Se for a o comprimento do parte maior e b o da menor, o problema de Euclides é idêntico a dizer que $\frac{a+b}{a} = \frac{a}{b}$. Daqui resulta que $a^2 - ab - b^2 = 0$ ou que $a = \frac{b \pm \sqrt{b^2 + 4b^2}}{2} = b \frac{1 \pm \sqrt{5}}{2}$. A raiz que faz sentido é, obviamente, $a = b \frac{1 + \sqrt{5}}{2}$ e, consequentemente, a razão de ouro é $\phi = \frac{a}{b} = \frac{1 + \sqrt{5}}{2}$.

4.3 Variáveis Indefinidas

A linguagem Auto Lisp difere da maioria dos outros dialecto de Lisp no tratamento que dá a variáveis *indefinidas*, i.e., variáveis a que nunca foi atribuído um valor.³⁰ Em geral, as linguagens de programação consideram que a avaliação de uma variável indefinida é um erro. No caso da linguagem Auto Lisp, a avaliação de qualquer variável indefinida é simplesmente nil.

A consequência deste comportamento é que existe o potencial de os erros tipográficos passarem despercebidos, tal como é demonstrado no seguinte exemplo:

```
_\$ pi  
3.14159  
_\$ razao-de-ouro  
1.61803  
_\$ (list pi razao-de-ouro)  
(3.14159 nil)
```

Reparemos, no exemplo anterior, que a variável que foi definida não é a mesma que está a ser avaliada pois a segunda tem uma gralha no nome. Em consequência, a expressão que tenta agrupar o pi e a razao-de-ouro está, na realidade, a agrupar 3.14159 com nil.

Este género de erros pode ser extremamente difícil de detectar e, por isso, a maioria das linguagens abandonou este comportamento. O Auto Lisp, por motivos históricos, manteve-o, o que nos obriga a termos de ter muito cuidado para não cometermos erros que depois passem indetectados.

4.4 Proporções de Vitrúvio

A modelação de colunas dóricas que desenvolvemos na secção 3.18.4 permite-nos facilmente construir colunas, bastando para isso indicarmos os valores dos parâmetros relevantes, como a altura e o raio da base do fuste, a altura e raio da base do coxim e a altura e largura do ábaco. Cada um destes parâmetros constitui um grau de liberdade que podemos fazer variar livremente.

Embora seja lógico pensar que quantos mais graus de liberdade tivermos mais flexível é a modelação, a verdade é que um número excessivo de parâmetros pode conduzir a modelos pouco realistas. Esse fenómeno é evidente na Figura 26 onde mostramos uma perspectiva de um conjunto de colunas cujos parâmetros foram escolhidos aleatoriamente.

Na verdade, de acordo com os *cânones* da Ordem Dórica, os diversos parâmetros que regulam a forma de uma coluna devem relacionar-se entre

³⁰Na terminologia original do Lisp, estas variáveis diziam-se *unbound*.

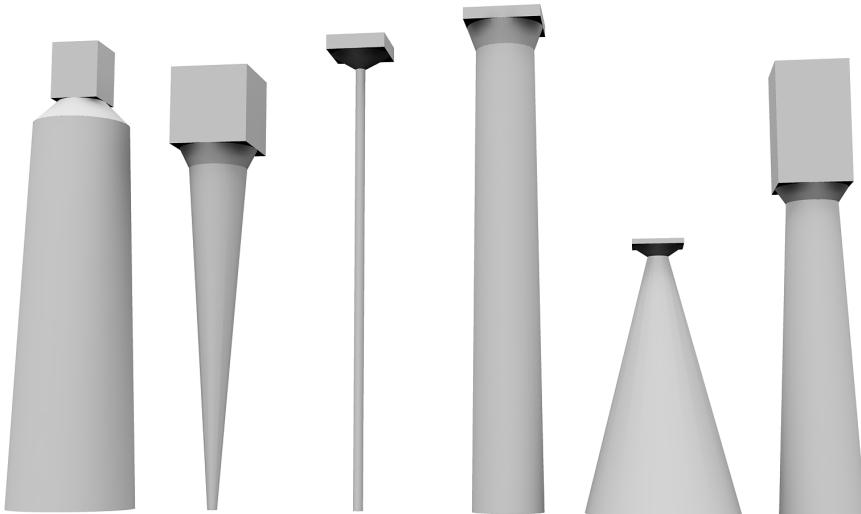


Figura 26: Perspectiva da modelação tri-dimensional de colunas cujos parâmetros foram escolhidos aleatoriamente. Apenas uma das colunas obedece aos *cânones* da Ordem Dórica.

si segundo um conjunto de proporções bem definidas. Vitrúvio,³¹ no seu famoso tratado de arquitectura, considera que essas proporções derivam das proporções do próprio ser humano:

Uma vez que pretendiam erguer um templo com colunas mas não tinham conhecimento das proporções adequadas, [...] mediram o comprimento dum pé de um homem e viram que era um sexto da sua altura e deram à coluna uma proporção semelhante, i.e., fizeram a sua altura, incluindo o capitel, seis vezes a largura da coluna medida na base. Assim, a ordem Dórica obteve a sua proporção e a sua beleza, da figura masculina.

Mais concretamente, Vitrúvio caracteriza as colunas da Ordem Dórica em termos do conceito de *módulo*:

- *A largura das colunas, na base, será de dois modulos e a sua altura, incluindo os capitéis, será de catorze.*

Daqui se deduz que um módulo iguala o raio da base da coluna e que a altura da coluna deverá ser 14 vezes esse raio. Dito de outra forma, o raio da base da coluna deverá ser $\frac{1}{14}$ da altura da coluna.

- *A altura do capitel será de um módulo e a sua largura de dois módulos e um sexto.*

³¹Vitrúvio foi um escritor, arquitecto e engenheiro romano que viveu no século um antes de Cristo e autor do único tratado de arquitectura que sobreviveu a antiguidade.

Isto implica que a altura do coxim somado à do ábaco será um módulo, ou seja, igual ao raio da base da coluna e a largura do ábaco será de $2\frac{1}{6}$ módulos ou $\frac{13}{6}$ do raio. Juntamente com o facto de a altura da coluna ser de 14 módulos, implica ainda que a altura do fuste será de 13 vezes o raio.

- *Seja a altura do capitel dividida em três partes, das quais uma formará o ábaco com o seu cimáteo, o segundo o équino (coxim) com os seus anéis e o terceiro o pescoço.*

Isto quer dizer que o ábaco tem uma altura de um terço de um módulo, ou seja $\frac{1}{3}$ do raio da base, e o coxim terá os restantes dois terços, ou seja, $\frac{2}{3}$ do raio da base.

Estas considerações levam-nos a poder determinar o valor de alguns dos parâmetros de desenho das colunas dóricas em termos do raio da base do fuste. Em termos de implementação, isso quer dizer que os parâmetros da função passam a ser variáveis locais cuja atribuição é feita aplicando as proporções estabelecidas por Vitrúvio ao parâmetro `r-base-fuste`. A definição da função fica então:

```
(defun coluna (p
    r-base-fuste r-base-coxim
    / a-fuste a-coxim a-abaco l-abaco)
  (setq a-fuste (* 13 r-base-fuste)
        a-coxim (* (/ 2.0 3) r-base-fuste)
        a-abaco (* (/ 1.0 3) r-base-fuste)
        l-abaco (* (/ 13.0 6) r-base-fuste))
  (fuste p a-fuste r-base-fuste r-base-coxim)
  (coxim (+z p a-fuste) a-coxim r-base-coxim (/ l-abaco 2.0))
  (abaco (+z p (+ a-fuste a-coxim)) a-abaco l-abaco))
```

Usando esta função já é possível desenhar colunas que se aproximam mais do padrão dórico (tal como estabelecido por Vitrúvio). A Figura 27 apresenta as colunas desenhadas pelas seguintes invocações:

```
(coluna (xyz 0 0 0) 0.3 0.2)
(coluna (xyz 3 0 0) 0.5 0.3)
(coluna (xyz 6 0 0) 0.4 0.2)
(coluna (xyz 9 0 0) 0.5 0.4)
(coluna (xyz 12 0 0) 0.5 0.5)
(coluna (xyz 15 0 0) 0.4 0.7)
```

As proporções de Vitrúvio permitiram-nos reduzir o número de parâmetros independentes de uma coluna Dórica a apenas dois: o raio da base do fuste e o raio da base do coxim. Contudo, não parece correcto que estes parâmetros sejam totalmente independentes pois isso permite construir colunas aberrantes, em que o topo do fuste é mais largo do que a base, tal como acontece com a coluna mais à direita na Figura 27.



Figura 27: Variações de colunas dóricas segundo as proporções de Vitrúvio.

Na verdade, a caracterização da Ordem Dórica que apresentámos encontra-se incompleta pois, acerca das proporções das colunas, Vitrúvio afirmou ainda que:

A diminuição no topo de uma coluna parece ser regulada segundo os seguintes princípios: se uma coluna tem menos de quinze pés, divida-se a largura na base em seis partes e usem-se cinco dessas partes para formar a largura no topo. Se a coluna tem entre quinze e vinte pés, divida-se a largura na base em seis partes e meio e usem-se cinco e meio dessas partes para a largura no topo da coluna. Se a coluna tem entre vinte e trinta pés, divida-se a largura na base em sete partes e faça-se o topo diminuido medir seis delas. Uma coluna de trinta a quarenta pés deve ser dividida na base em sete partes e meia e, no princípio da diminuição, deve ter seis partes e meia no topo. Colunas de quarenta a cinquenta pés devem ser divididas em oito partes e diminuidas para sete delas no topo da coluna debaixo do capitel. No caso de colunas mais altas, determine-se proporcionalmente a diminuição com base nos mesmos princípios. [Vitrúvio, Os Dez Livros da Arquitectura, Livro III, Cap. 3.1]

Estas considerações de Vitrúvio permitem-nos determinar a razão entre o topo e a base de uma coluna em função da sua altura em pés.³²

Consideremos então a definição de uma função, que iremos denominar de **raio-topo-fuste**, que recebe como parâmetros a largura da base da

³²O pé foi a unidade fundamental de medida durante inúmeros séculos, mas a sua real dimensão variou ao longo do tempo. O comprimento do pé internacional é de 304.8 milímetros e foi estabelecido, por acordo, em 1958. Antes disso, vários outros comprimentos foram usados, como o pé Dórico de 324 milímetros, os pés Jónico e Romano de 296 milímetros, o pé Ateniense de 315 milímetros, os pés Egípcio e Fenício de 300 milímetros, etc.

coluna e a altura da coluna e devolve como resultado a largura do topo da coluna.

Uma tradução literal das considerações de Vitrúvio para Lisp permite-nos começar por escrever:

```
(defun raio-topo-fuste (raio-base altura)
  (cond ((< altura 15) (* (/ 5.0 6.0) raio-base))
        ...))
```

O fragmento anterior corresponde, obviamente, à afirmação: *se uma coluna tem menos de quinze pés, divida-se a largura na base em seis partes e usem-se cinco dessas partes para formar a largura no topo*. No caso de a coluna não ter menos de quinze pés, então passamos ao caso seguinte: *se a coluna tem entre quinze e vinte pés, divida-se a largura na base em seis partes e meio e usem-se cinco e meio dessas partes para a largura no topo da coluna*. A tradução deste segundo caso permite-nos escrever:

```
(defun raio-topo-fuste (raio-base altura)
  (cond ((< altura 15)
         (* (/ 5.0 6.0) raio-base))
        ((and (>= altura 15) (< altura 20))
         (* (/ 5.5 6.5) raio-base))
        ...))
```

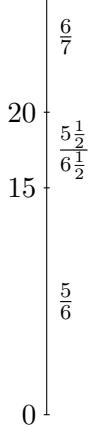
Uma análise cuidadosa das duas cláusulas anteriores mostra que, na realidade, estamos a fazer testes a mais na segunda cláusula. De facto, se conseguimos chegar à segunda cláusula é porque a primeira é falsa, i.e., a altura não é menor que 15 e, portanto, é maior ou igual a 15. Nesse caso, é inútil estar a testar novamente se a altura é maior ou igual a 15. Assim, podemos simplificar a função e escrever:

```
(defun raio-topo-fuste (raio-base altura)
  (cond ((< altura 15) (* (/ 5.0 6.0) raio-base))
        ((< altura 20) (* (/ 5.5 6.5) raio-base))
        ...))
```

A continuação da tradução levar-nos-á, então, a:

```
(defun raio-topo-fuste (raio-base altura)
  (cond ((< altura 15) (* (/ 5.0 6.0) raio-base))
        ((< altura 20) (* (/ 5.5 6.5) raio-base))
        ((< altura 30) (* (/ 6.0 7.0) raio-base))
        ((< altura 40) (* (/ 6.5 7.5) raio-base))
        ((< altura 50) (* (/ 7.0 8.0) raio-base))
        ...))
```

O problema agora é que Vitrúvio deixou a porta aberta para colunas arbitrariamente altas, dizendo simplesmente que, *no caso de colunas mais altas, determine-se proporcionalmente a diminuição com base nos mesmos princípios*.



Para percebermos claramente de que princípios estamos a falar, consideremos a evolução da relação entre o topo e a base das colunas que é visível na imagem lateral.

A razão entre o raio do topo da coluna e o raio da base da coluna é, tal como se pode ver na margem (e já era possível deduzir da função raio-topo-fuste), uma sucessão da forma

$$\frac{5}{6}, \frac{5\frac{1}{2}}{6\frac{1}{2}}, \frac{6}{7}, \frac{6\frac{1}{2}}{7\frac{1}{2}}, \frac{7}{8}, \dots$$

Torna-se agora óbvio que, para colunas mais altas, “os mesmos princípios” de que Vitrúvio fala se resumem a, para cada 10 pés adicionais, somar $\frac{1}{2}$ quer ao numerador, quer ao denominador. No entanto, é importante reparar que este princípio só pode ser aplicado a partir dos 15 pés de altura, pois o primeiro intervalo é maior que os restantes. Assim, temos de tratar de forma diferente as colunas até aos 15 pés e, daí para a frente, basta subtrair 20 pés à altura e determinar a divisão inteira por 10 para saber o número de vezes que precisamos de somar $\frac{1}{2}$ quer ao numerador quer ao denominador de $\frac{6}{7}$.

É este “tratar de forma diferente” um caso e outro que, mais uma vez, sugere a necessidade de um mecanismo de selecção: é necessário distinguir dois casos e reagir em conformidade para cada um. No caso da coluna de Vitrúvio, se a coluna tem uma altura a até 15 pés, a razão entre o topo e a base é $r = \frac{5}{6}$; se a altura a não é inferior a 15 pés, a razão r entre o topo e a base será:

$$r = \frac{6 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}}{7 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}}$$

A título de exemplo, consideremos uma coluna com 43 pés de altura. A divisão inteira de $43 - 20$ por 10 é 2 portanto temos de somar $2 \cdot \frac{1}{2} = 1$ ao numerador e denominador de $\frac{6}{7}$, obtendo $\frac{7}{8} = 0.875$.

Para um segundo exemplo, nos limites do absurdo, consideremos uma coluna da altura do Empire State Building, i.e., com 449 metros de altura. Um pé, na ordem Dórica, media 324 milímetros pelo que em 449 metros existem $449/0.324 \approx 1386$ pés. A divisão inteira de $1386 - 20$ por 10 é 136. A razão entre o topo e a base desta hipotética coluna será então de $\frac{6+136/2}{7+136/2} = \frac{74}{75} = 0.987$. Este valor, por ser muito próximo da unidade, mostra que a coluna seria praticamente cilíndrica.

Com base nestas considerações, podemos agora definir uma função que, dado um número inteiro representando a altura da coluna em pés, calcula a razão entre o topo e a base da coluna. Antes, contudo, convém simplificar a fórmula para as colunas com altura não inferior a 15 pés. Assim,

$$r = \frac{6 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}}{7 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}} = \frac{12 + \lfloor \frac{a-20}{10} \rfloor}{14 + \lfloor \frac{a-20}{10} \rfloor} = \frac{12 + \lfloor \frac{a}{10} \rfloor - 2}{14 + \lfloor \frac{a}{10} \rfloor - 2} = \frac{10 + \lfloor \frac{a}{10} \rfloor}{12 + \lfloor \frac{a}{10} \rfloor}$$

A definição da função fica então:

```
(defun raio-topo-fuste (raio-base altura / divisoes)
  (setq divisoes (fix (/ altura 10)))
  (if (< altura 15)
      (* (/ 5.0 6.0)
          raio-base)
      (* (/ (+ 10.0 divisoes)
             (+ 12.0 divisoes))
          raio-base)))
```

Esta é a última relação que nos falta para especificarmos completamente o desenho de uma coluna dórica de acordo com as proporções referidas por Vitrúvio no seu tratado de arquitectura. Vamos considerar, para este desenho, que vamos fornecer as coordenadas do centro da base da coluna e a sua altura. Todos os restantes parâmetros serão calculados em termos destes. Eis a definição da função:

```
(defun coluna-dorica (p altura /
                        a-fuste r-base-fuste
                        a-coxim r-base-coxim
                        a-abaco l-abaco)
  (setq r-base-fuste (/ altura 14.0)
        r-base-coxim (raio-topo-fuste r-base-fuste altura)
        a-fuste (* 13.0 r-base-fuste)
        a-coxim (* (/ 2.0 3) r-base-fuste)
        a-abaco (* (/ 1.0 3) r-base-fuste)
        l-abaco (* (/ 13.0 6) r-base-fuste))
  (fuste p a-fuste r-base-fuste r-base-coxim)
  (coxim (+z p a-fuste) a-coxim r-base-coxim (/ l-abaco 2.0))
  (abaco (+z p (+ a-fuste a-coxim)) a-abaco l-abaco))
```

O seguinte exemplo de utilização da função produz a sequência de colunas apresentadas na Figura 28:³³

```
(coluna-dorica (xy 0 0) 10)
(coluna-dorica (xy 10 0) 15)
(coluna-dorica (xy 20 0) 20)
(coluna-dorica (xy 30 0) 25)
(coluna-dorica (xy 40 0) 30)
(coluna-dorica (xy 50 0) 35)
```

Exercicio 4.1 A função `raio-topo-fuste` calcula o valor da variável local `divisoes` mesmo quando o parâmetro `altura` é menor ou igual a 15. Redefina a função de modo a que a variável `divisoes` só seja definida valor quando o seu valor é realmente necessário.

³³Note-se que, agora, a altura da coluna tem de ser especificada em pés dóricos.

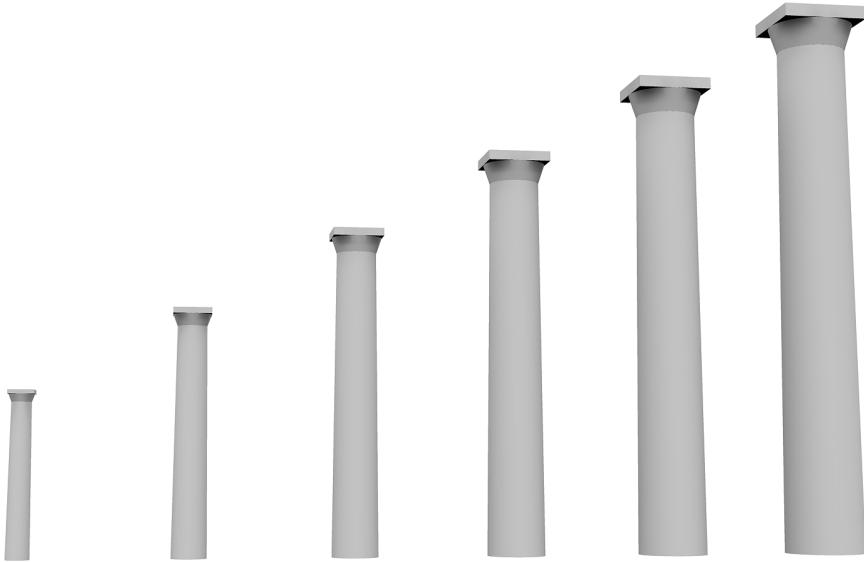


Figura 28: Variações de colunas dóricas segundo as proporções de Vitrúvio.

4.5 Recursão

Vimos que as nossas funções, para fazerem algo útil, precisam de invocar outras funções. Por exemplo, se já tivermos a função que calcula o quadrado de um número e pretendermos definir a função que calcula o cubo de um número, podemos facilmente fazê-lo à custa do quadrado e de uma multiplicação adicional, i.e.:

```
(defun cubo (x)
  (* (quadrado x) x))
```

Do mesmo modo, podemos definir a função quarta-potencia à custa do cubo e de uma multiplicação adicional, i.e.:

```
(defun quarta-potencia (x)
  (* (cubo x) x))
```

Como é óbvio, podemos continuar a definir sucessivamente novas funções para calcular potências crescentes, mas isso não só é moroso como será sempre limitado. Seria muito mais útil podermos generalizar o processo e definir simplesmente a função potência que, a partir de dois números (a *base* e o *expoente*), calcula o primeiro elevado ao segundo.

No entanto, aquilo que fizemos para a quarta-potencia, o cubo e o quadrado dão-nos uma pista muito importante: *se tivermos uma função que calcula a potência de expoente imediatamente inferior, então basta-nos uma multiplicação adicional para calcular a potência seguinte.*

Dito de outra forma, temos:

```
(defun potencia (x n)
  (* (potencia-inferior x n) x))
```

Embora tenhamos conseguido simplificar o problema do cálculo de potência, sobrou uma questão por responder: como podemos calcular a potência imediatamente inferior? A resposta poderá não ser óbvia mas, uma vez percebida, é trivial: *a potência imediatamente inferior à potência de expoente n é a potência de expoente (n-1)*. Isto implica que $(\text{potencia-inferior } x \ n)$ é exactamente o mesmo que $(\text{potencia } x \ (- \ n \ 1))$. Com base nesta ideia, podemos reescrever a definição anterior:

```
(defun potencia (x n)
  (* (potencia x (- n 1)) x))
```

Apesar da nossa solução engenhosa, esta definição tem um problema: qualquer que seja a potência que tentemos calcular, nunca conseguiremos obter o resultado final. Para percebermos este problema, é mais simples usar um caso real: tentemos calcular a terceira potência do número 4, i.e., $(\text{potencia } 4 \ 3)$.

Para isso, de acordo com a definição da função `potencia`, será preciso avaliar a expressão

```
(* (potencia 4 2) 4)  
que, por sua vez, implica avaliar  
(* (* (potencia 4 1) 4) 4)  
que, por sua vez, implica avaliar  
(* (* (* (potencia 4 0) 4) 4) 4)  
que, por sua vez, implica avaliar  
(* (* (* (* (potencia 4 -1) 4) 4) 4) 4)  
que, por sua vez, implica avaliar  
(* (* (* (* (* (potencia 4 -2) 4) 4) 4) 4) 4)  
que, por sua vez, implica avaliar  
(* (* (* (* (* (potencia 4 -3) 4) 4) 4) 4) 4)  
que, por sua vez, implica avaliar ...
```

É fácil vermos que este processo nunca termina. O problema está no facto de termos reduzido o cálculo da potência de um número elevado a um expoente ao cálculo da potência desse número elevado ao expoente imediatamente inferior, mas não dissemos em que situação é que já temos um expoente suficientemente simples cuja solução seja imediata. Quais são as situações em que isso acontece? Já vimos que quando o expoente é 2, a função quadrado devolve o resultado correcto, pelo que o caso $n = 2$ é já suficientemente simples. No entanto, é possível ter um caso ainda mais simples: quando o expoente é 1, o resultado é simplesmente a base. Finalmente, o caso mais simples de todos: quando o expoente é zero, o resultado é 1, independentemente da base. Este último caso é fácil de perceber quando vemos que a avaliação de $(\text{potencia } 4 \ 2)$ (i.e., do quadrado

de quatro) se reduz, em última análise, a $(\star (\star (\text{potencia} \ 4 \ 0) \ 4) \ 4)$. Para que esta expressão seja equivalente a $(\star 4 \ 4)$ é necessário que a avaliação de $(\text{potencia} \ 4 \ 0)$ produza 1.

Estamos então em condições de definir correctamente a função `potencia`:

1. Quando o expoente é zero, o resultado é um.
2. Caso contrário, calculamos a potência de expoente imediatamente inferior e multiplicamo-la pela base.

```
(defun potencia (x n)
  (if (zerop n)
      1
      (* (potencia x (- n 1)) x)))
```

A função anterior é um exemplo de uma função *recursiva*, i.e., uma função que está definida em termos de si própria. Dito de outra forma, uma função recursiva é uma função que se usa a si própria na sua definição. Esse uso é óbvio quando “desenrolamos” a avaliação de $(\text{potencia} \ 4 \ 3)$:

$$\begin{array}{c}
 (\text{potencia} \ 4 \ 3) \\
 \downarrow \\
 (\star (\text{potencia} \ 4 \ 2) \ 4) \\
 \downarrow \\
 (\star (\star (\text{potencia} \ 4 \ 1) \ 4) \ 4) \\
 \downarrow \\
 (\star (\star (\star (\text{potencia} \ 4 \ 0) \ 4) \ 4) \ 4) \\
 \downarrow \\
 (\star (\star (\star 1 \ 4) \ 4) \ 4) \\
 \downarrow \\
 (\star (\star 4 \ 4) \ 4) \\
 \downarrow \\
 (\star 16 \ 4) \\
 \downarrow \\
 64
 \end{array}$$

A *recursão* é o mecanismo que permite que uma função se possa invocar a si própria durante a sua própria avaliação. A recursão é uma das mais importantes ferramentas de programação, pelo que é fundamental que a percebamos bem. Muitos problemas aparentemente complexos possuem soluções recursivas surpreendentemente simples.

Existem inúmeros exemplos de funções recursivas. Uma das mais simples é a função factorial que se define matematicamente como:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n - 1)!, & \text{caso contrário.} \end{cases}$$

A tradução desta fórmula para Lisp é directa:

```
(defun factorial (n)
  (if (zerop n)
      1
      (* n (factorial (- n 1)))))
```

É importante repararmos que em todas as funções recursivas existe:

- Um caso básico (também chamado *caso de paragem*) cujo resultado é imediatamente conhecido.
- Um caso não básico (também chamado *caso recursivo*) em que se transforma o problema original num sub-problema mais simples.

Se analisarmos a função factorial, o caso básico é o teste de igualdade a zero (`(zerop n)`), o resultado imediato é 1, e o caso recursivo é, obviamente, `(* n (factorial (- n 1)))`.

Geralmente, uma função recursiva só está correcta se tiver uma expressão condicional que identifique o caso básico, mas não é obrigatório que assim seja. A invocação de uma função recursiva consiste em ir resolvendo subproblemas sucessivamente mais simples até se atingir o caso mais simples de todos, cujo resultado é imediato. Desta forma, o padrão mais comum para escrever uma função recursiva é:

- Começar por testar os casos básicos.
- Fazer uma invocação recursiva com um subproblema cada vez mais próximo de um caso básico.
- Usar o resultado da invocação recursiva para produzir o resultado da invocação original.

Dado este padrão, os erros mais comuns associados às funções recursivas são, naturalmente:

- Não detectar um caso básico.
- A recursão não diminuir a complexidade do problema, i.e., não passar para um problema mais simples.
- Não usar correctamente o resultado da recursão para produzir o resultado originalmente pretendido.

Repare-se que uma função recursiva que funciona perfeitamente para os casos para que foi prevista pode estar completamente errada para outros casos. A função `factorial` é um exemplo: quando o argumento é negativo, o problema torna-se cada vez mais complexo, cada vez mais longe do caso simples:

```

(factorial -1)
  ↓
(* -1 (factorial -2))
  ↓
(* -1 (* -2 (factorial -3)))
  ↓
(* -1 (* -2 (* -3 (factorial -4))))
  ↓
(* -1 (* -2 (* -3 (* -4 (factorial -5)))))
  ↓
(* -1 (* -2 (* -3 (* -4 (* -5 (factorial -6)))))
  ↓
...

```

O caso mais frequente de erro numa função recursiva é a recursão nunca parar, ou porque não se detecta correctamente o caso básico, ou por a recursão não diminuir a complexidade do problema. Neste caso, o número de invocações recursivas cresce indefinidamente até esgotar a memória do computador, altura em que o programa gera um erro. No caso do Auto Lisp, esse erro não é inteiramente óbvio pois o avaliador apenas interrompe a avaliação sem apresentar qualquer resultado. Eis um exemplo:

```

_ $ (factorial 3)
6
_ $ (factorial -1)
_ $

```

É muito importante compreendermos bem o conceito de recursão. Embora a princípio possa ser difícil abranger por completo as implicações deste conceito, a recursão permite resolver, com enorme simplicidade, problemas aparentemente muito complexos.

Exercicio 4.2 A função de Ackermann é definida para números não negativos da seguinte forma:

$$A(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ A(m - 1, 1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

Defina, em Lisp, a função de Ackermann.

Exercicio 4.3 Indique o valor de

1. (ackermann 0 8)
2. (ackermann 1 8)
3. (ackermann 2 8)
4. (ackermann 3 8)
5. (ackermann 4 8)

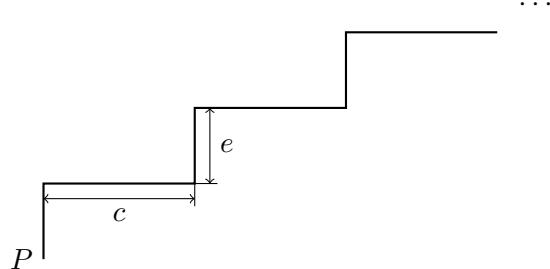


Figura 29: Perfil de uma escada com n degraus cujo primeiro degrau começa no ponto P e cujos degraus possuem um cobertor c e um espelho e .

4.6 Recursão em Arquitectura

Como iremos ver, também em arquitectura a recursão é um conceito fundamental. A título de exemplo, consideremos o perfil de uma escada, tal como esquematizado na Figura 29 e supunhamos que pretendemos definir uma função denominada *escada* que, dado o ponto P , dados o comprimento do cobertor c e o comprimento do espelho e de cada degrau e, finalmente, dado o número n de degraus, desenha a escada em AutoCad com o primeiro espelho a começar a partir do ponto P . Dados estes parâmetros, a definição da função deverá começar por:

```
(defun escada (p c e n)
  ...)
```

Para implementarmos esta função temos de ser capazes de decompor o problema em subproblemas menos complexos e é aqui que a recursão dá uma enorme ajuda: ela permite-nos decompor o desenho de uma escada com n degraus no desenho de um degrau seguido do desenho de uma escada com $n - 1$ degraus, tal como se apresenta no esquema da Figura 30.

Isto quer dizer que a função ficará algo da forma:

```
(defun escada (p c e n)
  ...
  (degrau p c e)
  (escada (+xy p c e) c e (- n 1)))
```

Para desenharmos um degrau, podemos definir a seguinte função que cria os segmentos do espelho e do cobertor:

```
(defun degrau (p c e)
  (command "_pline" p (+y p e) (+xy p c e) ""))
```

O problema que se coloca agora é que a função *escada* precisa de parar de desenhar degraus num determinado momento. É fácil vermos que

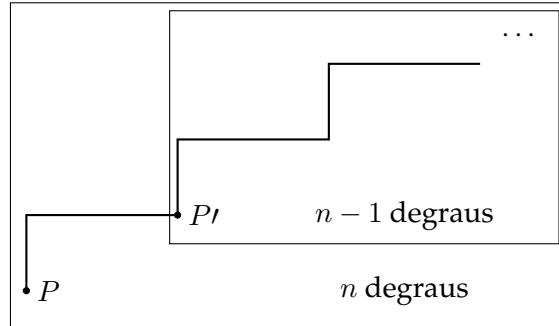


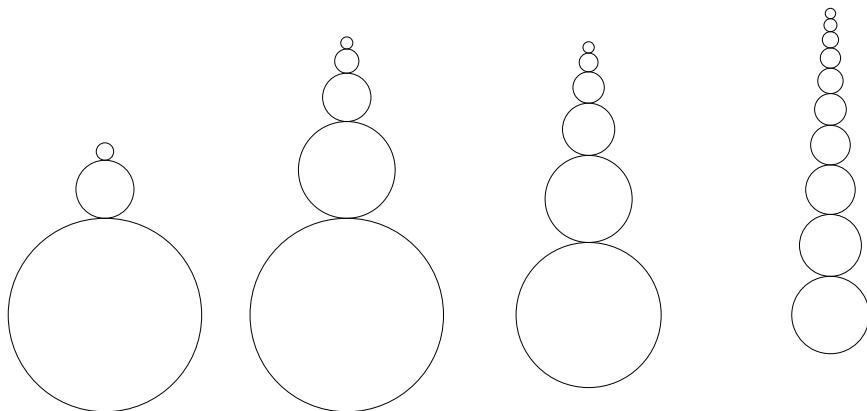
Figura 30: Decomposição do desenho de uma escada de n degraus no desenho de um degrau seguido do desenho de uma escada de $n - 1$ degraus.

esse momento chega quando, ao reduzirmos sucessivamente o números de degraus, atingimos um ponto em que esse número é zero. Assim, quando pedimos para desenhar uma escada com zero degraus, a função escada já não precisa de fazer nada. Isso quer dizer que a função tem de ter a seguinte forma:

```
(defun escada (p c e n)
  (if (= n 0)
    ...
    ...
    (degrau p c e)
    (escada (+xy p c e) c e (- n 1)))

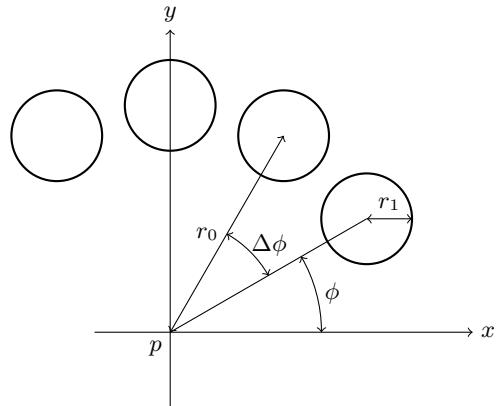
(defun escada (p0 degraus espelho cobertor / p1 p2)
  (if (= degraus 0)
    nil
    (progn
      (setq p1 (+y p0 espelho)
            p2 (+x p1 cobertor))
      (command "_pline" p0 p1 p2 "")
      (escada p2 (1- degraus) espelho cobertor))))
```

Exercicio 4.4 Defina uma função equilibrio-circulos capaz de criar qualquer uma das figuras apresentadas em seguida:



Note que os círculos possuem raios que estão em progressão geométrica de razão f , com $0 < f < 1$. Assim, cada círculo (excepto o primeiro) tem um raio que é o produto de f pelo raio do círculo maior em que está apoiado. O círculo mais pequeno de todos tem raio maior ou igual a 1. A sua função deverá ter como parâmetros o centro e o raio do círculo maior e, ainda, o factor de redução f .

Exercicio 4.5 Considere o desenho de círculos tal como apresentado na seguinte imagem:

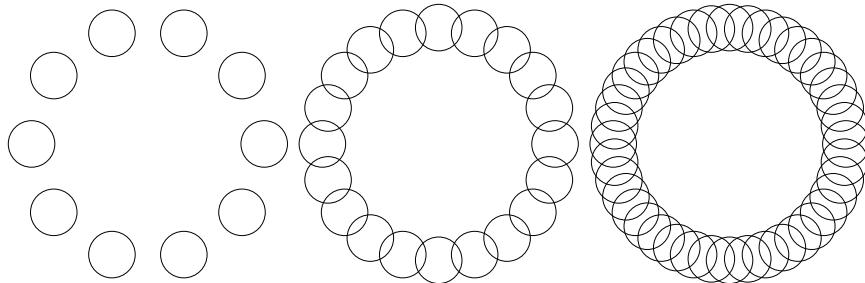


Escreva uma função denominada `circulos-radiais` que, a partir das coordenadas p do centro de rotação, do número de círculos n , do raio de translacção r_0 , do raio de circunferência r_1 , do ângulo inicial ϕ e do incremento de ângulo $\Delta\phi$, desenha os círculos tal como apresentados na figura anterior.

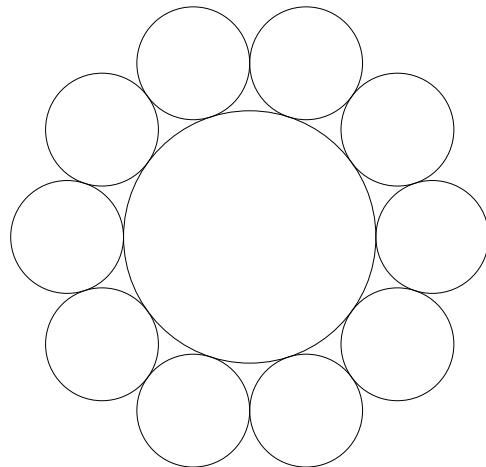
Teste a sua função com os seguintes expressões:

```
(circulos-radiais (xy 0 0) 10 10 2 0 (/ pi 5))
(circulos-radiais (xy 25 0) 20 10 2 0 (/ pi 10))
(circulos-radiais (xy 50 0) 40 10 2 0 (/ pi 20))
```

cuja avaliação deverá gerar a imagem seguinte:



Exercicio 4.6 Considere o desenho de flores simbólicas compostas por um círculo interior em torno da qual estão dispostos círculos radiais correspondentes a pétalas. Estes círculos deverão ser tangentes uns aos outros e ao círculo interior, tal como se apresenta na seguinte imagem:

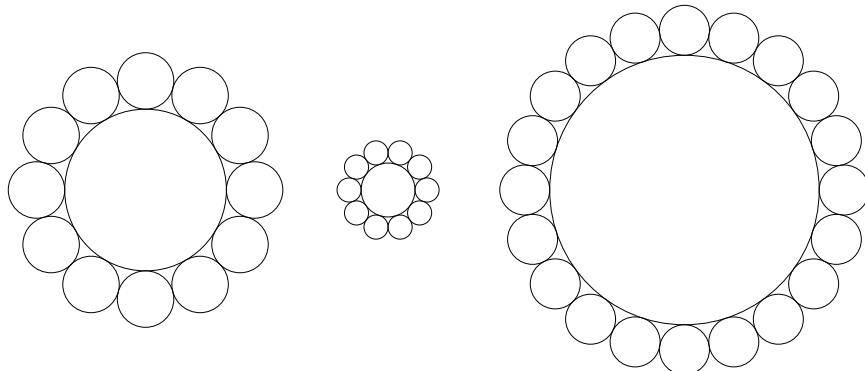


Defina a função `flor` que recebe apenas o ponto correspondente ao centro da flor, o raio do círculo interior e o número de pétalas.

Teste a sua função com as expressões

```
(flor (xy 0 0) 5 10)
(flor (xy 18 0) 2 10)
(flor (xy 40 0) 10 20)
```

que deverão gerar a imagem seguinte:



4.7 Depuração de Programas Recursivos

Vimos, na secção 3.17, que os erros de um programa se podem classificar em termos de erros sintáticos ou erros semânticos. Os erros sintáticos ocorrem quando escrevemos frases inválidas na linguagem. Como exemplo de erro sintático, consideremos a seguinte definição da função potencia onde nos esquecemos da lista de parâmetros:

```
(defun potencia
  (if (= n 0)
      1
      (* (potencia x (- n 1)) x)))
```

Este esquecimento é o suficiente para baralhar o Auto Lisp: ele esperava encontrar a lista de parâmetros imediatamente a seguir a nome da função e que, no lugar dela, encontra uma lista que começa com o símbolo `if`. Isto faz com que o Auto Lisp erradamente acredite que o símbolo `if` é o nome do primeiro parâmetro e, daí para a frente, a sua confusão só aumenta. A dado momento, quando o Auto Lisp deixa, por completo, de compreender aquilo que se pretendia definir, apresenta-nos um erro.

Os erros semânticos são mais complexos que os sintáticos, pois, em geral, só podem ser detectados durante a execução do programa.³⁴ Por exemplo, se tentarmos calcular o factorial de uma *string*, iremos ter um erro, tal como o seguinte exemplo mostra:

```
_\$ (factorial 5)
120
_\$ (factorial "cinco")
; error: bad argument type: numberp: "cinco"
```

Este último erro, como é óbvio, não tem a ver com as regras da gramática do Lisp: a “frase” da invocação da função `factorial` está correcta. O problema está no facto de não fazer sentido calcular o factorial de uma *string*, pois o cálculo do factorial envolve operações aritméticas e estas não são aplicáveis a *strings*. Assim sendo, o erro tem a ver com o significado da “frase” escrita, i.e., com a semântica. Trata-se, portanto, de um erro semântico.

A recursão infinita é outro exemplo de um erro semântico. Vimos que a função `factorial` só pode ser invocada com um argumento não negativo ou provoca recursão infinita. Consequentemente, se usarmos um argumento negativo, estaremos a cometer um erro semântico.

³⁴Há casos de erros semânticos que podem ser detectados *antes* da execução do programa, mas essa detecção depende muito da qualidade da implementação da linguagem e da sua capacidade em antecipar as consequências dos programas.

4.7.1 Trace

Vimos, na secção 3.17, que o Visual Lisp disponibiliza vários mecanismos para fazer a detecção de erros. Um dos mais simples é a forma `trace` que permite visualizar as invocações das funções. A forma `trace` recebe o nome das funções cuja invocação se pretende analisar e altera essas funções de forma a que elas escrevam as sucessivas invocações com os respectivos argumentos, bem como o resultado da invocação. Se o ambiente do Visual Lisp estiver activo, a escrita é feita numa janela especial do ambiente do Visual Lisp, denominada `Trace`,³⁵ caso contrário, é feita na janela de comandos do AutoCad. A informação apresentada em resultado do `trace` é, em geral, extremamente útil para a depuração das funções.

Por exemplo, para visualizarmos uma invocação da função `factorial`, consideremos a seguinte interacção:

```
_\$ (trace factorial)
FACTORIAL
_\$ (factorial 5)
120
```

Em seguida, se consultarmos o conteúdo da janela de `Trace`, encontraremos:

```
Entering (FACTORIZATION 5)
Entering (FACTORIZATION 4)
Entering (FACTORIZATION 3)
Entering (FACTORIZATION 2)
Entering (FACTORIZATION 1)
Entering (FACTORIZATION 0)
Result: 1
Result: 1
Result: 2
Result: 6
Result: 24
Result: 120
```

Note-se, no texto anterior escrito em consequência do `trace`, que a invocação que fizemos da função `factorial` aparece encostada à esquerda. Cada invocação recursiva aparece ligeiramente para a direita, permitindo assim visualizar a “profundidade” da recursão, i.e., o número de invocações recursivas. O resultado devolvido por cada invocação aparece alinhado na mesma coluna dessa invocação.

É de salientar que a janela de `Trace` não tem dimensão infinita, pelo que as recursões excessivamente profundas acabarão por não caber na janela. Neste caso, o Visual Lisp reposiciona a escrita na coluna esquerda mas indica numericamente o nível de profundidade em que se encontra. Por exemplo, para o `(factorial 15)`, aparece:

³⁵Para se visualizar a janela de `Trace` basta seleccioná-la a partir do menu `Window` do Auto Lisp.

```

Entering (FACTORIAL 15)
Entering (FACTORIAL 14)
Entering (FACTORIAL 13)
Entering (FACTORIAL 12)
Entering (FACTORIAL 11)
Entering (FACTORIAL 10)
Entering (FACTORIAL 9)
Entering (FACTORIAL 8)
Entering (FACTORIAL 7)
Entering (FACTORIAL 6)
[10] Entering (FACTORIAL 5)
[11]   Entering (FACTORIAL 4)
[12]     Entering (FACTORIAL 3)
[13]       Entering (FACTORIAL 2)
[14]         Entering (FACTORIAL 1)
[15]           Entering (FACTORIAL 0)
[15]             Result: 1
[14]               Result: 1
[13]                 Result: 2
[12]                   Result: 6
[11]                     Result: 24
[10]                       Result: 120
                           Result: 720
                           Result: 5040
                           Result: 40320
                           Result: 362880
                           Result: 3628800
                           Result: 39916800
                           Result: 479001600
                           Result: 1932053504
                           Result: 1278945280
                           Result: 2004310016

```

No caso de recursões infinitas, apenas são mostradas as últimas invocações realizadas antes de ser gerado o erro. Por exemplo, para (factorial -1), teremos:

```

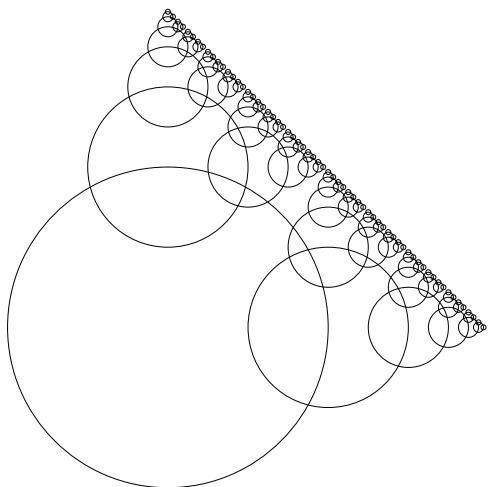
...
[19215]      Entering (FACTORIAL -19216)
[19216]          Entering (FACTORIAL -19217)
[19217]              Entering (FACTORIAL -19218)
[19218]                  Entering (FACTORIAL -19219)
[19219]                      Entering (FACTORIAL -19220)
[19220]  Entering (FACTORIAL -19221)
[19221]      Entering (FACTORIAL -19222)
[19222]          Entering (FACTORIAL -19223)
[19223]              Entering (FACTORIAL -19224)
[19224]                  Entering (FACTORIAL -19225)
...
[19963]      Entering (FACTORIAL -19964)
[19964]          Entering (FACTORIAL -19965)
[19965]              Entering (FACTORIAL -19966)
[19966]                  Entering (FACTORIAL -19967)
[19967]                      Entering (FACTORIAL -19968)
[19968]                          Entering (FACTORIAL -19969)
[19969]                              Entering (FACTORIAL -19970)
[19970]  Entering (FACTORIAL -19971)
[19971]      Entering (FACTORIAL -19972)
[19972]          Entering (FACTORIAL -19973)
[19973]              Entering (FACTORIAL -19974)
[19974]                  Entering (FACTORIAL -19975)
[19975]                      Entering (FACTORIAL -19976)

```

Para se parar a depuração de uma função, usa-se a forma especial `untrace`, que recebe o nome da função ou funções que se pretende retirar do `trace`.

Exercicio 4.7 Faça o `trace` da função potência. Qual é o `trace` resultante da avaliação (`potencia 2 10`)?

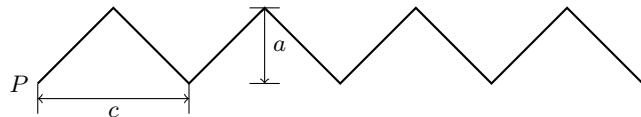
Exercicio 4.8 Defina uma função `circulos` capaz de criar a figura apresentada em seguida:



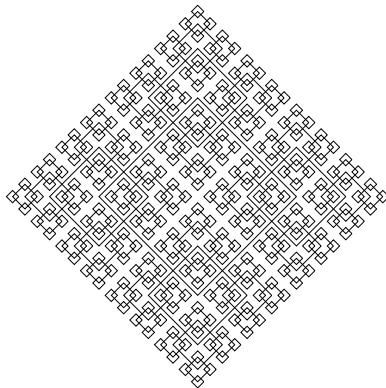
Note que os círculos possuem raios que estão em progressão geométrica de razão $\frac{1}{2}$. Dito de outra forma, os círculos mais pequenos têm metade do raio do círculo maior que

lhes é adjacente. Os círculos mais pequenos de todos têm raio maior ou igual a 1. A sua função deverá ter como parâmetros apenas o centro e o raio do círculo maior.

Exercicio 4.9 Defina uma função denominada `serra` que, dado um ponto P , um número de dentes, o comprimento c de cada dente e a altura a de cada dente, desenha uma serra em AutoCad com o primeiro dente a começar a partir do ponto P , tal como se vê na imagem seguinte:

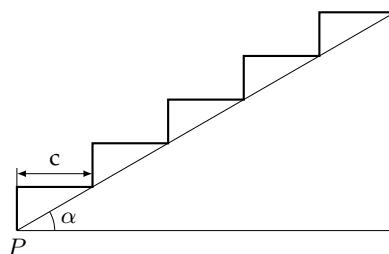


Exercicio 4.10 Defina uma função `losangulos` capaz de criar a figura apresentada em seguida:



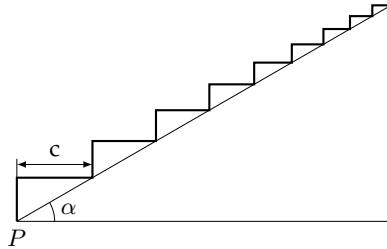
Note que os losangos possuem dimensões que estão em progressão geométrica de razão $\frac{1}{2}$. Dito de outra forma, os losangos mais pequenos têm metade do tamanho do losango maior em cujas extremidades estão centrados. Os losangos mais pequenos de todos têm largura maior ou igual a 1. A sua função deverá ter como parâmetros apenas o centro e a largura do losango maior.

Exercicio 4.11 Considere a escada esquematizada na seguinte figura e destinada a vencer uma rampa de inclinação α .



Defina a função `escada-rampa` que recebe o ponto P , o ângulo α , o cobertor c e o número de degraus n e constrói a escada descrita no esquema anterior.

Exercicio 4.12 Considere a escada esquematizada na seguinte figura e destinada a vencer uma rampa de inclinação α .



Note que os degraus da escada possuem dimensões que estão em progressão geométrica de razão f , i.e., dado um degrau cujo cobertor é de dimensão c , o degrau imediatamente acima tem um cobertor de dimensão $f \cdot c$. Defina a função `escada-progressao-geometrica` que recebe o ponto P , o ângulo α , o cobertor c , o número de degraus n e o factor f e constrói a escada descrita no esquema anterior.

4.8 Templos Dóricos

Vimos, pelas descrições de Vitrúvio, que os Gregos criaram um elaborado sistema de proporções para colunas. Estas colunas eram usadas para a criação de *pórticos*, em que uma sucessão de colunas encimadas por um telhado servia de entrada para os edifícios e, em particular, para os templos. Quando esse arranjo de colunas era avançado em relação ao edifício, denominava-se o mesmo de *próstilo*, classificando-se este pelo número de colunas que possuem em Distilo, Tristilo, Tetrastilo, Pentastilo, Hexastilo, etc. Quando o *próstilo* se alargava a todo o edifício, colocando colunas a toda a sua volta, denominava-se de *peristilo*.

Para além de descrever as proporções das colunas, Vitrúvio também explicou no seu famoso tratado as regras a que devia obedecer a construção dos templos, em particular, no que diz respeito à sua orientação, que devia ser de este para oeste, e no que diz respeito à separação entre colunas, distinguindo vários casos de templos, desde aqueles em que o espaço entre colunas era muito reduzido (*picnostilo*) até aos templos com espaçamento excessivamente alargado (*araeostilo*), passando pelo seu favorito (*eustilo*) em que o espaço entre colunas é variável, sendo maior nas colunas centrais.

Para simplificar a nossa implementação, vamos ignorar estes detalhes e, ao invés de distinguir vários *stilo*, vamos simplesmente considerar o posicionamento de colunas distribuídas linearmente segundo uma determinada orientação, tal como esquematizamos na Figura 31.

Até este momento, temos considerado as coordenadas meramente como posições no espaço. Agora, para a modelação deste templo, será útil adoptarmos uma perspectiva *vectorial* do conceito de coordenadas. Nesta perspectiva, as coordenadas passam a ser vistas como a extremidade de vectores posicionados na origem. Isto é visível na Figura 31 onde assinalámos as posições de duas colunas do templo através dos vectores P e Q .

Estando todos os vectores posicionados na origem torna-se irrelevante falar dela, o que nos permite caracterizar os vectores como tendo apenas

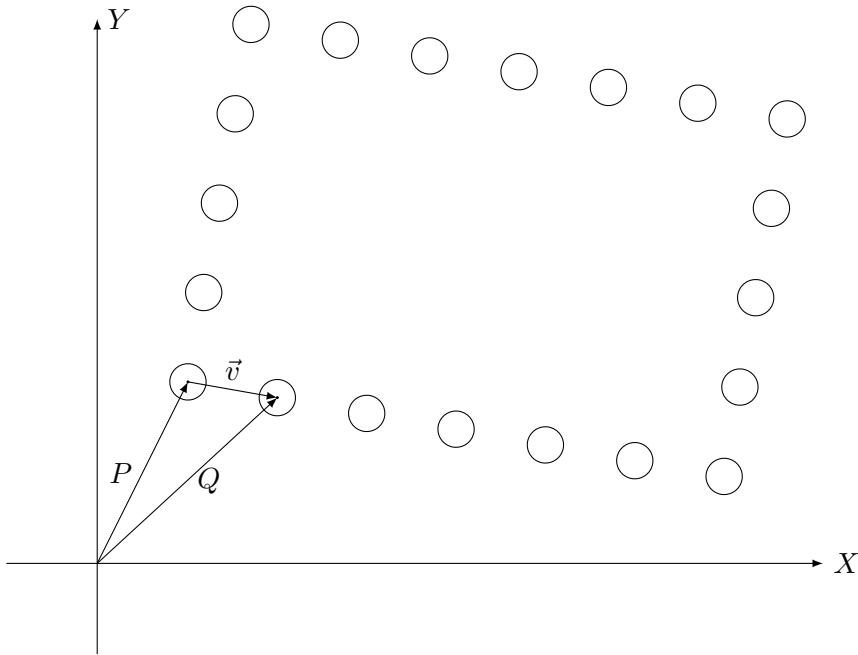


Figura 31: Planta de um templo com uma orientação arbitrária.

uma magnitude e uma direcção. É fácil verificarmos que esta magnitude e direcção são, precisamente, as coordenadas polares da extremidade do vector, i.e., a distância da origem à extremidade do vector e o ângulo que o vector faz com o eixo X .

A grande vantagem da perspectiva vectorial é que ela permite-nos conceber uma *álgebra* para operar com vectores. Por exemplo, a soma de vectores é o vector cujos componentes são a soma dos componentes correspondentes, i.e.,

$$P + Q = (P_x + Q_x, P_y + Q_y, P_z + Q_z)$$

Do mesmo modo, podemos definir a subtracção de vectores através de

$$P - Q = (P_x - Q_x, P_y - Q_y, P_z - Q_z)$$

e a multiplicação de um vector por um escalar α :

$$P \cdot \alpha = (P_x \cdot \alpha, P_y \cdot \alpha, P_z \cdot \alpha)$$

Finalmente, a divisão de um vector por um escalar pode ser definida à custa da multiplicação pelo inverso do escalar, i.e.,

$$\frac{P}{\alpha} = P \frac{1}{\alpha}$$

Estas operações estão esquematizadas graficamente na Figura 32 para o caso bidimensional.

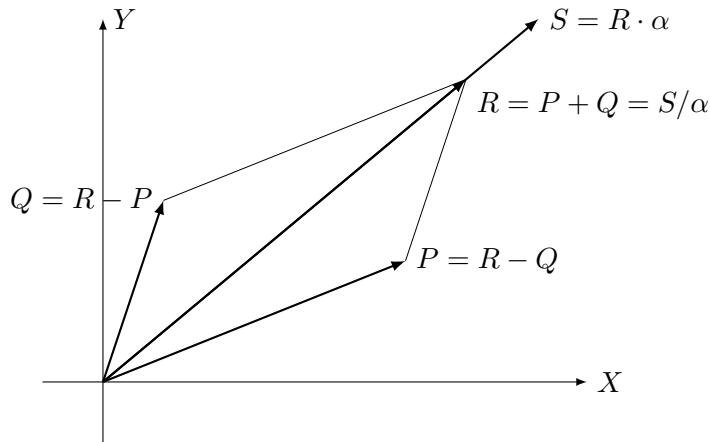


Figura 32: Operações algébricas com vectores.

A tradução destas definições para Auto Lisp é directa. Uma vez que estamos criar operações algébricas para coordenadas, vamos baptizá-las combinando os nomes dos operadores aritméticos com a letra “c” (de “coordenadas”).

```
(defun +c (p0 p1)
  (xyz (+ (cx p0) (cx p1))
        (+ (cy p0) (cy p1))
        (+ (cz p0) (cz p1)))))

(defun -c (p0 p1)
  (xyz (- (cx p0) (cx p1))
        (- (cy p0) (cy p1))
        (- (cz p0) (cz p1))))

(defun *c (p a)
  (xyz (* (cx p) a)
        (* (cy p) a)
        (* (cz p) a)))

(defun /c (p a)
  (*c p (/ 1.0 a)))
```

Exercicio 4.13 Um vector *unitário* é um vector de magnitude unitária. Defina a operação *vector-unitario* que, dado um vector qualquer, calcula o vector unitário que tem a mesma direcção que o vector dado.

Exercicio 4.14 O vector *simétrico* de um vector \vec{v} é o vector \vec{v}' tal que $\vec{v} + \vec{v}' = 0$. Dito de outro modo, é o vector de igual magnitude mas direcção oposta. Defina a operação *vector-simetrico* que, dado um vector qualquer, calcula o vector simétrico do vector dado.

Voltando agora ao problema do posicionamento das colunas no templo, ilustrado na Figura 31, dado o vector \vec{v} de orientação e separação de colunas, a partir da posição de qualquer coluna P , determinamos a posição da

coluna seguinte através de $P + \vec{v}$. Este raciocínio permite-nos definir uma primeira função capaz de criar uma fileira de colunas. Esta função irá usar, como parâmetros, as coordenadas P da base do eixo da primeira coluna, a altura h da coluna, o vector \vec{v} de separação entre os eixos das colunas e, finalmente, o número n de colunas que pretendemos colocar. O raciocínio para a definição desta função é, mais uma vez, recursivo:

- Se o número de colunas a colocar for zero, então não é preciso fazer nada e podemos simplesmente retornar `nil`.
- Caso contrário, colocamos uma coluna no ponto P e, recursivamente, colocamos as restantes colunas a partir do ponto que resulta de somarmos o vector de separação v ao ponto P . Como são duas acções que queremos realizar sequencialmente, teremos de usar o operador `progn` para as agrupar numa acção conjunta.

Traduzindo este raciocínio para Lisp, temos:

```
(defun colunas-doricas (p h v n)
  (if (= n 0)
      nil
      (progn
        (coluna-dorica p h)
        (colunas-doricas (+c p v)
                         h
                         v
                         (- n 1)))))
```

Podemos testar a criação das colunas usando, por exemplo:

```
(colunas-doricas (xy 0 0) 10 (xy 5 0) 8)
```

cujo resultado está apresentado na Figura 33:

Exercício 4.15 Embora a utilização do vector de separação entre colunas seja relativamente simples, é possível simplificar ainda mais através do cálculo desse vector a partir dos pontos inicial e final da fileira de colunas. Usando a função `colunas-doricas`, defina uma função denominada `colunas-doricas-entre` que, dados os centros P e Q da base das colunas inicial e final, a altura h das colunas e, finalmente, o número de colunas, cria a fileira de colunas entre aqueles dois centros.

A título de exemplo, a imagem seguinte mostra o resultado da avaliação das seguintes expressões:

```
(colunas-doricas-entre (pol 10 0.0) (pol 50 0.0) 8 6)
(colunas-doricas-entre (pol 10 0.4) (pol 50 0.4) 8 6)
(colunas-doricas-entre (pol 10 0.8) (pol 50 0.8) 8 6)
(colunas-doricas-entre (pol 10 1.2) (pol 50 1.2) 8 6)
(colunas-doricas-entre (pol 10 1.6) (pol 50 1.6) 8 6)
```

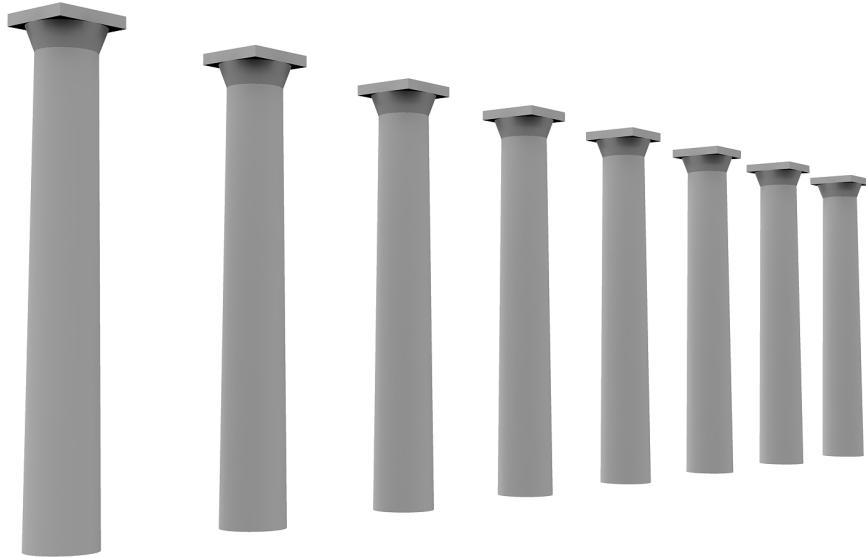
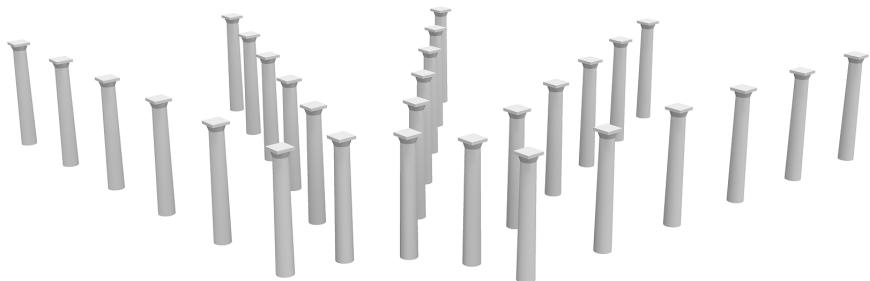


Figura 33: Uma perspectiva de um conjunto de oito colunas dóricas com 10 unidades de altura e 5 unidades de separação entre os eixos da colunas ao longo do eixo x .



A partir do momento em que sabemos construir fileiras de colunas, torna-se relativamente fácil a construção das quatro fileiras necessárias para os templos em peristilo. Normalmente, a descrição destes templos faz-se em termos do número de colunas da frente e do número de colunas do lado, mas assumindo que as colunas dos cantos contam para ambas as medidas. Isto quer dizer que num templo de, por exemplo, 6×12 colunas existem, na realidade, apenas $4 \times 2 + 10 \times 2 + 4 = 32$ colunas. Para a construção do peristilo, para além do número de colunas das frontes e lados, será necessário conhecer a posição das colunas extremas do templo e, claro, a altura das colunas.

Em termos de algoritmo, vamos começar por construir um dos cantos do peristilo, i.e., uma frente e um lado:

```
(defun canto-peristilo-dorico (p altura v0 n0 v1 n1)
  (colunas-doricas p altura v0 n0)
  (colunas-doricas (+c p v1) altura v1 (- n1 1)))
```

Note-se que, para evitar repetir colunas, a segunda fileira começa na segunda coluna e, logicamente, coloca menos uma coluna.

Para construirmos o peristilo completo, basta-nos construir um canto e, de seguida, construir o outro canto mas com menos uma coluna em cada lado e progredindo nas direcções opostas.

```
(defun peristilo-dorico (p altura v0 n0 v1 n1)
  (canto-peristilo-dorico p altura v0 n0 v1 n1)
  (canto-peristilo-dorico
    (+c p (+c (*c v0 (- n0 1)) (*c v1 (- n1 1))))
    altura
    (*c v0 -1) (- n0 1) (*c v1 -1) (- n1 1)))
```

Para um exemplo realista podemos considerar o templo de Segesta que se encontra representado da Figura 11. Este templo é do tipo peristilo, composto por 6 colunas (i.e., Hexastilo) em cada frente e 14 colunas nos lados, num total de 36 colunas de 9 metros de altura. A distância entre os eixos das colunas é de aproximadamente 4.8 metros nas frontes e de 4.6 nos lados. A expressão que cria o peristilo deste templo é, então:

```
(peristilo-dorico (xy 0 0) 9 (xy 4.8 0) 6 (xy 0 4.6) 14)
```

O resultado da avaliação da expressão anterior está representado na Figura 34

Embora a grande maioria dos templos Gregos fosse de formato rectangular, também foram construídos templos de formato circular, a que chamaram *Tholos*. O Santuário de Atenas Pronaia, em Delfos, contém um bom exemplo de um destes edifícios. Embora pouco reste deste templo, não é difícil imaginar a sua forma original a partir do que ainda é visível na Figura 35.

Para simplificar a construção do *Tholos*, vamos dividi-lo em duas partes. Numa, iremos desenhar a base e, na outra, iremos posicionar as colunas.

Para o desenho da base, podemos considerar um conjunto de cilindros achatados, sobrepostos de modo a formar degraus circulares, tal como se apresenta na Figura 36. Desta forma, a altura total da base a_b será dividida em passos de Δa_b e o raio da base também será reduzido em passos de Δr_b .

Para cada cilindro teremos de considerar o seu raio e a altura do espelho do degrau d-altura. Para passarmos ao cilindro seguinte temos ainda de ter em conta o aumento do raio d-raio devido ao comprimento do cobertor do degrau. Estes degraus serão construídos segundo um processo recursivo:

- Se o número de degraus a colocar é zero, não é preciso fazer nada.

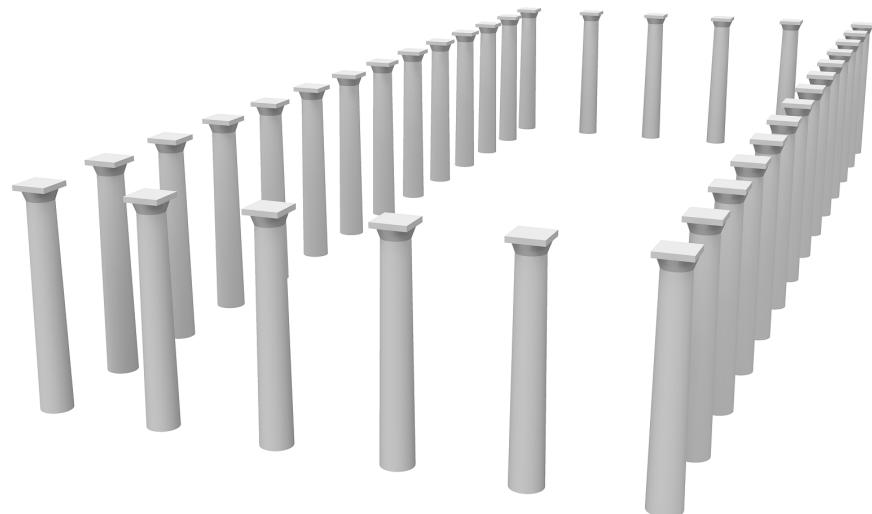


Figura 34: Uma perspectiva do peristilo do templo de Segesta. As colunas foram geradas pela função `peristilo-dorico` usando, como parâmetros, 6 colunas na frente e 14 no lado, com distância intercolunar de 4.8 metros na frente e 4.6 metros no lado, e colunas de 9 metros de altura.



Figura 35: O Templo de Atenas Pronaia em Delfos, construído no século quarto antes de Cristo. Fotografia de Michelle Kelley.

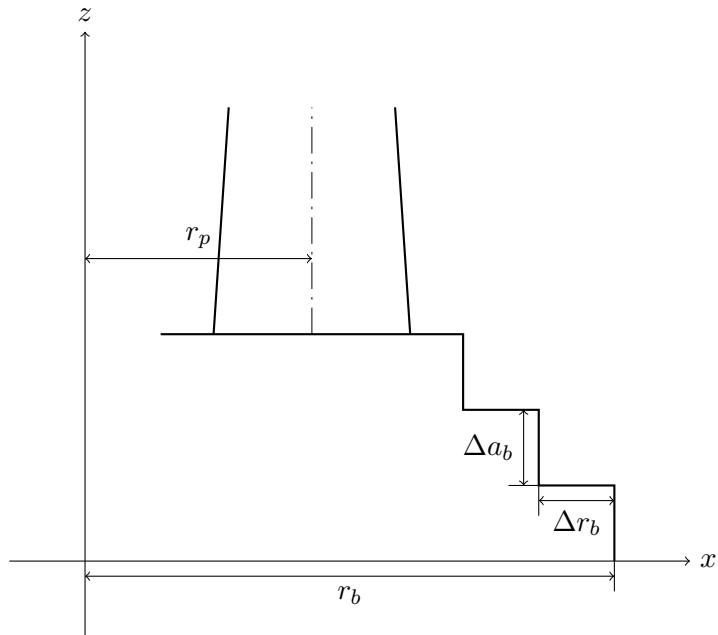


Figura 36: Corte da base de um *Tholos*. A base é composta por uma sequência de cilindros sobrepostos cujo raio de base r_b encolhe de Δr_b a cada degrau e cuja altura incrementa Δa_b a cada degrau.

- Caso contrário, colocamos um degrau (modelado por um cilindro) com o raio e a altura do degrau e, recursivamente, colocamos os restantes degraus em cima deste, i.e., numa cota igual à altura do degrau agora colocado e com um raio reduzido do comprimento do cobertor do degrau agora colocado.

Este processo é implementado pela seguinte função:

```
(defun base-tholos (p n-degraus raio d-altura d-raio)
  (if (= n-degraus 0)
      nil
      (progn
        (command "_cylinder" p raio d-altura)
        (base-tholos (+xyz p 0 0 d-altura)
                     (- n-degraus 1)
                     (- raio d-raio)
                     d-altura
                     d-raio))))
```

Para o posicionamento das colunas, vamos também considerar um processo em que em cada passo apenas posicionamos uma coluna numa dada posição e, recursivamente, colocamos as restantes colunas a partir da posição circular seguinte.

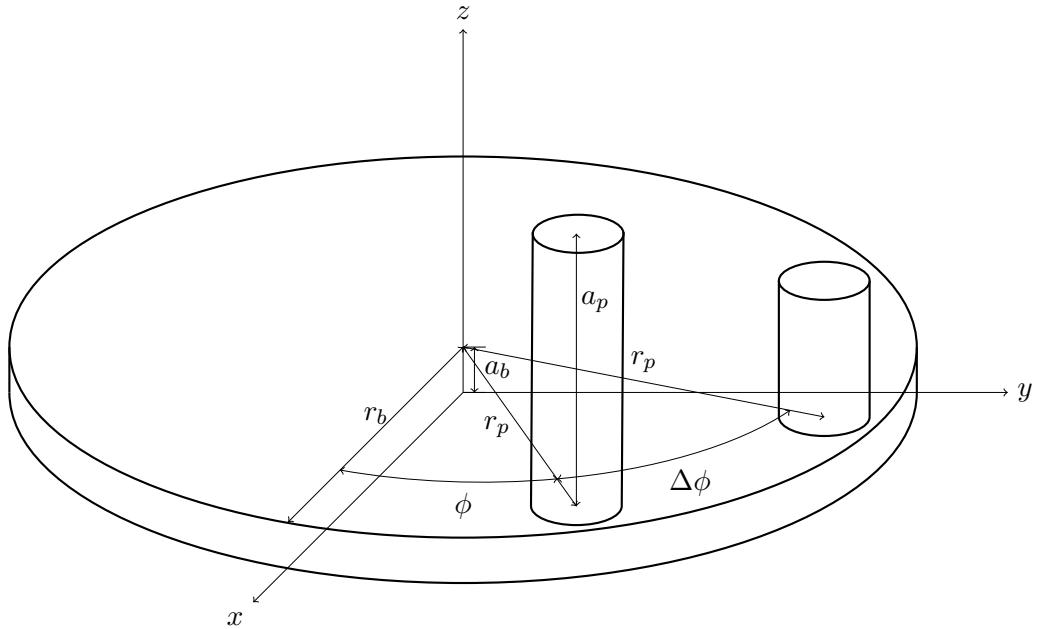


Figura 37: Esquema da construção de um *Tholos*: r_b é o raio da base, r_p é a distância do centro das colunas ao centro da base, a_p é a altura das colunas, a_b é a altura da base, ϕ é o ângulo inicial de posicionamento das colunas e $\Delta\phi$ é o ângulo entre colunas.

Dada a sua estrutura circular, a construção deste género de edifícios é simplificada pelo uso de coordenadas circulares. De facto, podemos conceber um processo recursivo que, a partir do raio r_p do peristilo e do ângulo inicial ϕ , coloca uma coluna nessa posição e que, de seguida, coloca as restantes colunas usando o mesmo raio mas incrementando o ângulo ϕ de $\Delta\phi$, tal como se apresenta na Figura 37. O incremento angular $\Delta\phi$ obtém-se pela divisão da circunferência pelo número n de colunas a colocar, i.e., $\Delta\phi = \frac{2\pi}{n}$. Uma vez que as colunas se dispõem em torno de um círculo, o cálculo das coordenadas de cada coluna fica facilitado pelo uso de coordenadas polares. Tendo este algoritmo em mente, a definição da função fica:

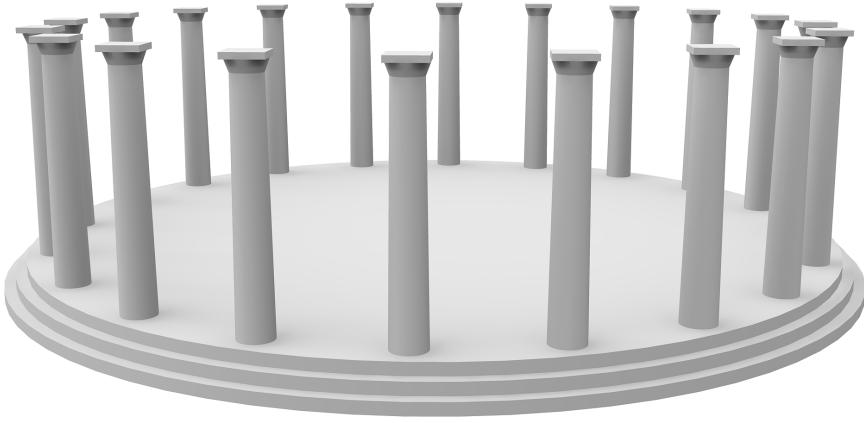


Figura 38: Perspectiva do *Tholos* de Atenas em Delfos, constituído por 20 colunas de estilo Dórico, de 4 metros de altura e colocadas num círculo com 7 metros de raio.

```
(defun colunas-tholos (p n-colunas raio fi d-fi altura)
  (if (= n-colunas 0)
      nil
      (progn
        (coluna-dorica (+pol p raio fi) altura)
        (colunas-tholos p
          (- n-colunas)
          raio
          (+ fi d-fi)
          d-fi
          altura))))
```

Finalmente, definimos a função *tholos* que, dados os parâmetros necessários às duas anteriores, as invoca em sequência:

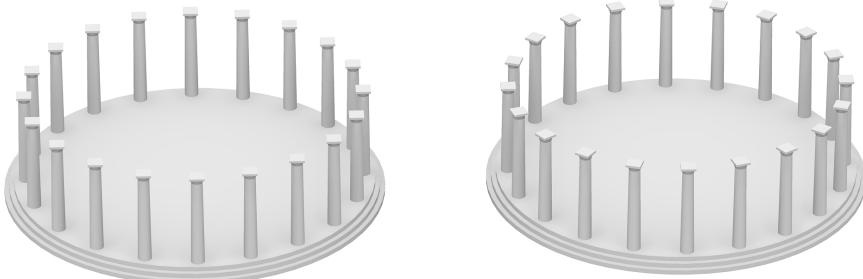
```
(defun tholos (p n-degraus rb dab drb n-colunas rp ap)
  (base-tholos p n-degraus rb dab drb)
  (colunas-tholos (+xyz p 0 0 (* n-degraus dab))
    n-colunas
    rp
    0
    (/ 2*pi n-colunas)
    ap))
```

A Figura 38 mostra a imagem gerada pela avaliação da seguinte expressão:

```
(tholos (xyz 0 0 0) 3 7.9 0.2 0.2 20 7 4)
```

Exercicio 4.16 Uma observação atenta do *Tholos* apresentado na Figura 38 mostra que existe um erro: os ábacos das várias colunas são paralelos uns aos outros (e aos eixos das abcissas e ordenadas) quando, na realidade, deveriam ter uma orientação radial. Essa diferença

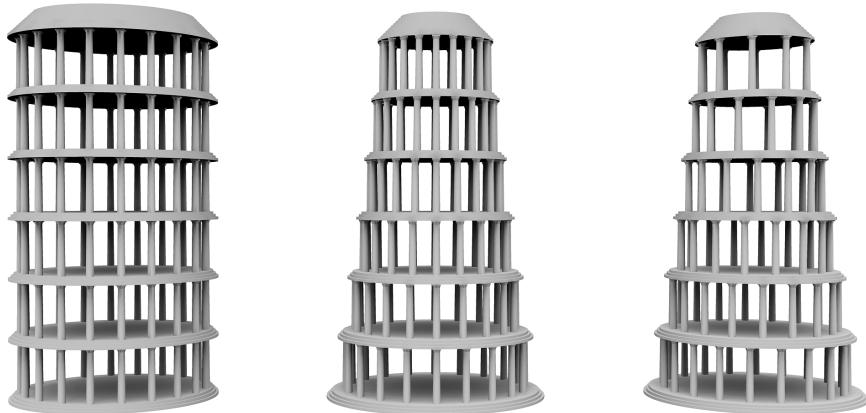
é evidente quando se compara uma vista de topo do desenho actual (à esquerda) com a mesma vista daquele que seria o desenho correcto (à direita):



Defina uma nova função `coluna-dorica-rodada` que, para além da altura da coluna, recebe ainda o ângulo de rotação que a coluna deve ter. Uma vez que o fuste e o coxim da coluna têm simetria axial, esse ângulo de rotação só influencia o ábaco, pelo que deverá também definir uma função para desenhar um ábaco rodado.

De seguida, redefina a função `colunas-tholos` de modo a que cada coluna esteja orientada correctamente relativamente ao centro do *Tholos*.

Exercicio 4.17 Considere a construção de uma torre composta por vários módulos em que cada módulo tem exactamente as mesmas características de um *Tholos*, tal como se apresenta na figura abaixo, à esquerda:



O topo da torre tem uma forma semelhante à da base de um *Tholos*, embora com mais degraus.

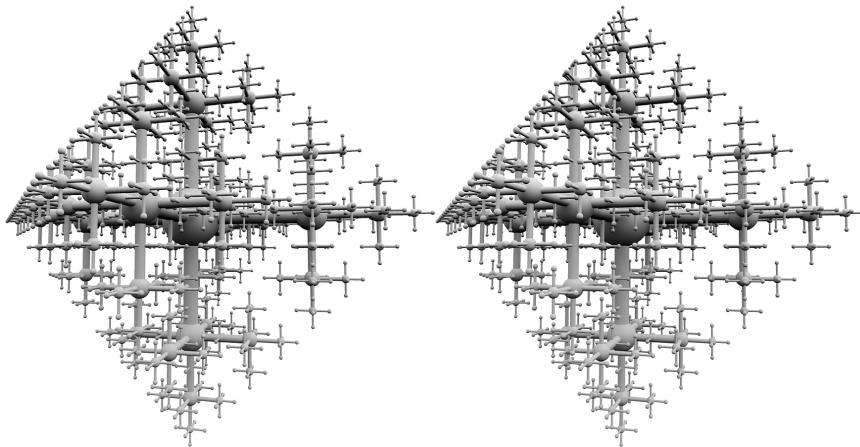
Defina a função `torre-tholos` que, a partir do centro da base da torre, do número de módulos, do número de degraus a considerar para o topo e dos restantes parâmetros necessários para definir um módulo idêntico a um *Tholos*, constrói a torre apresentada anteriormente.

Experimente a sua função criando uma torre composta por 6 módulos, com 10 degraus no topo, 3 degraus por módulo, qualquer deles com comprimento de espelho e de cobertor de 0.2, raio da base de 7.9 e 20 colunas por módulo, com raio de peristilo de 7 e altura de coluna de 4.

Exercicio 4.18 Com base na resposta ao exercício anterior, redefina a construção da torre de forma a que a dimensão radial dos módulos se vá reduzindo à medida que se ganha altura, tal como acontece na torre apresentada no centro da imagem anterior.

Exercicio 4.19 Com base na resposta ao exercício anterior, redefina a construção da torre de forma a que o número de colunas se vá também reduzindo à medida que se ganha altura, tal como acontece na torre da direita da imagem anterior.

Exercicio 4.20 Considere a criação de uma cidade no espaço, composta apenas por cilindros com dimensões progressivamente mais pequenas, unidos uns aos outros por intermédio de esferas, tal como se apresenta (em perspectiva) na seguinte imagem estereoscópica:³⁶



Defina uma função que, a partir do centro da cidade e do raio dos cilindros centrais constrói uma cidade semelhante à representada.

4.9 A Ordem Jónica

A *voluta* foi um dos elementos arquitectónicos introduzidos na transição da Ordem Dórica para a Ordem Jónica. Uma voluta é um ornamento em forma de espiral colocado em cada extremo de um capitel Jónico. A Figura 39 mostra um exemplo de um capitel Jónico contendo duas volutas. Embora tenham sobrevivido inúmeros exemplos de volutas desde a antiguidade, nunca foi claro o processo do seu desenho.

Vitrúvio, no seu tratado de arquitectura, descreve a voluta Jónica: uma curva em forma de espiral que se inicia na base do ábaco, desenrola-se numa série de voltas e junta-se a um elemento circular denominado o *olho*.

Vitrúvio descreve o processo de desenho da espiral através da composição de quartos de circunferência, começando pelo ponto mais exterior e diminuindo o raio a cada quarto de circunferência, até se dar a conjunção

³⁶Para visualizar a imagem estereoscópica, foque a atenção no meio das duas imagens e cruze os olhos, como se quisesse focar um objecto muito próximo. Irá reparar que as duas imagens passaram a ser quatro, embora ligeiramente desfocadas. Tente então alterar o cruzar dos olhos de modo a só ver três imagens, i.e., até que as duas imagens centrais fiquem sobrepostas. Concentre-se nessa sobreposição e deixe os olhos relaxarem até a imagem ficar focada.



Figura 39: Volutas de um capitel Jônico. Fotografia de See Wah Cheng.

com o olho. Nesta descrição há ainda alguns detalhes por explicar, em particular, o posicionamento dos centros dos quartos de circunferência, mas Vitrúvio refere que será incluída uma figura e um cálculo no final do livro.

Infelizmente, nunca se encontrou essa figura ou esse cálculo, ficando assim por esclarecer um elemento fundamental do processo de desenho de volutas descrito por Vitrúvio. As dúvidas sobre esse detalhe tornaram-se ainda mais evidentes quando a análise de muitas das volutas que sobreviveram a antiguidade revelou diferenças em relação às proporções descritas por Vitrúvio.

Durante a Renascença, estas dúvidas levaram os investigadores a repensar o método de Vitrúvio e a sugerir interpretações pessoais ou novos métodos para o desenho da voluta. De particular relevância foram os métodos propostos por:

- Sebastiano Serlio (1537), baseado na composição de semi-circunferências,
- Giuseppe Salviati (1552), baseado na composição de quartos-de-circunferência e
- Guillaume Philandrier (1544), baseado na composição de oitavos-de-circunferência.

Todos estes métodos diferem em vários detalhes mas, de forma genérica, todos se baseiam em empregar arcos de circunferência de abertura constante mas raio decrescente. Obviamente, para que haja continuidade nos arcos, os centros dos arcos vão mudando à medida que estes vão sendo desenhados. A Figura 40 esquematiza o processo para espirais feitas empregando quartos de circunferência.

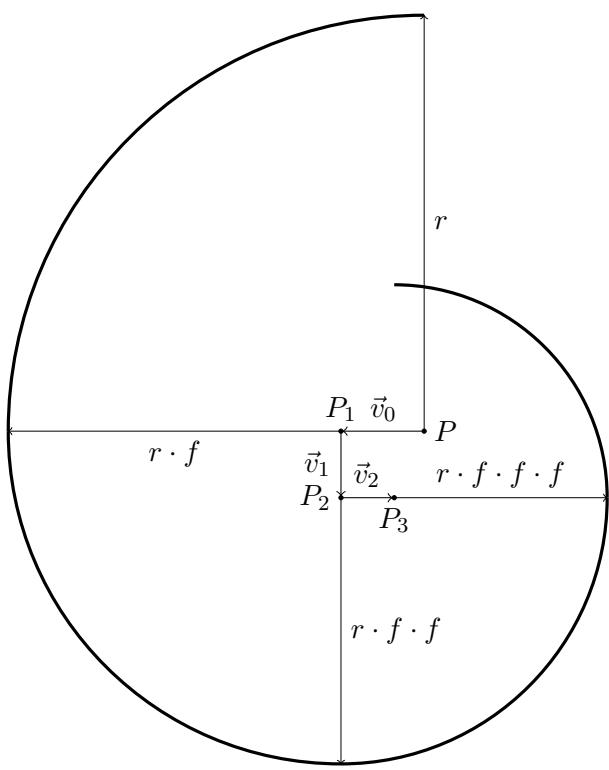


Figura 40: O desenho de uma espiral com arcos de circunferência.

Como se vê pela figura, para se desenhar a espiral temos de ir desenhando sucessivos quartos de circunferência. O primeiro quarto de circunferência será centrado no ponto P e terá raio r . Este primeiro arco vai desde o ângulo $\pi/2$ até ao ângulo π . O segundo quarto de circunferência será centrado no ponto P_1 e terá raio $r \cdot f$, sendo f um coeficiente de “redução” da espiral. Este segundo arco vai desde o ângulo π até ao ângulo $\frac{3}{2}\pi$. Um detalhe importante é a relação entre as coordenadas P_1 e P : para que o segundo arco tenha uma extremidade coincidente com o primeiro arco, o seu centro tem de estar na extremidade do vector \vec{v}_0 de origem em P , comprimento $r(1 - f)$ e ângulo igual ao ângulo final do primeiro arco.

Este processo deverá ser seguido para todos os restantes arcos de circunferência, i.e., teremos de calcular as coordenadas P_2, P_3 , etc., bem como os raios $r \cdot f \cdot f, r \cdot f \cdot f \cdot f$, etc, necessários para traçar os sucessivos arcos de circunferência.

Dito desta forma, o processo de desenho parece ser complicado. No entanto, é possível reformulá-lo de modo a ficar muito mais simples. De facto, podemos pensar no desenho da espiral completa como sendo o desenho de um quarto de circunferência seguido do desenho de uma espiral mais pequena. Mais concretamente, podemos especificar o desenho da espiral de centro no ponto P , raio r e ângulo inicial θ como sendo o desenho de um arco de circunferência de raio r centrado em P com ângulo inicial θ e final $\theta + \frac{\pi}{2}$ seguido de uma espiral de centro em $P + \vec{v}$, raio $r \cdot f$ e ângulo inicial $\theta + \frac{\pi}{2}$. O vector \vec{v} terá origem em P , módulo $r(1 - f)$ e ângulo $\theta + \frac{\pi}{2}$.

Obviamente, sendo este um processo recursivo, é necessário definir o caso de paragem, havendo (pelo menos) duas possibilidades:

- Terminar quando o raio r é inferior a um determinado limite.
- Terminar quando o ângulo θ é superior a um determinado limite.

Por agora, vamos considerar a segunda possibilidade. De acordo com o nosso raciocínio, vamos definir a função que desenha a espiral de modo a receber, como parâmetros, o ponto inicial p , o raio inicial r , o ângulo inicial $a\text{-ini}$, o ângulo final $a\text{-fin}$ e o factor de redução f :

```
(defun espiral (p r a-ini a-fin f)
  (if (> a-ini a-fin)
      nil
      (progn
        (quarto-circunferencia p r a-ini)
        (espiral (+pol p (* r (- 1 f)) (+ a-ini (/ pi 2)))
                  (* r f)
                  (+ a-ini (/ pi 2))
                  a-fin
                  f))))
```

Reparemos que a função `espiral` é recursiva, pois está definida em termos de si própria. Obviamente, o caso recursivo é mais simples que o

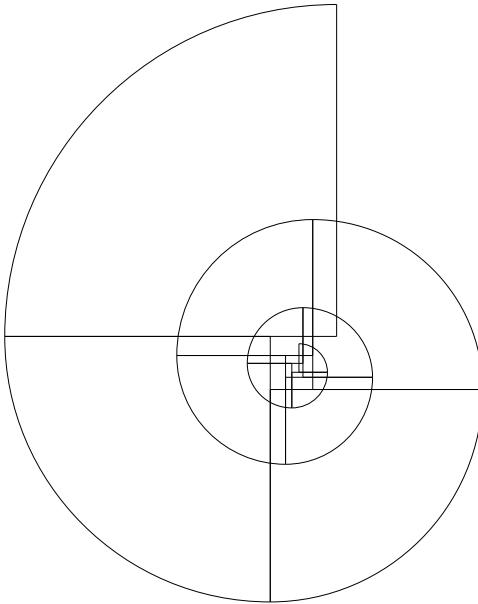


Figura 41: O desenho da espiral.

caso original, pois a diferença entre o ângulo inicial e o final é mais pequena, aproximando-se progressivamente do caso de paragem em que o ângulo inicial ultrapassa o final.

Para desenhar o quarto de circunferência vamos empregar a operação `arc` do Auto Lisp que recebe o centro do circunferência, o ponto inicial do arco e o ângulo em graus. Para melhor percebermos o processo de desenho da espiral vamos também traçar duas linhas com origem no centro a delimitar cada quarto de circunferência. Mais tarde, quando tivermos terminado o desenvolvimento destas funções, removeremos essas linhas.

Desta forma, o quarto de circunferência apenas precisa de saber o ponto `p` correspondente ao centro da circunferência, o raio `r` da mesma e o ângulo inicial `a-ini`:

```
(defun quarto-circunferencia (p r a-ini)
  (command "_.arc" "_c" p (+pol p r a-ini) "_a" 90)
  (command "_.line" p (+pol p r a-ini) "")
  (command "_.line" p (+pol p r (+ a-ini (/ pi 2)))) "")
```

Podemos agora experimentar um exemplo:

```
(espiral (xy 0 0) 10 (/ pi 2) (* pi 6) 0.8)
```

A espiral traçada pela expressão anterior está representada na Figura 41.

A função `espiral` permite-nos definir um sem-número de espirais, mas tem uma restrição: cada arco de círculo corresponde a um ângulo de

$\frac{\pi}{2}$. Logicamente, a função tornar-se-á mais útil se também este incremento de ângulo for um parâmetro.

As modificações a fazer são relativamente triviais, bastando acrescentar um parâmetro *a-inc*, representando o incremento de ângulo de cada arco e substituir as ocorrências de $(/\ pi 2)$ por este parâmetro. Naturalmente, em vez de desenharmos um quarto de circunferência, temos agora de desenhar um arco de circunferência.

A nova definição é, então:

```
(defun espiral (p r a-ini a-inc a-fin f)
  (if (> a-ini a-fin)
      nil
      (progn
        (arco p r a-ini a-inc)
        (espiral (+pol p (* r (- 1 f)) (+ a-ini a-inc))
                  (* r f)
                  (+ a-ini a-inc)
                  a-inc
                  a-fin
                  f))))
```

A função que desenha o arco é uma variante da que desenha o quarto de circunferência. Apenas é preciso ter em conta que o comando *arc* pretende o ângulo em graus e não em radianos, o que nos obriga a usar uma conversão:

```
(defun arco (p r a-ini a-inc)
  (command "_.arc"
    "_c" p (+pol p r a-ini)
    "_a" (graus<-radianos a-inc))
  (command "_.line" p (+pol p r a-ini) "")
  (command "_.line" p (+pol p r (+ a-ini a-inc)) ""))
  
(defun graus<-radianos (radianos)
  (* (/ 180 pi) radianos))
```

Agora, para desenhar a mesma espiral representada na Figura 41, temos de avaliar a expressão:

```
(espiral (xy 0 0) 10 (/ pi 2) (/ pi 2) (* pi 6) 0.8)
```

É claro que agora podemos facilmente construir outras espirais. As seguintes expressões produzem as espirais representadas na Figura 42:

```
(espiral (xy 0 0) 10 (/ pi 2) (/ pi 2) (* pi 6) 0.9)
(espiral (xy 20 0) 10 (/ pi 2) (/ pi 2) (* pi 6) 0.7)
(espiral (xy 40 0) 10 (/ pi 2) (/ pi 2) (* pi 6) 0.5)
```

Outra possibilidade de variação está no ângulo de incremento. As seguintes expressões experimentam aproximações aos processos de Sebasti-

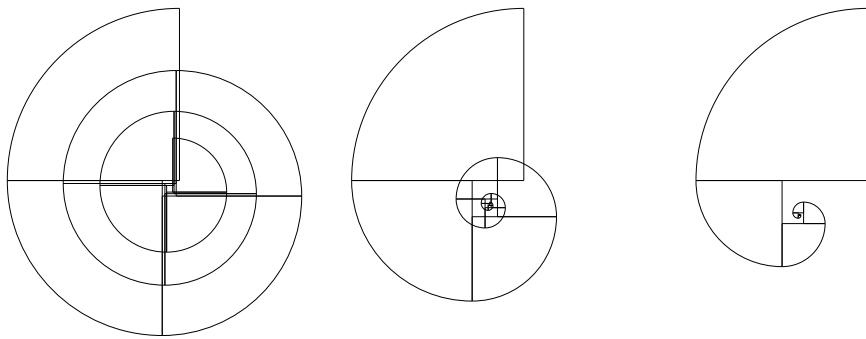


Figura 42: Várias espirais com razões de redução de 0.9, 0.7 e 0.5, respectivamente.

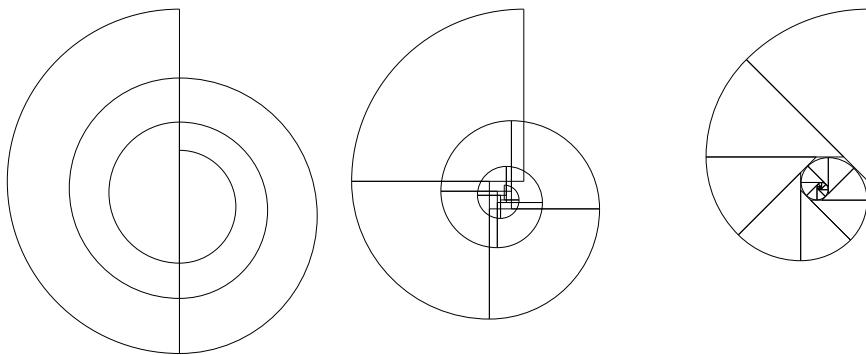


Figura 43: Várias espirais com razão de redução de 0.8 e incremento de ângulo de π , $\frac{\pi}{2}$ e $\frac{\pi}{4}$, respectivamente.

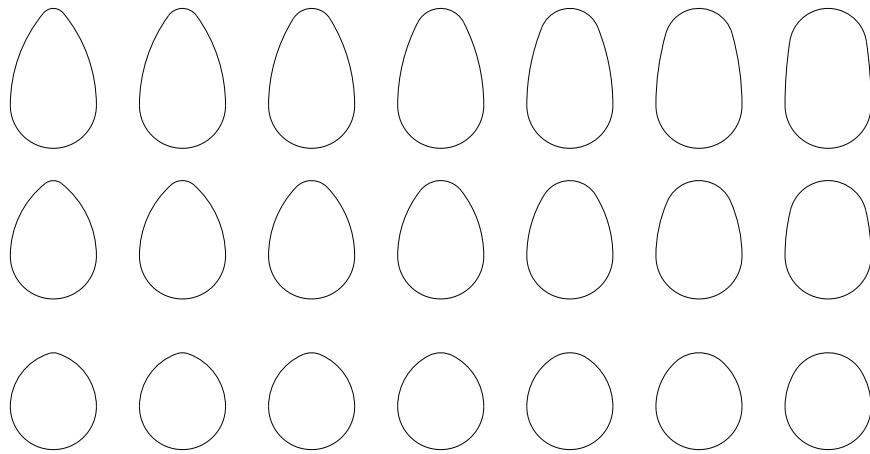
ano Serlio (semi-circunferências), Giuseppe Salviati (quartos-de-circunferência) e Guillaume Philandrier (oitavos-de-circunferência):³⁷

```
(espiral (xy 0 0) 10 (/ pi 2) pi (* pi 6) 0.8)
(espiral (xy 20 0) 10 (/ pi 2) (/ pi 2) (* pi 6) 0.8)
(espiral (xy 40 0) 10 (/ pi 2) (/ pi 4) (* pi 6) 0.8)
```

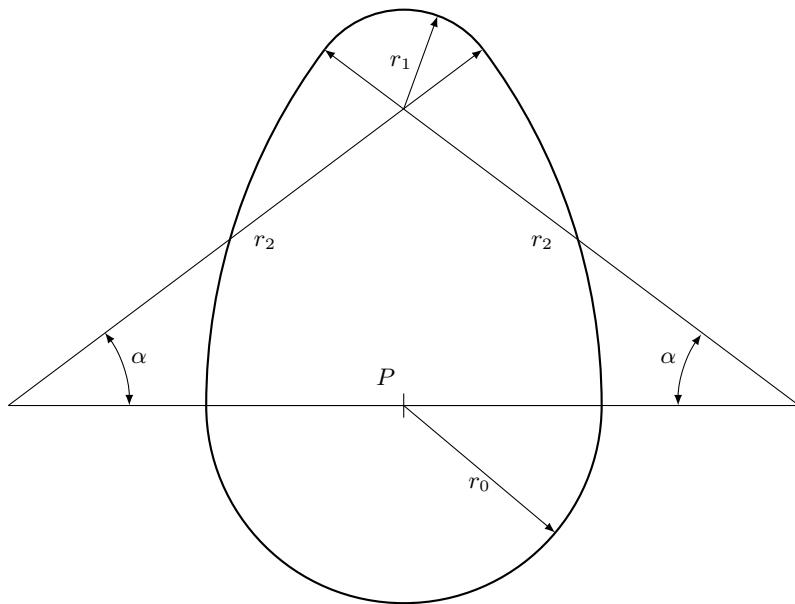
Os resultados estão representados na Figura 43.

Exercicio 4.21 Um *óvulo* é uma figura geométrica que corresponde ao contorno dos ovos das aves. Dada a variedade de formas de ovos na Natureza, é natural considerar que também existe uma grande variedade de óvulos geométricos. A figura seguinte apresenta alguns exemplos em que se variam sistematicamente alguns dos parâmetros que caracterizam o óvulo:

³⁷Note-se que se trata, tão somente, de aproximações. Os processos originais eram bastante mais complexos.



Um óvulo é composto por quatro arcos de circunferência concordantes, tal como se apresenta na imagem seguinte:



Os arcos de círculos necessários para a construção do ovo são caracterizados pelos raios r_0 , r_1 e r_2 . Note que o arco de circunferência de raio r_0 cobre um ângulo de π e o arco de circunferência de raio r_2 cobre um ângulo de α .

Defina uma função `ovo` que desenha um ovo. A função deverá receber, apenas, as coordenadas do ponto P , os raios r_0 e r_1 e, finalmente, a altura h do ovo.

4.10 Recursão na Natureza

A recursão está presente em inúmeros fenómenos naturais. As montanhas, por exemplo, apresentam irregularidades que, quando observadas numa escala apropriada, são em tudo idênticas a ... montanhas. Um rio possui afluentes e cada afluente é idêntico a ... um rio. Uma vaso sanguíneo



Figura 44: A estrutura recursiva das árvores. Fotografia de Michael Bezina.

possui ramificações e cada ramificação é idêntica a ... um vaso sanguíneo. Todas estas entidades naturais constituem exemplos de estruturas recursivas.

Uma árvore é outro bom exemplo de uma estrutura recursiva, pois os ramos de uma árvore são como pequenas árvores que emanam do tronco. Como se pode ver da Figura 44, de cada ramo de uma árvore emanam outras pequenas árvores, num processo que se repete até se atingir uma dimensão suficientemente pequena em que aparecem outras estruturas, como folhas, flores, frutos, pinhas, etc.

Se, de facto, uma árvore possui uma estrutura recursiva, então deverá ser possível “construir” árvores através de funções recursivas. Para testarmos esta teoria, vamos começar por considerar uma versão muito simplista de uma árvore, em que temos um tronco que, a partir de uma certa altura, se divide em dois. Cada um destes subtroncos cresce fazendo um certo ângulo com o tronco de onde emanou e com um comprimento que deverá ser uma fração do comprimento desse tronco, tal como se apresenta na Figura 45. O caso de paragem ocorre quando o comprimento do tronco se tornou tão pequeno que, em vez de se continuar a divisão, aparece simplesmente uma outra estrutura. Para simplificar, vamos designar a extremidade de um ramo por folha e iremos representá-la com um pequeno círculo.

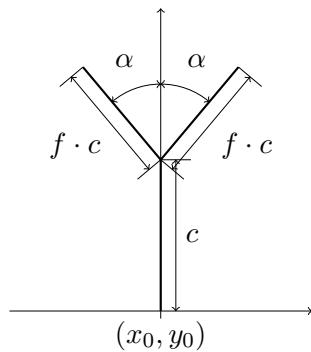


Figura 45: Parâmetros de desenho de uma árvore.

Para darmos dimensões à árvore, vamos considerar que a função `arvore` recebe, como argumento, as coordenadas da base da árvore, o comprimento do tronco e o ângulo actual do tronco. Para a fase recursiva, teremos como parâmetros o ângulo de abertura `alfa` que o novo tronco deverá fazer com o actual e o factor de redução `f` do comprimento do tronco. O primeiro passo é a computação do topo do tronco usando a função `+pol`. Em seguida, desenhamos o tronco desde a base até ao topo. Finalmente, testamos se o tronco desenhado é suficientemente pequeno. Se for, terminamos com o desenho de um círculo centrado no topo. Caso contrário fazemos uma dupla recursão para desenhar uma sub-árvore para a direita e outra para a esquerda. A definição da função fica:

```
(defun arvore (base comprimento angulo alfa f / topo)
  (setq topo (+pol base comprimento angulo))
  (ramo base topo)
  (if (< comprimento 2)
      (folha topo)
      (progn
        (arvore topo
          (* comprimento f)
          (+ angulo alfa)
          alfa f)
        (arvore topo
          (* comprimento f)
          (- angulo alfa)
          alfa f)))))

(defun ramo (base topo)
  (command "_.line" base topo ""))
  
(defun folha (topo)
  (command "_.circle" topo 0.2))
```

Um primeiro exemplo de árvore gerado com a expressão

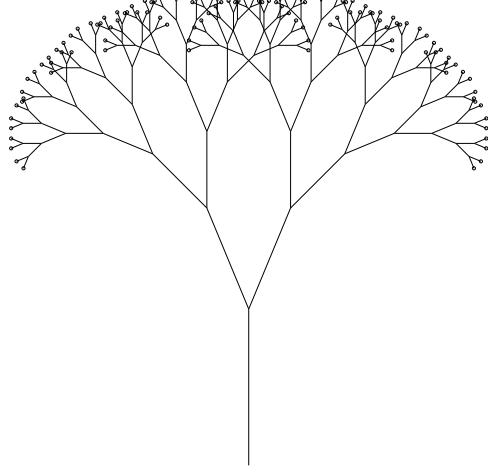


Figura 46: Uma “árvore” de comprimento 20, ângulo inicial $\frac{\pi}{2}$, abertura $\frac{\pi}{8}$ e factor de redução 0.7.

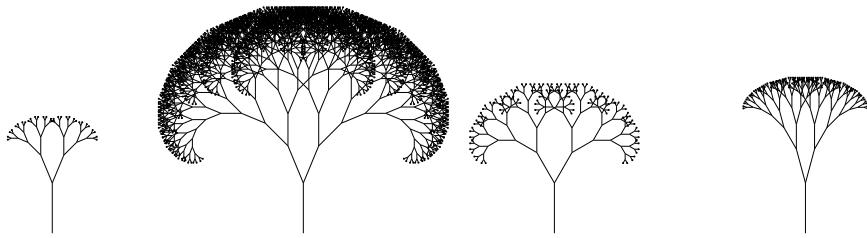


Figura 47: Várias “árvores” geradas com diferentes ângulos de abertura e factores de redução do comprimento dos ramos.

```
(arvore (xy 0 0) 20 (/ pi 2) (/ pi 8) 0.7)
```

está representado na Figura 46.

A Figura 47 apresenta outros exemplos em que se fez variar o ângulo de abertura e o factor de redução. A sequência de expressões que as gerou foi a seguinte:

```
(arvore (xy 0 0) 20 (/ pi 2) (/ pi 8) 0.6)
(arvore (xy 100 0) 20 (/ pi 2) (/ pi 8) 0.8)
(arvore (xy 200 0) 20 (/ pi 2) (/ pi 6) 0.7)
(arvore (xy 300 0) 20 (/ pi 2) (/ pi 12) 0.7)
```

Infelizmente, as árvores apresentadas são “excessivamente” simétricas: no mundo natural é literalmente impossível encontrar simetrias perfeitas. Por este motivo, convém tornar o modelo um pouco mais sofisticado através da introdução de parâmetros diferentes para o crescimento dos troncos

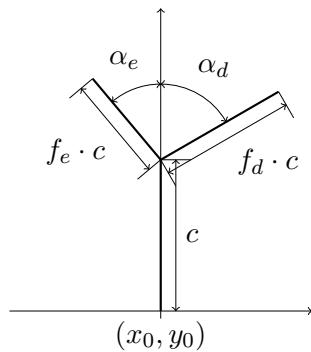


Figura 48: Parâmetros de desenho de uma árvore com crescimento assimétrico.

à direita e à esquerda. Para isso, em vez de termos um só ângulo de abertura e um só factor de redução de comprimento, vamos empregar dois, tal como se apresenta na Figura 48.

A adaptação da função arvore para lidar com os parâmetros adicionais é trivial:

```
(defun arvore (base comprimento angulo
              alfa-e f-e alfa-d f-d
              / topo)
  (setq topo (+pol base comprimento angulo))
  (ramo base topo)
  (if (< comprimento 2)
      (folha topo)
      (progn
        (arvore topo
                (* comprimento f-e)
                (+ angulo alfa-e)
                alfa-e f-e alfa-d f-d)
        (arvore topo
                (* comprimento f-d)
                (- angulo alfa-d)
                alfa-e f-e alfa-d f-d))))
```

A Figura 49 apresenta novos exemplos de árvores com diferentes ângulos de abertura e factores de redução dos ramos esquerdo e direito, geradas pelas seguintes expressões:

```
(arvore (xy 0 0) 20 (/ pi 2) (/ pi 8) 0.6 (/ pi 8) 0.7)
(arvore (xy 100 0) 20 (/ pi 2) (/ pi 4) 0.7 (/ pi 16) 0.7)
(arvore (xy 200 0) 20 (/ pi 2) (/ pi 6) 0.6 (/ pi 16) 0.8)
```

As árvores geradas pela função arvore são apenas um modelo muito grosso da realidade. Embora existam sinais evidentes de que vários fe-

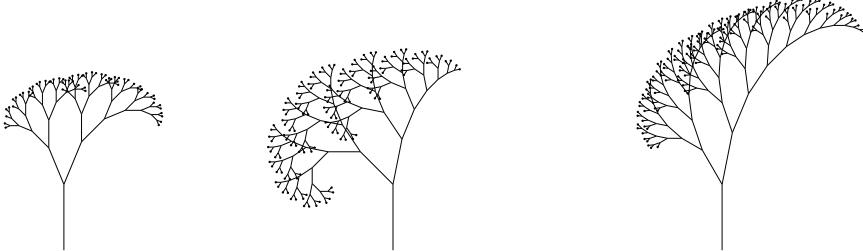


Figura 49: Várias árvores geradas com diferentes ângulos de abertura e factores de redução do comprimento para os ramos esquerdo e direito.

nómenos naturais se podem modelar através de funções recursivas, a natureza não é tão determinista quanto as nossas funções e, para que a modelação se aproxime mais da realidade, é fundamental incorporar também alguma aleatoriedade. Esse será o tema da próxima secção.

5 Atribuição

Como vimos, a *atribuição* é a capacidade de estabelecermos uma associação entre um nome e uma entidade. Como iremos ver, a atribuição permite-nos também alterar a associação entre um nome e uma entidade, fazendo com que o nome passe a estar associado a uma outra entidade. Este conceito introduz uma *história* no valor do nome, pois passamos a poder falar do valor *antes* da atribuição e do valor *depois* da atribuição.

Nesta secção vamos discutir mais em profundidade o conceito de atribuição. Para motivar vamos começar por introduzir um tópico importante onde a atribuição tem um papel primordial: a aleatoriedade.

5.1 Aleatoriedade

Desenhar implica tomar decisões conscientes que conduzam a um objectivo pretendido. Neste sentido, desenhar aparenta ser um processo racional onde não há lugar para a *aleatoriedade*, a *sorte* ou a *incerteza*. De facto, nos desenhos que temos feito até agora a racionalidade tem sido crucial, pois o computador exige uma especificação rigorosa do que se pretende, não permitindo quaisquer ambiguidades. No entanto, é sabido que um problema de desenho frequentemente exige que o arquitecto experimente diferentes soluções antes de encontrar aquela que o satisfaz. Essa experimentação passa por empregar alguma aleatoriedade no processo de escolha dos “parâmetros” do desenho. Assim, embora um desenho final possa apresentar uma estrutura que espelha uma intenção racional do arquitecto, o processo que conduziu a esse desenho final não é necessariamente racional e pode ter passado por fases de ambiguidade e incerteza.

Quando a arquitectura se inspira na natureza surge um factor adicional de aleatoriedade: em inúmeros casos, a natureza é intrinsecamente aleatória. Essa aleatoriedade, contudo, não é total e está sujeita a restrições. Este facto é facilmente compreendido quando pensamos que, embora não existam dois pinheiros iguais, todos somos capazes de reconhecer o padrão que caracteriza um pinheiro. Em todas as suas obras, a natureza combina estruturação e aleatoriedade. Nalguns casos, como o crescimento de cristais, por exemplo, há mais estruturação que aleatoriedade. Noutros casos, como no comportamento de partículas subatómicas, há mais aleatoriedade que estruturação. À semelhança do que acontece na natureza, esta combinação de aleatoriedade e estruturação é claramente visível em inúmeras obras arquitecturais modernas.

Se pretendemos empregar computadores para o desenho e este desenho pressupõe aleatoriedade, então teremos de ser capazes de a incorporar nos nossos algoritmos. A aleatoriedade pode ser incorporada de inúmeras formas, como, por exemplo:

- A cor de um artefacto pode ser escolhida aleatoriamente.
- O comprimento de uma parede pode ser um valor aleatório entre determinados limites.
- A decisão de dividir uma área ou mantê-la íntegra pode ser tomada aleatoriamente.
- A forma geométrica a empregar para um determinado elemento arquitectónico pode ser escolhida aleatoriamente.

5.2 Números Aleatórios

Em qualquer dos casos anteriores, podemos reduzir a aleatoriedade à escolha de números dentro de certos limites. Para uma cor aleatória, podemos gerar aleatoriamente três números que representem os valores RGB³⁸ da cor. Para um comprimento aleatório, podemos gerar aleatoriamente um número dentro dos limites desse comprimento. Para uma decisão lógica, podemos gerar aleatoriamente um número inteiro entre zero e um, decidindo de um modo se o número for zero, e de outro modo se o número for um. Para uma escolha aleatória de entre um conjunto de alternativas podemos simplesmente gerar um número aleatório entre um e o número de elementos do conjunto e escolher a alternativa correspondente ao número aleatório gerado.

³⁸RGB é a abreviatura de *Red-Green-Blue*, um modelo de cores em que qualquer cor é vista como uma combinação das três cores primárias vermelho, verde e azul.

Estes exemplos mostram-nos que o fundamental é conseguirmos gerar um número aleatório dentro de um certo intervalo. A partir dessa operação podemos implementar todas as outras.

Há dois processos fundamentais para se gerarem números aleatórios. O primeiro processo baseia-se na medição de processos físicos intrínsecamente aleatórios, como seja o ruido electrónico ou o decaimento radioativo. O segundo processo baseia-se na utilização de funções aritméticas que, a partir de um valor inicial (denominado a *semente*), produzem uma sequência de números aparentemente aleatórios, sendo cada número da sequência gerado a partir do número anterior. Neste caso dizemos que estamos perante um *gerador de números pseudo-aleatórios*. O termo *pseudo* justifica-se, pois, se repetirmos o valor da semente original, repetiremos também a sequência de números de gerados.

Muito embora um gerador de números pseudo-aleatórios gere uma sequência de números que, na verdade, não são aleatórios, ele possui duas importantes vantagens:

- Pode ser facilmente implementado usando uma qualquer linguagem de programação, não necessitando de outros mecanismos adicionais para se obter a fonte de aleatoriedade.
- É frequente os nossos programas conterem erros. Para identificarmos a causa do erro pode ser necessário reproduzirmos exactamente as mesmas condições que deram origem ao erro. Para além disso, depois da correcção do erro é necessário repetirmos novamente a execução do programa para nos certificarmos que o erro não ocorre de novo. Infelizmente, quando o comportamento de um programa depende de uma sequência de números verdadeiramente aleatórios torna-se impossível reproduzir as condições de execução: da próxima vez que executarmos o programa, a sequência de números aleatórios será quase de certeza diferente.

Por estes motivos, de agora em diante vamos apenas considerar geradores de números pseudo-aleatórios, que iremos abusivamente designar por geradores de números aleatórios. Um gerador deste tipo caracteriza-se por uma função f que, a partir de um argumento x_i , produz um número $x_{i+1} = f(x_i)$ aparentemente não relacionado com x_i . A semente do gerador é o elemento x_0 da sequência.

Falta agora encontrarmos uma função f apropriada. Para isso, entre outras qualidades, exige-se que os números gerados por f sejam *equiprováveis*, i.e., todos os números dentro de um determinado intervalo têm igual probabilidade de serem gerados e que a sequência de números gerados tenha um *período* tão grande quanto possível, i.e., que a sequência dos números gerados só se comece a repetir ao fim de muito tempo.

Têm sido estudadas inúmeras funções com estas características. A mais utilizada é denominada *função geradora congruencial linear* que é da forma³⁹

$$x_{i+1} = (ax_i + b) \bmod m$$

Por exemplo, se tivermos $a = 25173$, $b = 13849$, $m = 65536$ e começarmos com uma semente qualquer, por exemplo, $x_0 = 12345$, obtemos a sequência de números pseudo-aleatórios⁴⁰ 2822, 11031, 21180, 42629, 27202, 49667, 50968, 33041, 37566, 43823, 2740, 43997, 57466, 29339, 39312, 21225, 61302, 58439, 12204, 57909, 39858, 3123, 51464, 1473, 302, 13919, 41380, 43405, 31722, 61131, 13696, 63897, 42982, 375, 16540, 25061, 24866, 31331, 48888, 36465, ...

Podemos facilmente confirmar estes resultados usando o Lisp:

```
(defun proximo-aleatorio (ultimo-aleatorio)
  (rem (+ (* 25173 ultimo-aleatorio) 13849)
        65536))

_ $ (proximo-aleatorio 12345)
2822
_ $ (proximo-aleatorio 2822)
11031
_ $ (proximo-aleatorio 11031)
21180
```

Esta abordagem, contudo, implica que só podemos gerar um número “aleatório” x_{1+i} se nos recordarmos do número x_i gerado imediatamente antes para o podermos dar como argumento à função `proximo-aleatorio`. Infelizmente, o momento e o ponto do programa em que podemos precisar de um novo número aleatório pode ocorrer muito mais tarde e muito mais longe do que o momento e o ponto do programa em que foi gerado o último número aleatório, o que complica substancialmente a escrita do programa. Seria preferível que, ao invés da função `proximo-aleatorio`, que depende do último valor gerado x_i , tivéssemos uma função `aleatorio` que não precisasse de receber o último número gerado para conseguir gerar o próximo número. Assim, em qualquer ponto do programa em que precisássemos de gerar um novo número aleatório, limitávamo-nos a invocar a função `aleatorio` sem termos de nos recordar do último número gerado. Partindo do mesmo valor da semente, teríamos:

³⁹O operador `mod` implementa o *modulo*, correspondente ao resto da divisão do primeiro operando (o dividendo, à esquerda do operador) pelo segundo operando (o divisor, à direita do operador) e tendo o mesmo sinal do divisor. O operador `rem` implementa o *remainder*, correspondente ao resto da divisão do dividendo pelo divisor, mas tendo o mesmo sinal do dividendo. Nem todas as linguagens implementam ambos os operadores. A linguagem Auto Lisp apenas implementa o operador `rem`.

⁴⁰Na verdade, esta sequência não é suficientemente aleatória pois existe um padrão que se repete continuamente. Consegue descobri-lo?

```
_§ (aleatorio)
2822
_§ (aleatorio)
11031
_§ (aleatorio)
21180
```

5.3 Estado

A função `aleatorio` apresenta um comportamento diferente do das funções que vimos até agora. Até este momento, todas as funções que tínhamos definido comportavam-se como as funções matemáticas tradicionais: dados argumentos, a função produz resultados e, mais importante, dados os mesmos argumentos, a função produz *sempre* os mesmos resultados. Por exemplo, independentemente do número de vezes que invocarmos a função `quadrado`, para um dado argumento ela irá *sempre* produzir o quadrado desse argumento.

A função `aleatorio` é diferente de todas as outras, pois, para além de não precisar de argumentos, ela produz um resultado diferente de cada vez que é invocada.

Do ponto de vista matemático, uma função sem parâmetros não tem nada de anormal: é precisamente aquilo que se designa por uma *constante*. De facto, tal como escrevemos $\sin \cos x$ para designar $\sin(\cos(x))$, também escrevemos $\sin \pi$ para designar $\sin(\pi())$, onde se vê que π pode ser interpretado como uma função sem argumentos.

Do ponto de vista matemático, uma função que produz resultados diferentes a cada invocação não tem nada de normal: é uma aberração, pois, de acordo com a definição matemática de função, esse comportamento não é possível. E, no entanto, é precisamente este o comportamento que gostaríamos de ter na função `aleatorio`.

Para se obter o pretendido comportamento é necessário introduzir o conceito de *estado*. Dizemos que uma função tem estado quando o seu comportamento depende da sua história, i.e., das suas invocações anteriores. A função `aleatório` é um exemplo de uma função com estado, mas há inúmeros outros exemplos no mundo real. Uma conta bancária também tem um estado que depende de todas as transacções anteriores que lhe tiverem sido feitas. O depósito de combustível de um carro também tem um estado que depende dos enchimentos e dos trajectos realizados.

Para que uma função possa ter história é necessário que tenha *memória*, i.e., que se possa recordar de acontecimentos passados para assim poder influenciar os resultados futuros. Até agora vimos que o operador `setq` nos permitia *definir* associações entre nomes e valores, mas o que ainda não tínhamos discutido é a possibilidade desse operador *modificar* associações, i.e., alterar o valor que estava associado a um determinado nome. É esta possibilidade que nos permite incluir memória nas nossas funções.

No caso da função `aleatorio` vimos que a memória que nos interessa é saber qual foi o último número aleatório gerado. Imaginemos então que tínhamos uma associação entre o nome `ultimo-aleatorio-gerado` e esse número. No momento inicial, naturalmente, esse nome deverá estar associado à semente da sequência de números aleatórios. Para isso, definimos:

```
_\$ (setq ultimo-aleatorio-gerado 12345)
```

De seguida, já podemos definir a “função” `aleatorio` que não só usa o último valor associado àquele nome como actualiza essa associação com o novo valor gerado, i.e.⁴¹

```
(defun aleatorio ()
  (setq ultimo-aleatorio-gerado
        (proximo-aleatorio ultimo-aleatorio-gerado)))
```

```
_\$ (aleatorio)
2822
_\$ (aleatorio)
11031
```

Como se vê, de cada vez que a função `aleatorio` é invocada, o valor associado a `ultimo-aleatorio-gerado` é actualizado, permitindo assim influenciar o futuro comportamento da função. Obviamente, em qualquer momento, podemos re-iniciar a sequência de números aleatórios simplesmente repondo o valor da semente:

```
_\$ (aleatorio)
21180
_\$ (aleatorio)
42629
_\$ (setq ultimo-aleatorio-gerado 12345)
12345
_\$ (aleatorio)
2822
_\$ (aleatorio)
11031
_\$ (aleatorio)
21180
```

5.4 Estado Local e Estado Global

Vimos anteriormente que uma função pode ter variáveis locais bastando, para isso, a sua indicação a seguir aos parâmetros da função. Cada variável local possui um nome que apenas é visível para a função que a contém.⁴² Acontece que no caso da função `aleatorio`, ela usa uma variá-

⁴¹Note-se que o operador `setq`, para além de estabelecer uma associação entre um nome e um valor devolve o valor que ficou associado.

⁴²Esta afirmação não é inteiramente verdade para o Auto Lisp. Mais à frente explicaremos aquilo que verdadeiramente acontece no caso do Auto Lisp.

vel denominada `ultimo-aleatorio-gerado` que não é referida na lista de parâmetros da função. Consequentemente, não é uma variável local, dizendo-se então que é uma variável *livre*.

Quando uma variável livre é visível de todos os lados, tal como acontece com `ultimo-aleatorio-gerado`, dizemos que se trata de uma variável *global*. Ao conjunto de todas as variáveis globais de um programa chama-se *estado global* pois, em cada instante, elas determinam o estado do programa.

Uma variável global é, potencialmente, uma enorme fonte de problemas, pois pode ser alterada em qualquer momento e a partir de qualquer ponto do programa. Quando essa alteração é incorrecta pode ser muito difícil saber quando e onde é que o erro foi cometido.

No entanto, nalguns casos como o da função `aleatorio`, a única forma de podermos dar memória às funções é através de variáveis globais. De facto, em Auto Lisp, as variáveis locais possuem *duração dinâmica*, i.e., apenas existem enquanto a função que as introduz está a ser invocada (e existem ocorrências *diferentes* dessas variáveis para diferentes invocações da função). Por este motivo, em Auto Lisp, uma variável local não pode nunca ser usada para manter uma memória que dure mais do que a invocação da função. Isto implica que não é possível termos *estado local* em Auto Lisp, i.e., não é possível termos um conjunto de variáveis que descrevem o estado de uma função mas que não são visíveis globalmente.⁴³

Por estes motivos, em Auto Lisp, a única possibilidade que temos de dar memória às nossas funções é através da utilização de variáveis globais. Pelos problemas apontados devemos, contudo, ter extremo cuidado com a utilização de variáveis globais e devemos estar alerta para a possibilidade de ocorrerem conflitos de nomes, em que duas funções independentes pretendem definir uma mesma variável global.⁴⁴ Para evitar este último problema, as nossas variáveis globais devem ter nomes perfeitamente explícitos e com pouca probabilidade de aparecerem repetidos em contextos diferentes.

5.5 Escolhas Aleatórias

Se observarmos a definição da função `proximo-aleatorio` constatamos que a sua última operação é computar o resto da divisão por 65536, o que implica que a função produz sempre valores entre 0 e 65535. Embora (pseudo) aleatórios, estes valores estão contidos num intervalo que só muito excepcionalmente será útil. Na realidade, é muito mais frequente

⁴³Há outros dialectos de Lisp e muitas outras linguagens de programação que permitem o estado local.

⁴⁴Alguns dialectos de Lisp possuem mecanismos que permitem evitar estes conflitos. O Auto Lisp, infelizmente, não possui qualquer destes mecanismos.

precisarmos de números aleatórios contidos em intervalos muito mais pequenos. Por exemplo, se pretendermos simular o lançar de uma moeda ao ar, estaremos interessados em ter apenas os números aleatórios 0 ou 1, representando “caras” ou “coroas.”

Tal como os números aleatórios que produzimos são limitados ao intervalo $[0, 65536]$ pela obtenção do resto da divisão por 65536, também agora podemos voltar aplicar a mesma operação para produzir um intervalo mais pequeno. Assim, no caso do lançamento de uma moeda ao ar, podemos simplesmente usar a função `aleatorio` para gerar um número aleatório e, de seguida, aplicamos-lhe o resto da divisão por dois. No caso geral, em que pretendemos números aleatórios no intervalo $[0, x]$, aplicamos o resto da divisão por x . Assim, podemos definir uma nova função que gera um número aleatório entre zero e o seu parâmetro e que, por tradição, iremos baptizar de `random`:

```
(defun random (x)
  (rem (aleatorio) x))
```

Note-se que a função `random` nunca deve receber um argumento maior que 65535 pois isso faria a função perder a característica da *equiprobabilidade* dos números gerados: todos os números superiores a 65535 terão probabilidade zero de ocorrerem.⁴⁵

É agora possível simularmos inúmeros fenómenos aleatórios como, por exemplo, o lançar de uma moeda:

```
(defun cara-ou-coroa ()
  (if (= (random 2) 0)
    "cara"
    "coroa"))
```

Infelizmente, quando testamos repetidamente a nossa função, o seu comportamento parece muito pouco aleatório:

```
_ $ (cara-ou-coroa)
"cara"
_ $ (cara-ou-coroa)
"coroa"
_ $ (cara-ou-coroa)
"cara"
_ $ (cara-ou-coroa)
"coroa"
_ $ (cara-ou-coroa)
"cara"
_ $ (cara-ou-coroa)
"coroa"
```

Na realidade, os resultados que obtemos são uma repetição sistemática

⁴⁵Na verdade, o limite superior deve ser bastante inferior ao limite da função `aleatorio` para manter a equiprobabilidade dos resultados.

do par cara/coroa, o que mostra que a expressão (random 2) se limita a gerar a sequência:

O mesmo fenômeno ocorre para outros limites do intervalo de geração. Por exemplo, a expressão `(random 4)` deveria gerar um número aleatório no conjunto $\{0, 1, 2, 3\}$ mas a sua aplicação repetida gera a seguinte sequência de números:

Embora a equiprobabilidade dos números seja perfeita, é evidente que não há qualquer aleatoriedade.

O problema das duas sequências anteriores está no facto de terem um período muito pequeno. O período é número de elementos que são gerados antes de o gerador entrar em ciclo e voltar a repetir os mesmos elementos que gerou anteriormente. Obviamente, quanto maior for o período, melhor é o gerador de números aleatórios e, neste sentido, o gerador que apresentámos é de baixa qualidade.

Enormes quantidades de esforço têm sido investidas na procura de bons geradores de números aleatórios e, embora os melhores sejam produzidos usando métodos bastante mais sofisticados do que os que empregámos até agora, também é possível encontrar um bom gerador congruencial linear desde que se faça uma judiciosa escolha dos seus parâmetros. De acordo com [?], o gerador congruencial linear

$$x_{i+1} = (ax_i + b) \bmod m$$

pode constituir um bom gerador pseudo-aleatório desde que tenhamos $a = 16807$, $b = 0$ e $m = 2^{31} - 1 = 2147483647$. Uma tradução directa da definição matemática produz a seguinte função Lisp:

```
(defun proximo-aleatorio (ultimo-aleatorio)
  (rem (* 16807 ultimo-aleatorio)
        2147483647))
```

Infelizmente, quando testamos a função `aleatorio`, obtemos resultados bizarros: 64454845, 960821323, -553057299, -924795749, 444490781, Sendo o parâmetro `ultimo-aleatorio` inicialmente positivo e sendo o produto e o resto da divisão obtidos usando operandos positivos, o resultado teria necessariamente de ser não-negativo. Uma vez que existem valores negativos na sequência, isso sugere que está a ocorrer o mesmo fenómeno que discutimos na secção 2.16: *overflow*, i.e., o resultado da multiplicação é, por vezes, um número demasiado grande para a capacidade de representação do Auto Lisp. Para resolver este problema, inventou-se

um algoritmo (ver [?]) capaz de computar os números sem exceder a capacidade da máquina, desde que esta seja capaz de representar todos os inteiros no intervalo $[-2^{31}, 2^{31} - 1]$, ou seja, $[-2147483648, 2147483647]$ que é precisamente a gama de valores representáveis em Auto Lisp. O algoritmo baseia-se simplesmente em garantir que todos os valores intermédios do cálculo são sempre representáveis pelo Auto Lisp.

```
(defun proximo-aleatorio (ultimo-aleatorio / teste)
  (setq teste (- (* 16807 (rem ultimo-aleatorio 127773))
                 (* 2836 (/ ultimo-aleatorio 127773))))
  (setq ultimo-aleatorio
        (if (> teste 0)
            teste
            (+ teste 2147483647))))
```

Esta definição tem a vantagem de permitir gerar aleatoriamente todos os inteiros representáveis pela linguagem. Usando esta definição da função, a repetida avaliação da expressão `(random 2)` produz a sequência:

010010100010111010010001100011100110110101101111000011110

e, para a expressão `(random 4)`, produz:

21312020300221331333233301112313001012333020321123122330222

É razoavelmente evidente que qualquer das sequências geradas tem agora um período demasiado grande para se conseguir detectar qualquer padrão de repetição. De ora em diante, será esta a definição de `proximo-aleatorio` que iremos empregar.

5.5.1 Números Aleatórios Fracionários

O processo de geração de números aleatórios que implementámos apenas é capaz de gerar números aleatórios do tipo inteiro. No entanto, é também frequente precisarmos de gerar números aleatórios fracionários, por exemplo, no intervalo $[0, 1]$.

Para combinar estes dois requisitos, é usual no mundo do Lisp que a função `random` receba o limite superior de geração x e o analise para determinar se é inteiro ou real, devolvendo um valor aleatório adequado em cada caso. Para isso, não é preciso mais do que mapear o intervalo de geração de inteiros que, como vimos, é $[0, 2147483647]$, no intervalo $[0, x]$. A implementação é trivial:⁴⁶

```
(defun random (x)
  (if (realp x)
      (* x (/ (aleatorio) 2147483647.0))
      (rem (aleatorio) x)))
```

⁴⁶Esta função usa o reconhecedor universal `realp` que foi definido na secção 2.29.

5.5.2 Números Aleatórios num Intervalo

Se, ao invés de gerarmos números aleatórios no intervalo $[0, x]$, preferirmos gerar números aleatórios no intervalo $[x_0, x_1]$, então basta-nos gerar um número aleatório no intervalo $[0, x_1 - x_0]$ e somar-lhe x_0 . A função `random-[]` implementa esse comportamento:

```
(defun random-[] (x0 x1)
  (+ x0 (random (- x1 x0))))
```

Tal como a função `random`, também `random-[]` produz um valor real no caso de algum dos limites ser real.

Para visualizarmos um exemplo de utilização desta função, vamos recuperar a função `arvore` que modelava uma árvore, cuja definição era:

```
(defun arvore (base comprimento angulo
               alfa-e f-e alfa-d f-d
               / topo)
  (setq topo (+pol base comprimento angulo))
  (ramo base topo)
  (if (< comprimento 2)
      (folha topo)
      (progn
        (arvore topo
                (* comprimento f-e)
                (+ angulo alfa-e)
                alfa-e f-e alfa-d f-d)
        (arvore topo
                (* comprimento f-d)
                (- angulo alfa-d)
                alfa-e f-e alfa-d f-d))))
```

Para incorporarmos alguma aleatoriedade neste modelo de árvore podemos considerar que os ângulos de abertura dos ramos e os factores de redução de comprimento desses ramos não possuem valores constantes ao longo da recursão, antes variando dentro de certos limites. Assim, em vez de nos preocuparmos em ter diferentes aberturas e factores para o ramo esquerdo e direito, teremos simplesmente variações aleatórias em ambos:



Figura 50: Várias árvores geradas com ângulos de abertura aleatórios no intervalo $[\frac{\pi}{2}, \frac{\pi}{16}[$ e factores de redução do comprimento aleatórios no intervalo $[0.6, 0.9[$.

```
(defun arvore (base comprimento angulo
               min-alfa max-alfa min-f max-f
               / topo)
  (setq topo (+pol base comprimento angulo))
  (ramo base topo)
  (if (< comprimento 2)
      (folha topo)
      (progn
        (arvore topo
                (* comprimento (random-[] min-f max-f))
                (+ angulo (random-[] min-alfa max-alfa))
                min-alfa max-alfa min-f max-f)
        (arvore topo
                (* comprimento (random-[] min-f max-f))
                (- angulo (random-[] min-alfa max-alfa))
                min-alfa max-alfa min-f max-f))))
```

Usando esta nova versão, podemos gerar inúmeras árvores semelhantes e, contudo, suficientemente diferentes para parecerem bastante mais naturais. As árvores apresentadas na Figura 50 foram geradas usando exatamente os mesmos parâmetros de crescimento:

```
(arvore (xy 0 0) 20 (/ pi 2) (/ pi 16) (/ pi 4) 0.6 0.9)
(arvore (xy 150 0) 20 (/ pi 2) (/ pi 16) (/ pi 4) 0.6 0.9)
(arvore (xy 300 0) 20 (/ pi 2) (/ pi 16) (/ pi 4) 0.6 0.9)
(arvore (xy 0 150) 20 (/ pi 2) (/ pi 16) (/ pi 4) 0.6 0.9)
(arvore (xy 150 150) 20 (/ pi 2) (/ pi 16) (/ pi 4) 0.6 0.9)
(arvore (xy 300 150) 20 (/ pi 2) (/ pi 16) (/ pi 4) 0.6 0.9)
```

Exercicio 5.1 As árvores produzidas pelas função `arvore` são pouco realista pois são totalmente bidimensionais, com troncos que são simples linhas e folhas que são pequenas circunferências. O cálculo das coordenadas dos ramos e folhas é também bidimensional, assentando sobre coordenadas polares que são dadas pelos parâmetros `comprimento` e `angulo`.

Pretende-se que redefina as funções `ramo`, `folha` e `arvore` de modo a aumentar o realismo das árvores geradas.

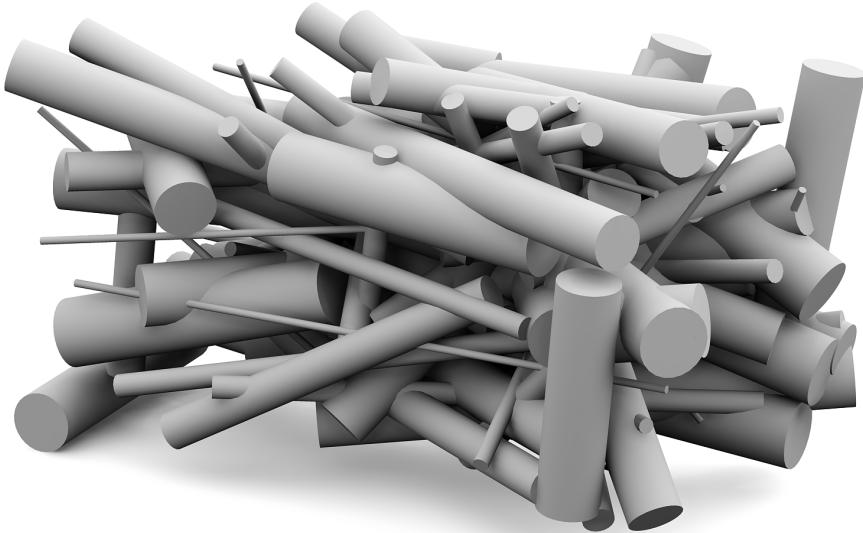
Para simplificar, considere que as folhas podem ser simuladas por pequenas esferas e que os ramos podem ser simulados por troncos de cone, cujo raio da base é 10% do comprimento do tronco e cujo raio do topo é 90% do raio da base.

Para que a geração de árvores passe a ser verdadeiramente tridimensional, redefina também a função `arvore` de modo a que o topo de cada ramo seja um ponto em coordenadas esféricas escolhidas aleatoriamente a partir da base do ramo. Isto implica que a função `árvore`, ao invés de ter dois parâmetros para o comprimento e o ângulo das coordenadas polares, precisará de ter três, para o comprimento, a longitude e a co-latitude. Do mesmo modo, ao invés de receber os quatro limites para a geração de comprimentos e ângulos aleatórios, precisará de seis limites para os três parâmetros.

Experimente diferentes argumentos para a sua redefinição da função `arvore` de modo a gerar imagens como a seguinte:

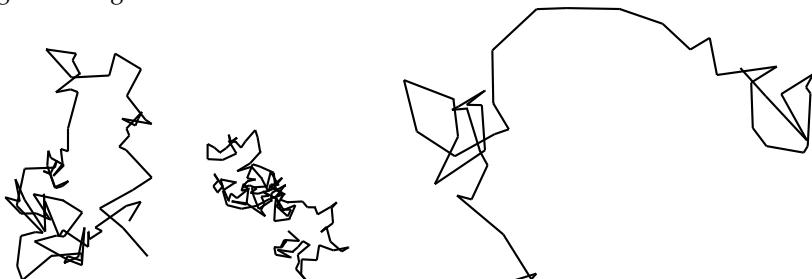


Exercicio 5.2 Defina uma função denominada `cilindros-aleatorios` que recebe como argumento um número de cilindros n e que gera n cilindros colocados em posições aleatórias, com raios aleatórios e comprimentos aleatórios, tal como se apresenta na seguinte imagem:



Exercicio 5.3 Uma *caminhada aleatória* é uma formalização do movimento de uma partícula que está sujeita a impulsos de intensidade aleatória vindos de direções aleatórias. É este fenômeno que acontece, por exemplo, a um grão de pólen caído na água: à medida que as moléculas da água vão chocando com ele, o grão vai se movendo aleatoriamente.

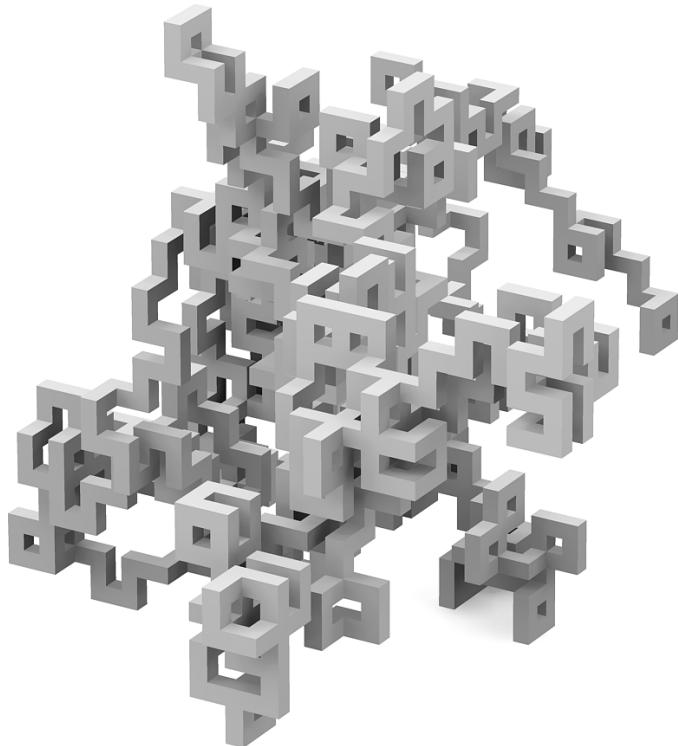
A seguinte imagem mostra três caminhadas aleatórias:



Considere uma modelação da *caminhada aleatória* num plano bi-dimensional. Defina a função `caminhada-aleatoria` que recebe o ponto inicial P da caminhada, a distância máxima d que a partícula pode percorrer quando é impulsionada e o número n de impulsos sucessivos que a partícula vai receber. Note que, a cada impulso, a partícula desloca-se, numa direção aleatória, uma distância aleatória entre zero e a distância máxima. A sua função deverá simular a caminhada aleatória correspondente, tal como se apresenta na figura anterior que foi criada por três execuções diferentes desta função. Da esquerda para a direita, foram usados os parâmetros $d = 5, n = 100, d = 2, n = 200$, e $d = 8, n = 50$,

Exercicio 5.4 Defina uma função `cara-ou-coroa viciada` de tal forma que, embora a função produza aleatoriamente a string "cara" ou a string "coroa", a "cara" sai, em média, apenas 30% das vezes em que a função é invocada.

Exercicio 5.5 Defina uma função denominada `blocos-conexos` capaz de construir um aglomerado de blocos conexos, i.e., onde os blocos estão fisicamente ligados entre si, tal como se vê na seguinte imagem:



Note que os blocos adjacentes possuem sempre orientações ortogonais.

5.6 Planeamento Urbano

As cidades constituem um bom exemplo de combinação entre estruturação e aleatoriedade. Embora muitas das cidades mais antigas aparentem ter uma forma caótica resultante do seu desenvolvimento não ter sido planeado, a verdade é que desde muito cedo se sentiu a necessidade de estruturar a cidade de modo a facilitar a sua utilização e o seu crescimento, sendo conhecidos exemplos de cidades planeadas desde 2600 antes de Cristo.

Embora haja bastantes variações, as duas formas mais usuais de planejar uma cidade são através do plano ortogonal ou através do plano circular. No plano ortogonal, as avenidas são rectas e fazem ângulos rectos entre si. No plano circular as avenidas principais convergem numa praça central e as avenidas secundárias desenvolvem-se em círculos concêntricos em torno deste ponto, acompanhando o crescimento da cidade. A Figura 51 apresenta um bom exemplo de uma cidade essencialmente desenvolvida a partir de planos ortogonais. Nesta secção vamos explorar a aleatoriedade para produzir variações em torno destes planos.

Num plano ortogonal, a cidade organiza-se em termos de quarteirões, em que cada quarteirão assume uma forma quadrada ou rectangular e pode conter vários edifícios. Para simplificar, vamos assumir que cada quarteirão será de forma quadrada e irá conter um único edifício. Cada



Figura 51: Vista aérea da cidade de New York nos Estados Unidos. Fotografia de Art Siegel.

edifício terá uma largura determinada pela largura do quarteirão e uma altura máxima imposta. Os edifícios serão separados uns dos outros por ruas com uma largura fixa.

Para estruturarmos a função que constrói a cidade em malha ortogonal, vamos decompor o processo na construção de sucessivas ruas. A construção de cada rua será então decomposta na construção dos sucessivos prédios. Assim, teremos de parametrizar a função com o número de ruas a construir na cidade e com o número de prédios por rua. Para além disso iremos necessitar de saber as coordenadas do ponto p onde se começa a construir a cidade, a largura e a altura de cada prédio, respectivamente, l_predio e a_predio e, finalmente, a largura da rua l_rua . A função irá construir uma faixa de prédios seguido de uma rua e, recursivamente, construirá as restantes faixas de prédios e ruas no ponto correspondente à faixa seguinte. Para simplificar, vamos assumir que as ruas estarão alinhadas com os eixos x e y , pelo que cada nova rua corresponde a um deslocamento ao longo do eixo y e cada novo prédio corresponde a um deslocamento ao longo do eixo x . Assim, temos:

```
(defun malha-predios (p ruas predios l-predio a-predio l-rua)
  (if (= ruas 0)
      nil
      (progn
        (rua-predios p predios l-predio a-predio l-rua)
        (malha-predios (+y p (+ l-predio l-rua))
                      (1- ruas)
                      predios
                      l-predio
                      a-predio
                      l-rua))))
```

Para a construção das ruas com prédios, o raciocínio é idêntico: colocamos um prédio nas coordenadas correctas e, recursivamente, colocamos os restantes prédios após o deslocamento correspondente. A seguinte função implementa este processo:

```
(defun rua-predios (p predios l-predio a-predio l-rua)
  (if (= predios 0)
      nil
      (progn
        (predio p l-predio a-predio)
        (rua-predios (+x p (+ l-predio l-rua))
                     (1- predios)
                     l-predio
                     a-predio
                     l-rua))))
```

Finalmente, precisamos de definir a função que cria um prédio. Para simplificar, vamos modelá-lo como um paralelipípedo:

```
(defun predio (p l-predio a-predio)
  (command "_.box" p (+xyz p l-predio l-predio a-predio)))
```

Com estas três funções já podemos experimentar a construção de uma nova urbanização. Por exemplo, a seguinte expressão cria um arranjo de dez ruas com dez prédios por rua:

```
(malha-predios (xyz 0 0 0) 10 10 100 400 40)
```

O resultado está representado na Figura 52.

Como é óbvio pela Figura 52, a urbanização produzida é pouco elegante pois falta-lhe alguma da aleatoriedade que caracteriza as cidades.

Para incorporarmos essa aleatoriedade vamos começar por considerar que a altura dos prédios pode variar aleatoriamente entre a altura máxima dada e um décimo dessa altura. Para isso, redefinimos a função `predio`:

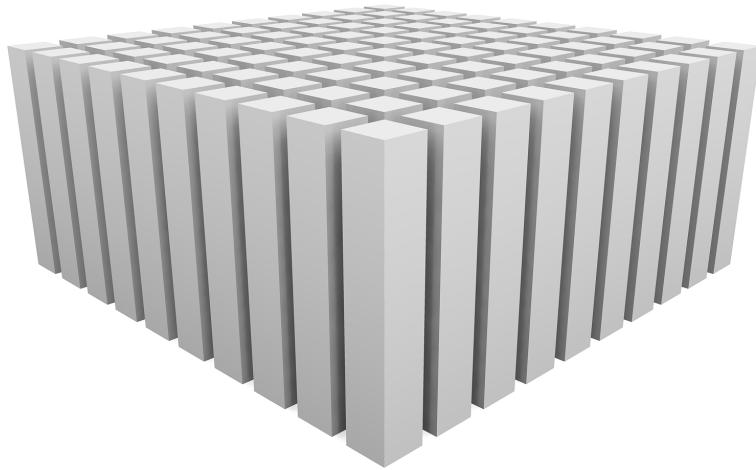


Figura 52: Uma urbanização em malha ortogonal com cem edifícios todos da mesma altura.

```
(defun predio (p l-predio a-predio)
  (command "_.box"
    p
    (+xyz p
      l-predio
      l-predio
      (* (random-[] 0.1 1.0) a-predio))))
```

Usando exactamente os mesmos parâmetros anteriores em duas avaliações sucessivas da mesma expressão, conseguimos agora construir as urbanizações esteticamente mais apelativas representadas nas Figuras 53 e 54.

Exercicio 5.6 As urbanizações produzidas pelas funções anteriores não apresentam variabilidade suficiente, pois os edifícios têm todos a mesma forma. Para melhorar a qualidade estética da urbanização pretende-se empregar diferentes funções para a construção de diferentes tipos de edifícios: a função `predio0` deverá construir um paralelipípedo de altura aleatória tal como anteriormente e a função `predio1` deverá construir uma torre cilíndrica de altura aleatória e contida no espaço de um quarteirão. Defina estas duas funções.

Exercicio 5.7 Pretende-se que use as duas funções `predio0` e `predio1` definidas no exercício anterior para a redefinição da função `predio` de modo a que esta, aleatoriamente, construa prédios diferentes. A urbanização resultante deverá ser constituída aproximadamente por 20% de torres circulares e 80% de prédios rectangulares, tal como é exemplificado na imagem seguinte:

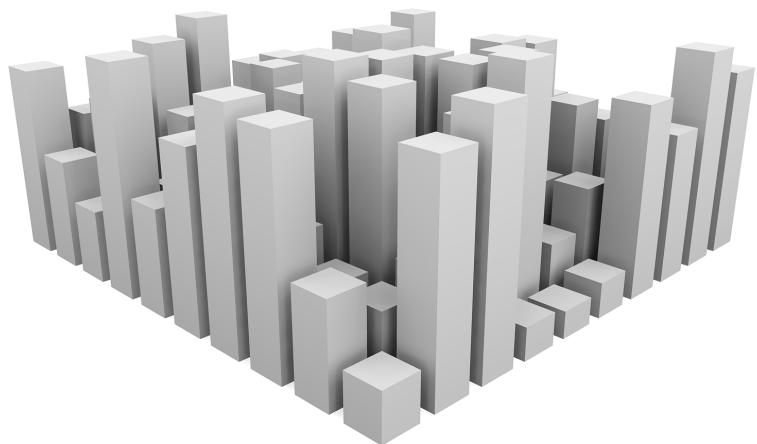


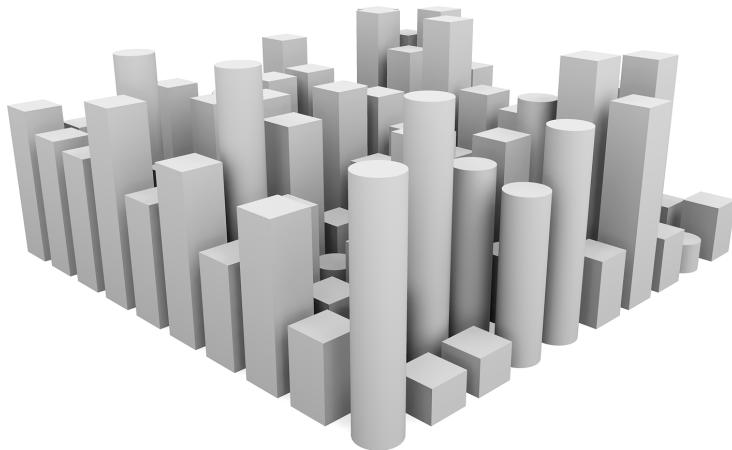
Figura 53: Uma urbanização em malha ortogonal com cem edifícios com alturas aleatórias.



Figura 54: Uma urbanização em malha ortogonal com cem edifícios com alturas aleatórias.



Figura 55: Vista de alguns prédios de Manhattan. Fotografia de James K. Poole.

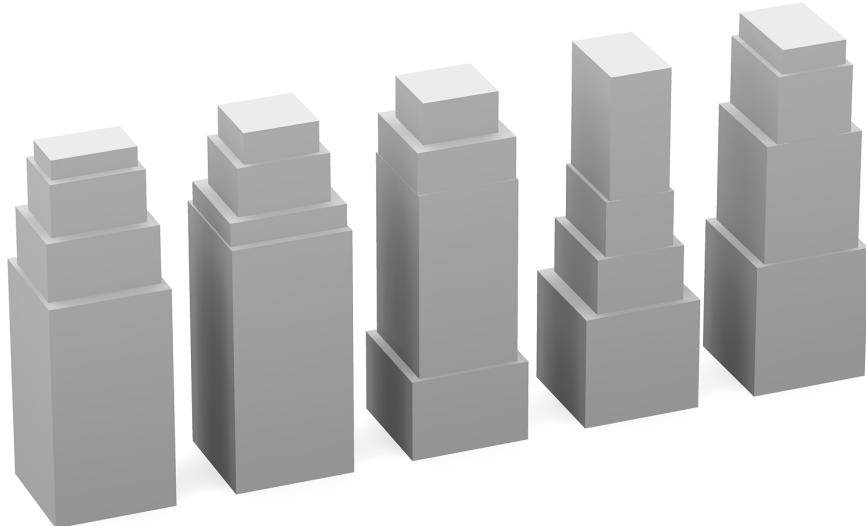


Exercício 5.8 As cidades criadas nos exercícios anteriores apenas permitem dois tipos de edifícios: torres paralelipípedicas ou torres cilíndricas. Contudo, quando observamos uma cidade real como a apresentada na Figura 55, verificamos que existem prédios com muitas outras formas pelo que, para aumentarmos orealismo das nossas simulações, teremos de implementar um vasto número de funções, cada uma capaz de construir um edifício diferente.

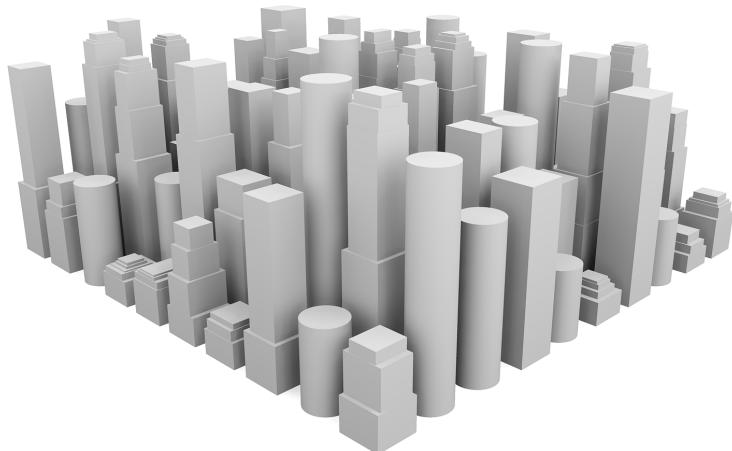
Felizmente, uma observação atenta da Figura 55 mostra que, na verdade, muitos dos prédios seguem um padrão e podem ser modelado por paralelipípedos sobrepostos com dimensões aleatórias mas sempre sucessivamente mais pequenos em função da altura, algo que podemos facilmente implementar com uma única função.

Considere que este modelo de edifício é parametrizado pelo número de “blocos” sobrepostos pretendidos, pelas coordenadas do primeiro bloco e pelo comprimento, largura

e altura do edifício. O bloco de base tem exactamente o comprimento e largura especificados mas a sua altura deverá estar entre 20% e 80% da altura total do edifício. Os blocos seguintes estão centrados sobre o bloco imediatamente abaixo e possuem um comprimento e largura que estão entre 70% e 100% dos parâmetros correspondentes no bloco imediatamente abaixo. A altura destes blocos deverá ser entre 20% e 80% da altura restante do edifício. A imagem seguinte mostra alguns exemplos deste modelo de edifícios:



Com base nesta especificação, defina a função `predio-blocos` e use-a para redefinir a função `predio0` de modo a que sejam gerados prédios com um número aleatório de blocos sobrepostos. Com esta redefinição, a função `malha-predios` deverá ser capaz de gerar urbanizações semelhantes à imagem seguinte, em que se empregou para cada prédio um número de blocos entre 1 e 6:



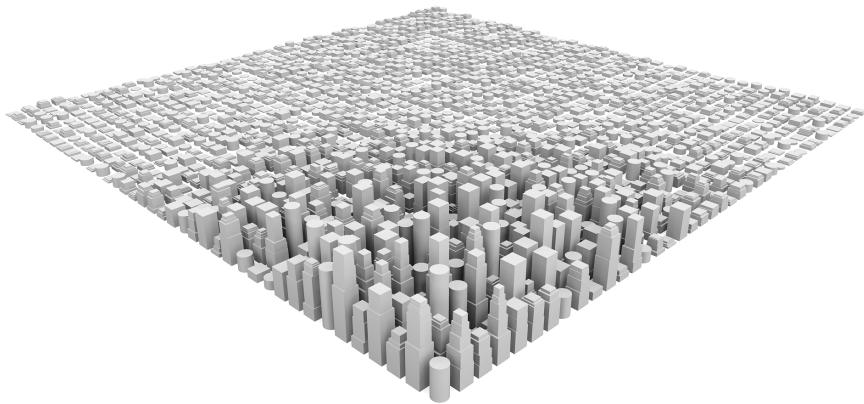
Exercício 5.9 Em geral, as cidades possuem um núcleo de edifícios relativamente altos no centro e, à medida que nos afastamos para a periferia, a altura tende a diminuir, sendo este fenômeno perfeitamente visível na Figura 51.

A variação da altura dos edifícios pode ser modelada por diversas funções matemáticas mas, neste exercício, pretende-se que empregue uma distribuição Gaussiana bi-dimensional

dada por

$$f(x, y) = e^{-\left(\frac{(x-x_0)^2}{\sigma_x^2} + \frac{(y-y_0)^2}{\sigma_y^2}\right)}$$

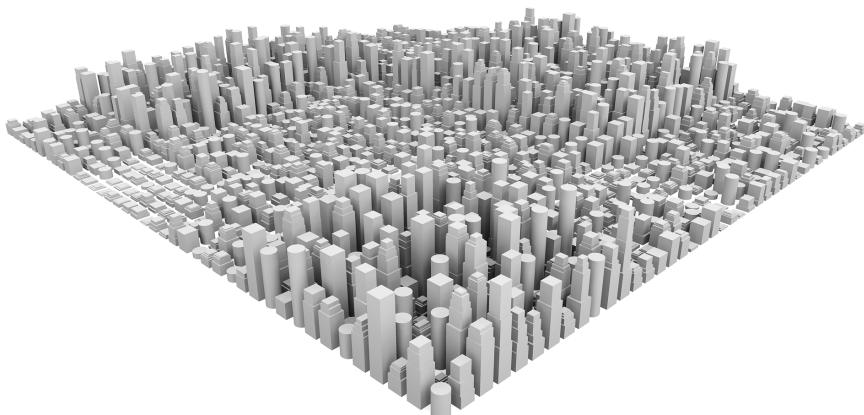
em que f é o factor de ponderação da altura, (x_0, y_0) são as coordenadas do ponto mais alto da superfície Gaussiana e σ_x e σ_y são os factores que afectam o alargamento bi-dimensional dessa superfície. Para simplificar, assuma que o centro da cidade fica nas coordenadas $(0, 0)$ e que $\sigma_x = \sigma_y = 25l$, sendo l a largura do edifício. A imagem seguinte mostra o aspecto de uma destas urbanizações:



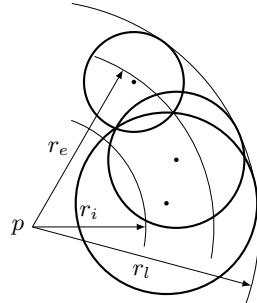
Incorpore esta distribuição no processo de construção de cidades para produzir cidades mais realistas.

Exercicio 5.10 Frequentemente, as cidades possuem mais do que um núcleo de edifícios altos. Estes núcleos encontram-se separados uns dos outros por zonas de edifícios menos altos. Tal como no exercício anterior, cada núcleo de edifícios pode ser modelado por uma distribuição Gaussiana. Admitindo a independência dos vários núcleos, a altura dos edifícios pode ser modelada por distribuições Gaussianas sobrepostas, i.e., em cada ponto, a altura do edifício será o máximo das alturas das distribuições Gaussianas dos vários núcleos.

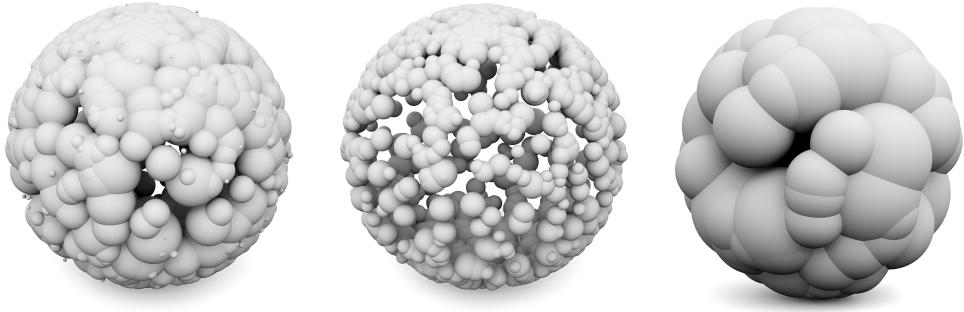
Utilize a abordagem anterior para exemplificar a modelação de uma cidade com três núcleos de “arranha-céus”, à semelhança do que se apresenta na imagem seguinte:



Exercicio 5.11 Pretende-se criar um conjunto de n esferas tangentes a uma esfera virtual de centro em p e raio limite r_l . As esferas possuem centros a uma distância de p que é um valor aleatório compreendido entre um raio interior r_i e um raio exterior r_e , tal como se pode visualizar no seguinte esquema:



Defina uma função denominada `esferas-na-esfera` que, a partir do centro p , dos raios r_i , r_e , e r_l e ainda do número n , cria este conjunto de esferas, permitindo produzir imagens como as que se seguem:



6 Listas

Vimos anteriormente que uma lista é uma sequência de elementos que podemos construir usando a função `list`. Vimos também que podemos aceder a cada um destes elementos utilizando a função `nth` que, dada a posição de um elemento e a lista que o contém, devolve esse elemento. Eis um exemplo do uso destas operações:

```
_\$ (setq amigos (list "Filipa" "Pedro" "Carlos" "Maria"))
("Filipa" "Pedro" "Carlos" "Maria")
_\$ (nth 0 amigos)
"Filipa"
_\$ (nth 3 amigos)
"Maria"
```

No passado, usámos as listas para implementar coordenadas tridimensionais, i.e., agrupamentos de três números representando posições no espaço. Agora, vamos usá-las para agrupar um número arbitrário de elementos. Para isso, temos de adoptar uma visão diferente das listas e temos de

introduzir novas funções que nos permitem trabalhar mais facilmente com listas. Comecemos por introduzir uma função—`cons`, abreviatura da palavra *construct*—que nos permite criar uma lista maior a partir de um novo elemento e de uma lista já existente.

```
_\$ (setq novos-amigos (cons "Bernardo" amigos))
("Bernardo" "Filipa" "Pedro" "Carlos" "Maria")
_\$ amigos
("Filipa" "Pedro" "Carlos" "Maria")
```

Como se podemos ver, a função `cons` junta um elemento a uma lista, produzindo uma nova lista que tem mais esse elemento à cabeça da lista. Notemos também que a lista original fica inalterada pela função `cons`.

Para “decompormos” listas, o Lisp fornece um par de funções—`car` e `cdr`—que podem ser vistas como o inverso da função `cons`. Enquanto a função `cons` junta um elemento a uma lista, a função `car` indica qual o elemento que se juntou e a função `cdr` indica qual a lista a que ele se juntou. Eis um exemplo:

```
_\$ (car novos-amigos)
"Bernardo"
_\$ (cdr novos-amigos)
("Filipa" "Pedro" "Carlos" "Maria")
```

Dito de outra forma, podemos afirmar que a função `car` devolve o *primeiro* elemento da lista e a função `cdr` devolve o *resto* da lista, i.e., a lista após o primeiro elemento. Os nomes `car` e `cdr` têm raízes históricas e, embora possa não parecer, são relativamente fáceis de decorar. Uma mnemónica que pode ajudar é pensar que o “cAr” obtém o que vem “Antes” e o “cDr” obtém o que vem “Depois.”

Naturalmente, as funções `car` e `cdr` podem ser encadeadas:

```
_\$ (car (cdr novos-amigos))
"Filipa"
_\$ (cdr (cdr (cdr novos-amigos)))
("Carlos" "Maria")
_\$ (car (cdr (cdr (cdr novos-amigos))))
"Carlos"
```

Dado que aquele género de expressões é muito utilizado em Lisp, foram compostas as várias combinações, e criaram-se funções do tipo `(caddr exp)`, que correspondem a `(car (cdr (cdr exp)))`. O nome da função indica quais as operações a realizar. Um “a” representa um `car` e um “d” representa um `cdr`.

A partir dos exemplos anteriores, é lógico deduzir que a função `list` faz algo de muito semelhante a uma combinação de `cons` e, na verdade, assim é:

```

_§ (cons 1 (list 2 3 4))
(1 2 3 4)
_§ (cons 1 (cons 2 (list 3 4)))
(1 2 3 4)
_§ (cons 1 (cons 2 (cons 3 (list 4))))
(1 2 3 4)
_§ (cons 1 (cons 2 (cons 3 (cons 4 (list))))))
(1 2 3 4)

```

Anteriormente, vimos também que o Lisp considera que o símbolo `nil` representa uma lista sem elementos e que o predicado `null` identifica estas listas. De facto, quando tentamos obter o resto de uma lista com um único elemento, obtemos `nil`, tal como quando tentamos construir uma lista sem elementos:

```

_§ (cdr (list 1))
nil
_§ (list)
nil
_§ (null (list))
T
_§ (null amigos)
nil

```

Embora o Auto Lisp use o símbolo `nil` para representar uma lista vazia, sabemos que este símbolo também é usado para representar a falsidade, tal como a última expressão demonstra. Embora internamente seja exatamente a mesma coisa, para distinguir os dois usos o Auto Lisp permite uma notação diferente para o símbolo `nil`: `()`. Isso é visível na seguinte interacção:

```

_§ ()
nil
_§ (null ())
T

```

6.1 Pares

Vimos que a função `cons` permite juntar um elemento a uma lista mas, na verdade, ela faz algo bastante mais fundamental. Esta função aceita quaisquer duas entidades como argumentos e produz um par com essas duas entidades, i.e., um valor que representa um aglomerado dessas duas entidades. Tradicionalmente, é usual dizer que o par é um *cons*. As funções `car` e `cdr` são meramente os *selectores* que devolvem o primeiro e o segundo elemento do par.

Eis um exemplo da criação de um par de números:

```
_\$ (cons 1 2)
(1 . 2)
_\$ (car (cons 1 2))
1
_\$ (cdr (cons 1 2))
2
```

Notemos que quando o Lisp pretende escrever o resultado da criação de um par, ele começa por escrever um abrir parênteses, depois escreve o primeiro elemento do par, depois escreve um ponto para separar, depois escreve o segundo elemento do par e, finalmente, escreve um fechar parênteses. Esta notação denomina-se “par com ponto” ou, no original, *dotted pair*. No entanto, quando o segundo elemento do par é também um par, o Lisp emprega uma notação mais simples que omite o ponto e um par de parêntesis:

```
_\$ (cons 1 (cons 2 3))
(1 2 . 3)
```

No exemplo anterior, se o Lisp seguisse apenas a notação de par com ponto, teria de escrever $(1 . (2 . 3))$.

Finalmente, quando o segundo elemento do par é uma lista vazia, o Lisp limita-se a escrever o primeiro elemento entre um par de parêntesis.

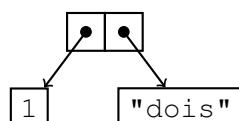
```
_\$ (cons 1 ())
(1)
```

É a combinação destas duas regras que nos permite visualizar sequência de pares terminada com uma lista vazia como sendo uma lista:

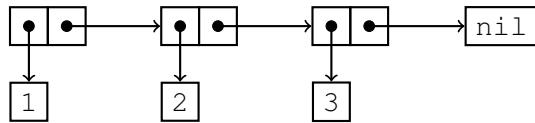
```
_\$ (cons 1 (cons 2 (cons 3 ())))
(1 2 3)
```

6.2 Representação Gráfica de Pares

Dissemos anteriormente que, ao fazermos $(\text{cons } \alpha \beta)$, estamos a criar um par com os elementos α e β . Em termos da memória do Lisp, isto implica que será reservada memória para conter esse par, que representamos graficamente com uma caixa dividida em duas metades, a mais à esquerda a apontar para o primeiro argumento do `cons` (o *car*) e a mais à direita a apontar para o segundo argumento do `cons` (o *cdr*). Por exemplo, a expressão `(cons 1 "dois")` pode-se representar nesta notação gráfica, denominada *de caixa e ponteiro*, tal como se segue:



Obviamente, um *cons* pode apontar para outro *cons*. É precisamente isso que acontece quando criamos listas. Como vimos, uma lista não é mais do que um *cons* cujo *cdr* aponta para outra lista (vazia ou não). Por exemplo, a lista criada pela expressão `(list 1 2 3)` é absolutamente equivalente à lista `(cons 1 (cons 2 (cons 3 nil)))`. A sua representação gráfica é:



Por aqui se vê que, do ponto de vista do Lisp, uma lista não é mais do que um particular arranjo de pares em que o segundo elemento de cada par é, ou outro par, ou uma lista vazia.

Assim, a lista vazia `()` é o ponto de partida para a construção de qualquer lista, tal como podemos ver na seguinte sequência de expressões:

```

_ $ (cons 3 ())
(3)
_ $ (cons 2 (cons 3 ()))
(2 3)
_ $ (cons 1 (cons 2 (cons 3 ())))
(1 2 3)
  
```

Repare-se, nas expressões anteriores, que o segundo argumento da função *cons* é, ou uma lista vazia, ou uma lista contendo alguns elementos. Nestes casos, o resultado da invocação da função *cons* será sempre uma lista.

6.3 Tipos Recursivos

Obviamente, qualquer que seja a lista que imaginemos, é sempre possível construí-la usando apenas invocações da função *cons* e a lista vazia `()`. De facto, temos que `(list)` é equivalente a `()` e que `(list e1 e2 ... en)` é equivalente a `(cons e1 (list e2 ... en))`.

Consequentemente, podemos dizer que uma lista é sempre:

- ou a lista vazia `()`,
- ou o *cons* de um elemento a uma lista.

Notemos que há um detalhe subtil na definição de lista que acabámos de apresentar: ela é recursiva! Para o confirmar basta reparar que estamos a definir uma *lista* como o *cons* de um elemento a uma *lista*, i.e., usamos o termo que queremos definir na própria definição. Como já sabemos de qualquer definição recursiva, é necessário termos um caso de paragem que, no caso das listas, é a lista vazia.

Quando um tipo de dados é definido de forma recursiva dizemos que temos um *tipo recursivo*. As listas são, portanto, um tipo recursivo. Num tipo recursivo tem de existir sempre um elemento primordial, a partir do qual se criam os restantes elementos. A esse elemento primordial chama-se *elemento primitivo*. No caso das listas, o elemento primitivo é, obviamente, a lista vazia.

Tal como acontecia com os outros tipos de dados, as várias operações que permitem manipular listas podem classificar-se em construtores, selector e reconhecedores. É fácil ver que () e cons são construtores, car e cdr são selectores e, finalmente, null é o reconhecedor. Usando estas operações podemos agora definir inúmeras outras operações sobre listas.

6.4 Recursão em Listas

Uma das propriedades interessantes dos tipos recursivos é que todas as operações que processam elementos do tipo tendem a ser implementadas por funções recursivas. Por exemplo, se quisermos definir uma função que nos diz quantos elementos existem numa lista, temos de pensar recursivamente:

- Se a lista é vazia, então o número de elementos é, obviamente, zero.
- Caso contrário, o número de elementos será um mais o número de elementos do resto da lista.

Para nos assegurarmos da correcção do nosso raciocínio temos de garantir que se verificam os pressupostos a que todas as definições recursivas têm de obedecer, nomeadamente:

- Que há uma redução da complexidade do problema a cada invocação recursiva: de facto, a cada invocação recursiva a lista usada é mais pequena.
- Que há um caso mais simples de todos onde existe uma resposta imediata: quando a lista é vazia, a resposta é zero.
- Que existe uma equivalência entre o problema original e o uso do resultado da recursão: é verdade que o número de elementos de uma lista é o mesmo que somar 1 ao número de elementos dessa lista sem o primeiro elemento.

A verificação destes pressupostos é suficiente para garantirmos que a função está correcta.

Traduzindo este raciocínio para Lisp, obtemos:

```
(defun numero-elementos (lista)
  (if (null lista)
    0
    (+ 1 (numero-elementos (cdr lista)))))
```

Experimentando, temos:

```
_\$ (numero-elementos (list 1 2 3 4))
4
_\$ (numero-elementos (list))
0
```

É importante percebermos que a presença de sublistas não altera o número de elementos de uma lista. De facto, os elementos das sublistas *não* são considerados elementos da lista. Por exemplo:

```
_\$ (list (list 1 2) (list 3 4 5 6))
((1 2) (3 4 5 6))
_\$ (numero-elementos (list (list 1 2) (list 3 4 5 6)))
2
```

Como se vê no exemplo anterior, a lista apenas contém dois elementos, embora cada um deles seja uma lista com mais elementos.

Na verdade, a função `numero-elementos` já existe em Lisp com o nome `length`.

Uma outra operação útil é aquela que nos permite obter o n -ésimo elemento de uma lista. Vimos que essa função já existe e se denomina `nth` mas agora vamos definir a nossa própria versão. É pragmática usual assumir-se que, para $n = 0$, se deve obter o primeiro elemento da lista. Para definir esta função devemos, mais uma vez, pensar de forma recursiva:

- Se n é zero, devolvemos o primeiro elemento da lista.
- Caso contrário, devolver o n -ésimo elemento da lista é o mesmo que devolver o $(n - 1)$ -ésimo elemento do resto da lista.

Novamente verificamos que há uma redução do problema, que a redução é equivalente ao problema original e que o problema se vai aproximando do caso básico. Em Lisp, temos:

```
(defun n-esimo (n lista)
  (if (= n 0)
    (car lista)
    (n-esimo (- n 1) (cdr lista))))
```

```
_\$ (n-esimo 2 (list 0 1 2 3 4))
2
```

É frequentemente necessário termos de concatenar listas. Para isso, podemos imaginar uma função `concatena` que, dadas duas listas, devolve uma lista contento, pela mesma ordem, os elementos das duas listas. Por exemplo:

```
_\$ (concatena (list 1 2 3) (list 4 5 6 7))  
(1 2 3 4 5 6 7)
```

Como sempre, a recursão ajuda a resolver o problema. Começemos por pensar na simplificação do problema, i.e., em transformar o problema original (`(concatena (list 1 2 3) (list 4 5 6 7))`) num problema ligeiramente mais simples.

Uma vez que há duas listas como parâmetros, podemos simplificar o problema fazendo uma redução numa delas, i.e., podemos considerar uma redução na primeira lista

```
(concatena (list 2 3) (list 4 5 6 7))
```

cujo resultado é `(2 3 4 5 6 7)`, ou, alternativamente, uma redução na segunda lista

```
(concatena (list 1 2 3) (list 5 6 7))
```

cujo resultado é `(1 2 3 5 6 7)`.

Para além da simplificação do problema, é também necessário garantir que conseguimos arranjar uma maneira de fazer com que a simplificação do problema seja equivalente ao problema original. Para a primeira alternativa isso é trivial, basta inserir o número 1 à cabeça da lista, i.e., fazer

```
(cons 1 (concatena (list 2 3) (list 4 5 6 7)))
```

Para a segunda alternativa isso é bastante mais difícil pois teríamos de inserir o número 4 algures no meio da lista resultante.

Assim sendo, não deverão restar muitas dúvidas que a função deverá ter a forma:

```
(defun concatena (l1 l2)  
  (if ???  
      ???  
      (cons (car l1)  
            (concatena (cdr l1) l2))))
```

Falta agora tratar o caso básico. Para isso, basta pensar que se estamos a reduzir a primeira lista a cada invocação recursiva então chegará um momento em que a primeira lista fica vazia. O que é então a concatenação de uma lista vazia a uma outra lista qualquer? Obviamente, é a outra lista.

Assim, podemos finalmente definir completamente a função:

```
(defun concatena (l1 l2)  
  (if (null l1)  
      l2  
      (cons (car l1)  
            (concatena (cdr l1) l2))))
```

Na verdade, esta função já existe em Lisp com o nome `append` e tem a vantagem adicional de receber qualquer número de argumentos, por exemplo:

```
_\$ (append (list 0) (list 1 2 3) (list 4 5) (list 6 7 8 9))
(0 1 2 3 4 5 6 7 8 9)
```

A operação inversa da concatenação de listas é a separação de listas. Para ser mais útil, vamos considerar uma função que obtém uma sublista a partir de uma lista dada. Essa sublista inclui todos os elementos entre dois índices dados. Usualmente, convenciona-se que a sublista inclui o elemento com o primeiro índice e *exclui* o elemento com o segundo índice. Como exemplo, temos:

```
_\$ (sublista (list 0 1 2 3 4 5 6) 2 4)
(2 3)
```

Para definirmos esta função, podemos recursivamente simplificar o problema da obtenção da sublista de $(e_0 \ e_1 \ \dots \ e_n)$ entre os índices i e j para passar a ser a obtenção da sublista de $(e_1 \ \dots \ e_n)$ entre os índices $i - 1$ e $j - 1$. Quando i for zero, então passamos a transformar o problema da obtenção da sublista de $(e_0 \ e_1 \ \dots \ e_n)$ entre os índices 0 e j para passar a ser o `cons` de e_0 com a sublista de $(e_1 \ \dots \ e_n)$ entre os índices 0 e $j - 1$. Traduzindo este algoritmo para Lisp, temos:

```
(defun sublista (lista inicio fim)
  (cond ((> inicio 0)
          (sublista (cdr lista) (1- inicio) (1- fim)))
        ((> fim 0)
         (cons (car lista)
               (sublista (cdr lista) inicio (1- fim))))
        (t
         ()))))
```

Exercício 6.1 Defina a função `elimina-n-esimo` que recebe um número n e uma lista, e elimina o n -ésimo elemento da lista. Note que o primeiro elemento da lista corresponde a n igual a zero.

Por exemplo:

```
_\$ (elimina-n-esimo 2 (list 0 1 2 3 4 5))
(0 1 3 4 5)
```

Exercício 6.2 Escreva uma função `muda-n-esimo` que recebe um número n , uma lista e um elemento, e substitui o n -ésimo elemento da lista por aquele elemento. Note que o primeiro elemento da lista corresponde a n igual a zero.

Por exemplo:

```
_\$ (muda-n-esimo 2 (list 0 1 2 3 4 5) 9)
(0 1 9 3 4 5)
_\$ (muda-n-esimo 2 (list "Vou" "para" "Coimbra") "Lisboa")
("Vou" "para" "Lisboa")
```

Exercício 6.3 Escreva uma função que dada uma lista de elementos devolve um elemento dessa lista, escolhido aleatoriamente.

Exercício 6.4 Defina a função `um-de-cada` que, dada uma lista de listas, constrói uma lista contendo, por ordem, um elemento aleatório de cada uma das listas. Por exemplo:

```

_§ (um-de-cada (list (list 0 1 2) (list 3 4) (list 5 6 7 8)))
(2 4 7)
_§ (um-de-cada (list (list 0 1 2) (list 3 4) (list 5 6 7 8)))
(1 3 5)
_§ (um-de-cada (list (list 0 1 2) (list 3 4) (list 5 6 7 8)))
(1 4 8)

```

Exercicio 6.5 Defina a função `elementos-aleatorios` que, dado um número n e uma lista de elementos devolve n elementos dessa lista escolhidos aleatoriamente. Por exemplo:

```

_§ (elementos-aleatorios 3 (list 0 1 2 3 4 5 6))
(1 5 2)

```

Exercicio 6.6 Redefina a função `elementos-aleatorios` de modo a que o resultado respeite a ordem dos elementos na lista de onde foram escolhidos.

Exercicio 6.7 Como vimos, a função `cons` acrescenta um novo primeiro elemento a uma lista, a função `car` devolve o primeiro elemento de uma lista e a função `cdr` devolve a lista sem o primeiro elemento.

Escreva as funções “inversas” do `cons`, `car` e `cdr`, designadas `snoc`, `rac` e `rdc` que, ao invés de operarem com o primeiro elemento, operam com o último. O `snoc` recebe um elemento e uma lista e junta o elemento ao fim da lista. O `rac` devolve o último elemento da lista. O `rdc` devolve uma lista com todos os elementos menos o último.

Exercicio 6.8 Defina uma função `inverte` que recebe uma lista e devolve outra lista que possui os mesmos elementos da primeira só que por ordem inversa.

6.5 Predicados sobre Listas

Para testarmos se uma determinada entidade é uma lista, podemos empregar o predicado `listp`. Este predicado é verdade para qualquer lista (incluindo a lista vazia) e falso para tudo o resto:

```

_§ (listp (list 1 2))
T
_§ (listp 1)
nil
_§ (listp (list))
T
_§ (listp (cons 1 2))
T

```

Note-se, no último exemplo, que o reconhecedor universal `listp` também considera como lista um mero par de elementos. Na realidade, este reconhecedor devolve verdade sempre que o seu argumento é um *dotted pair* ou o `nil`.

Uma vez que as funções `car` e `cdr` nos permitem inspecionar *dotted pairs*, a sua aplicação consecutiva permite-nos chegar até aos constituintes últimos dos pares, que serão números, *strings*, etc. Estes constituintes últimos, por já não serem mais decomponíveis por intermédio das funções `car` e `cdr`, denominam-se *átomicos*: em Lisp, tudo o que não for um *dotted pair* é considerado um *átomo*.

Para facilitar a identificação das entidades atómicas, a linguagem Lisp disponibiliza uma função denominada `atom` que só é verdadeira para as entidades atómicas:

```
_\$ (atom 1)
T
_\$ (atom "dois")
T
_\$ (atom (cons 1 "dois"))
nil
```

Uma vez que as listas são constituídas por pares, é lógico que não sejam consideradas átomos. Há, no entanto, uma excepção: a lista vazia é considerada simultaneamente uma lista e um átomo. Isso é visível na seguinte interacção:

```
_\$ (atom (list 1 "dois" 3.0))
nil
_\$ (atom (list))
T
```

Exercicio 6.9 Sendo as listas uma estrutura de dados tão flexível, é muito frequente usarmos listas contendo outras listas que, por sua vez, podem conter outras listas, até ao nível de profundidade que se queira. Por exemplo, consideremos a seguinte lista de posições:

```
_\$ (list (list 1 2) (list 3 4) (list 5 6))
((1 2) (3 4) (5 6))
```

ou a seguinte lista de listas de posições:

```
_\$ (list (list (list 1 2) (list 3 4) (list 5 6))
          (list (list 7 8) (list 9 0)))
(((1 2) (3 4) (5 6)) ((7 8) (9 0)))
```

Escreva uma função denominada `alisa` que recebe uma lista (possivelmente com sublistas) como argumento e devolve outra lista com todos os átomos da primeira e pela mesma ordem, i.e.

```
_\$ (alisa '(1 2 (3 4 (5 6)) 7))
(1 2 3 4 5 6 7)
```

Exercicio 6.10 Escreva uma função que calcula o número de átomos que uma lista (possivelmente com sublistas) tem.

Exercicio 6.11 A concatenação de *strings* pode ser realizada pela função `strcat`. Defina a função `concatena-strings` que, dada uma lista de *strings*, devolve uma só *string* com a concatenação de todas elas. Por exemplo:

```
_\$ (concatena-strings (list "A" "vista" "da" "Baixa" "da" "Banheira"))
"AvistadaBaixadaBanheira"
```

Exercicio 6.12 A função `concatena-strings` “cola” as palavras umas às outras sem deixar espaços entre elas. Defina uma nova função denominada `forma-frase` que recebe uma lista de *strings* e concatena-as mas deixando um espaço entre cada duas palavras. Por exemplo:

```

_§ (forma-frase (list "A" "vista" "da" "Baixa" "da" "Banheira"))
"A vista da Baixa da Banheira"

```

Exercicio 6.13 Defina a função `frase-aleatoria` que recebe uma lista de listas de palavras alternativas e devolve uma frase feita pela escolha aleatória de palavras de cada uma das listas. Por exemplo, sucessivas avaliações da seguinte expressão

```

(defun frase-aleatoria
  (list (list "AutoLisp" "Scheme" "Common Lisp")
        (list "é" "sempre foi" "continua a ser")
        (list "uma linguagem")
        (list "fantástica." "fabulosa." "moderna.")))

```

podem produzir os seguintes resultados:

```

"Common Lisp é uma linguagem moderna."
"Common Lisp é uma linguagem fantástica."
"Scheme sempre foi uma linguagem moderna."
"AutoLisp continua a ser uma linguagem fantástica."
"AutoLisp é uma linguagem moderna."
"Scheme é uma linguagem fantástica."
"Scheme continua a ser uma linguagem fantástica."

```

Sugestão: defina a função `frase-aleatoria` empregando as funções `forma-frase` e `um-de-cada`.

Exercicio 6.14 Considere a criação de uma função para fazer discursos. A ideia básica é que num discurso dizemos sempre a mesma coisa mas de formas diferentes para não parecermos repetitivos. Eis um exemplo da interacção pretendida:

```

_§ (discurso-aleatorio)
"Queridos amigos. É com grande satisfação que aqui vos revejo."
_§ (discurso-aleatorio)
"Queridos companheiros. É com imensa alegria que nesta sala vos reencontro."
_§ (discurso-aleatorio)
"Queridos colegas. É com grande satisfação que neste local vos reencontro."
_§ (discurso-aleatorio)
"Queridos colegas. É com imensa alegria que nesta sala vos revejo."
_§ (discurso-aleatorio)
"Queridos companheiros. É com enorme satisfação que aqui vos reencontro."
_§ (discurso-aleatorio)
"Caros colegas. É com grande satisfação que nesta sala vos revejo."
_§ (discurso-aleatorio)
"Estimados companheiros. É com grande alegria que neste local vos reencontro."
_§ (discurso-aleatorio)
"Caros amigos. É com grande alegria que aqui vos revejo."
_§ (discurso-aleatorio)
"Caros colegas. É com imensa alegria que neste local vos revejo."
_§ (discurso-aleatorio)
"Estimados amigos. É com imensa satisfação que nesta sala vos revejo."

```

Estude a interacção apresentada e defina a função `discurso-aleatorio`.

6.6 Enumerações

Vamos agora considerar uma função que será útil no futuro para criar enumerações. Dados os limites a e b de um intervalo $[a, b]$ e um incremento i , a função deverá devolver uma lista com todos os números $a, a + i, a + 2i, a + 3i, a + 4i, \dots$, até que $a + ni > b$.

Mais uma vez, a recursão ajuda: a enumeração dos números no intervalo $[a, b]$ com um incremento i é exactamente o mesmo que o número a

seguido da enumeração dos números no intervalo $[a + i, b]$. O caso mais simples de todos é uma enumeração num intervalo $[a, b]$ em que $a > b$. Neste caso o resultado é simplesmente uma lista vazia.

A definição da função é agora trivial:

```
(defun enumera (a b i)
  (if (> a b)
      ()
      (cons a
            (enumera (+ a i) b i))))
```

Como exemplo, temos:

```
_\$ (enumera 1 5 1)
(1 2 3 4 5)
_\$ (enumera 1 5 2)
(1 3 5)
```

Para tornarmos a função ainda mais genérica, podemos tratar também o caso em que o incremento d é negativo. Isto é útil, por exemplo, para uma contagem regressiva: $(\text{enumera } 10 \ 0 \ -1)$.

Note-se que a definição actual da função `enumera` não permite isso pois a utilização de um incremento negativo provoca recursão infinita. É claro que o problema está no facto do teste de paragem ser feito com a função `>` que só é apropriada para o caso em que o incremento é positivo. No caso de incremento negativo deveríamos usar `<`. Assim, para resolvemos o problema temos de identificar primeiro qual a função correcta a usar para a comparação, algo que podemos fazer com um simples teste:

```
(defun enumera (a b i)
  (if ((if (> i 0)
            >
            <)
       a b)
      ()
      (cons a
            (enumera (+ a i) b i))))
```

Agora já podemos ter:

```
_\$ (enumera 1 5 1)
(1 2 3 4 5)
_\$ (enumera 5 1 -1)
(5 4 3 2 1)
_\$ (enumera 6 0 -2)
(6 4 2 0)
```

Exercício 6.15 Infelizmente, a função `enumera` é desnecessariamente ineficiente pois, apesar do incremento nunca mudar ao longo das invocações recursivas, estamos sistematicamente a testar se ele é positivo. Obviamente, basta testar uma única vez para se decidir, de uma vez por todas, qual é o teste que se deve fazer. Redefina a função `enumera` de modo a que não sejam feitos quaisquer testes desnecessários.

Exercicio 6.16 A função ι (pronuncia-se “iota”) representa uma enumeração desde 0 até um limite superior n exclusive, com os elementos da enumeração separados por um dado incremento, i.e.:

```
_\$ (iota 10 1)
(0 1 2 3 4 5 6 7 8 9)
_\$ (iota 10 2)
(0 2 4 6 8)
```

Defina a função `iota` à custa da função `enumera`.

Exercicio 6.17 A função `enumera` pode apresentar um comportamento bizarro quando o incremento não é um número inteiro. Por exemplo:

```
_\$ (enumera 0 1 (/ 1.0 10))
(0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0)
_\$ (enumera 0 1 (/ 1.0 100))
(0 0.01 0.02 0.03 0.04 ... 0.94 0.95 0.96 0.97 0.98 0.99)
```

Como podemos verificar, ao invés de terminar no limite superior 1, a função termina mais cedo. A causa deste comportamento está no facto de as frações $\frac{1}{10}$ e $\frac{1}{100}$ não serem representáveis com exactidão em notação binária e, por isso, à medida que se soma uma quantidade suficientemente grande destes números, acumula-se um erro que se vai tornando significativo.

O algoritmo de Kahan permite minimizar este problema. Pesquise este algoritmo e use-o para implementar uma nova versão da função `enumera` que evite acumulação de erros quando o incremento não é um número inteiro.

Exercicio 6.18 Escreva uma função `membro?` que recebe um objecto e uma lista e verifica se aquele objecto existe na lista.

Exercicio 6.19 Escreva uma função `elimina` que recebe, como argumentos, um elemento e uma lista e devolve, como resultado, outra lista onde esse elemento não aparece.

Exercicio 6.20 Escreva uma função `substitui` que recebe dois elementos e uma lista como argumentos e devolve outra lista com todas as ocorrências do segundo elemento substituídas pelo primeiro.

Exercicio 6.21 Escreva uma função `remove-duplicados` que recebe uma lista como argumento e devolve outra lista com todos os elementos da primeira mas sem duplicados, i.e.:

```
_\$ (remove-duplicados '(1 2 3 3 2 4 5 4 1))
(3 2 5 4 1)
```

6.7 Comparação de Listas

Vimos que o Lisp disponibiliza várias funções de comparação e, em particular, a função `=` aplicável a números, *strings* e símbolos.

No entanto, quando aplicamos esta função a pares os resultados não são os esperados:

```

_ $ (= 1 1)
T
_ $ (= "dois" "dois")
T
_ $ (= (cons 1 "dois") (cons 1 "dois"))
nil

```

Note-se que embora a função não tenha dado qualquer erro (implícando que é aplicável a pares), devolveu falso a indicar que os pares são diferentes. Experimentemos agora uma variante:

```

_ $ (setq par-1-dois (cons 1 "dois"))
(1 . "dois")
_ $ (= par-1-dois par-1-dois)
T

```

Como se pode ver, a comparação foi feita entre duas referência para um par criado anteriormente. Supreendentemente, o resultado é agora afirmativo. Qual é então a semântica da função = aplicado a pares? A resposta é subtil: quando aplicada a dois argumentos do tipo par, a função = devolve verdade apenas quando os dois argumentos são o *mesmo* par.

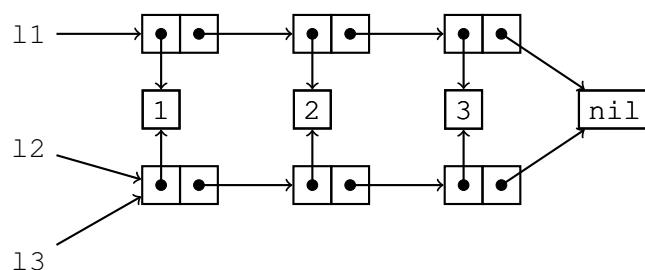
Consideremos, por exemplo, a seguinte interação:

```

_ $ (setq l1 (list 1 2 3))
(1 2 3)
_ $ (setq l2 (list 1 2 3))
(1 2 3)
_ $ (setq l3 l2)
(1 2 3)

```

É fácil de ver que a avaliação das expressões anteriores cria as seguintes estruturas em notação de caixa e ponteiro:



O aspecto crucial destas estruturas é que 12 e 13 se referem à *mesma* lista. Por este motivo, o operador = considera o valor de 12 igual ao valor de 13. Já o mesmo não acontece quando se compara 12 com 11 pois estes nomes designam diferentes estruturas. Este comportamento é visível na seguinte interacção:

```

_§ (= 11 12)
nil
_§ (= 12 13)
T

```

Note-se que o que está em causa nas expressões anteriores é apenas se os valores que são passados como argumentos da função = são, na realidade, o mesmo valor. É irrelevante se esses valores são provenientes da avaliação de variáveis ou da avaliação de quaisquer outras expressões.

Apesar de termos usado o operador = em todas as comparações que fizemos, pragmaticamente falando, o seu uso costuma estar reservado para comparações entre números e *strings*, utilizando-se um outro operador diferente denominado `eq` para testar a igualdade de valores. A função `eq` serve precisamente para testar se os seus dois argumentos são, na realidade, o mesmo objecto.⁴⁷

Exercicio 6.22 Explique a seguinte interacção:

```

_§ (setq l1 (list 3 4 5))
(3 4 5)
_§ (setq l2 (cons 1 (cons 2 l1)))
(1 2 3 4 5)
_§ (eq l1 l2)
nil
_§ (eq (cdr (cdr l2)) 11)
T

```

6.8 Comparação de Símbolos

Até agora, apenas temos usado símbolos para a construção dos nossos programas, como nomes de funções e variáveis. No entanto, os símbolos são também úteis noutras contextos em que são usados como valores.

Por exemplo, como vimos na secção 2.29, a função `type` devolve um símbolo que identifica o tipo de objecto. Vimos também que, para a definição de um reconhecedor para um determinado tipo, era útil poder fazer comparações entre símbolos, de modo a identificarmos o símbolo pretendido:

```
(defun realp (obj)
  (= (type obj) 'real))
```

Acontece que, em Lisp, os símbolos gozam de uma propriedade muito importante: são *únicos*. Isto quer dizer que não existem dois símbolos diferentes com o mesmo nome. Assim, se duas expressões avaliarem para símbolos com o mesmo nome, então essas expressões têm, como valor, exactamente o mesmo símbolo. Consequentemente, o operador mais adequado para fazer a comparação é, mais uma vez, o `eq`. Por este motivo, a definição da função `realp` deveria ser, na realidade:

⁴⁷Em Auto Lisp, a função `eq` consegue ainda comparar números de tipos diferentes, como 1 e 1.0. Este comportamento, contudo, não é usual na família de dialectos Lisp.

```
(defun realp (obj)
  (eq (type obj) 'real))
```

6.9 Comparação Estrutural de Listas

Até agora, a comparação de listas está restrita a sabermos se duas listas são, na realidade, a mesma lista. No entanto, frequentemente ocorre a necessidade de sabermos se duas listas são *estruturalmente iguais*. A igualdade estrutural define-se de forma recursiva:

- Qualquer objecto é estruturalmente igual a si próprio.
- Dois *cons* são estruturalmente iguais se os *cars* forem estruturalmente iguais e os *cdrs* forem estruturalmente iguais.

Esta definição sugere que temos de tratar distintamente os elementos atómicos dos não-atómicos. Se forem atómicos, então só serão iguais se forem *eq*. Caso contrário, se não são *eq* e algum deles é atómico, não podem ser iguais. Finalmente, se nenhum deles é atómico então são ambos não-atómicos e se os *cars* forem iguais, os *cdrs* também terão de ser. Desta forma, podemos trivialmente definir a igualdade estrutural com uma função igual cuja definição é:

```
(defun igual? (obj1 obj2)
  (cond ((eq obj1 obj2)
         t)
        ((or (atom obj1) (atom obj2))
         nil)
        ((igual? (car obj1) (car obj2))
         (igual? (cdr obj1) (cdr obj2)))))
```

A linguagem Auto Lisp generaliza ainda esta função de modo a que a comparação de entidades numéricas possa contemplar uma determinada *margem de erro*: dois números são considerados iguais se a sua diferença, em valor absoluto, é inferior a esta margem de erro. A função *equal* pré-definida em Auto Lisp recebe, para além das entidades a comparar, um terceiro argumento opcional correspondente a esta margem de erro. Note-se o seguinte exemplo:

```
_\$ (equal (list 1 3 (list 3 4))
            (list 1.01 2.99 (list 3 4.01)))
nil
_\$ (equal (list 1 3 (list 3 4))
            (list 1.01 2.99 (list 3 4.01))
            0.05)
T
```

7 Carregamento de Ficheiros

À medida que vamos definindo funções úteis torna-se conveniente poder contar com elas sempre que necessário. Para isso, o AutoCad disponibiliza uma função que, a partir de um ficheiro contendo todas as definições pretendidas, *carrega* essas definições para a sua memória. A função denomina-se `load`⁴⁸ e recebe, como argumento, a *string* com o caminho para o ficheiro.⁴⁹

Por exemplo, se o ficheiro `C:\AutoLisp\utilitarios.lsp` conter funções que pretendemos ter disponíveis no AutoCad, a avaliação da expressão:

```
(load "C:\\AutoLisp\\\\utilitarios.lsp")
```

irá provocar a avaliação de todas as expressões contidas nesse ficheiro e, no final, devolve o valor da última expressão.

Note-se, na expressão anterior, a utilização de `\` em vez de `\.`. Como vimos na secção 2.9, nas *strings*, o carácter `\` é um carácter de *escape*: serve para incluir na *string* outros caracteres como, por exemplo, as aspas. Infelizmente, o Windows usa o mesmo carácter como separador de directorias, pelo que somos obrigados a usar `\`. Para simplificar, o Auto Lisp também admite o carácter `/` como separador de directorias em *strings* que designam caminhos para ficheiros. Isto quer dizer que a expressão anterior também poderia ser escrita na forma:

```
(load "C:/AutoLisp/utilitarios.lsp")
```

Se, no caminho para o ficheiro, se omitir a extensão deste,⁵⁰ o Auto Lisp irá procurar pelo ficheiro mais recente que tenha aquele nome e uma das extensões `.vlx`, `.fas` e `.lsp`. Mais à frente veremos o significado destas extensões.

Infelizmente, o uso de caminhos completos para os nossos ficheiros torna os nossos programas dependentes da localização correcta desses ficheiros. Isto dificulta muito o trabalho de quem quer utilizar os nossos ficheiros mas usa uma estrutura de directórios diferente da nossa. Para obviar a este problema, o Auto Lisp tem um mecanismo que permite evitar especificar o caminho para um ficheiro, permitindo-nos escrever apenas:

```
(load "utilitarios.lsp")
```

ou, omitindo também a extensão:

⁴⁸O comando `APPLOAD` do AutoCad faz basicamente o mesmo mas usa uma interface gráfica para a selecção do ficheiro.

⁴⁹A função recebe ainda mais um argumento, opcional....

⁵⁰A extensão de um ficheiro é o conjunto de (usualmente) três letras que vem a seguir ao nome do ficheiro e que é separado deste por um ponto. A extensão é usada normalmente para identificar o *tipo* do ficheiro.

```
(load "utilitarios")
```

Quando omitimos o caminho para o ficheiro, então o Auto Lisp irá procurar o ficheiro segundo a seguinte estratégia:

1. O ficheiro é procurado na directória actual.⁵¹
2. Se não for encontrado, o ficheiro é procurado na directória do desenho actual.
3. Se não for encontrado, o ficheiro é pesquisado em cada um dos directórios especificados no caminho de pesquisa do AutoCad.
4. Se não for encontrado, o ficheiro é pesquisado na directória que contém o AutoCad.

Uma vez que não é conveniente alterar a directória actual ou a directória do desenho actual ou colocar os nossos ficheiros misturados com os do AutoCad, a melhor solução para facilmente podermos carregar os nossos utilitários consiste em criar uma directória onde colocamos os vários ficheiros e acrescentamos o caminho para essa directória no caminho de pesquisa do AutoCad. Esse caminho é especificado por uma sequência de directórios indicados no tabulador “Files,” opção “Support File Search Path” do diálogo que resulta do comando OPTIONS.

Note-se que a sequência de directórios indicados nesse tabulador define a ordem de pesquisa feita pelo AutoCad. Se pretendemos que os nossos ficheiros sejam encontrados primeiro que outros com o mesmo nome, então deveremos acrescentar o caminho para a nossa directória *antes* dos outros caminhos na sequência.

7.1 Carregamento de Dependências

Em geral, cada ficheiro de Lisp que criamos deve indicar explicitamente quais são as dependências que tem de outros ficheiros, i.e., quais são os ficheiros que devem estar previamente carregados para que aquele possa ser também carregado.

Como exemplo, consideremos um ficheiro `utils.lsp` que define operações básicas úteis. Com base nestas operações básicas criámos várias operações sobre listas que guardámos num ficheiro `listas.lsp`. Finalmente, temos um terceiro ficheiro `grafico.lsp` que depende de algumas das operações definidas em `listas.lsp` e ainda de algumas operações destinadas a facilitar o desenho de gráficos e que se encontram no ficheiro `grafico-utils.lsp`.

⁵¹Esta é a directória em que o AutoCad começa a funcionar. No caso do Windows, corresponde ao valor da opção “Start In” das propriedades do ícone do AutoCad.

Para que possamos avaliar (`(load "grafico")`) e, com isso, carregar tudo o que é preciso, é conveniente que cada ficheiro indique as suas dependências. Assim, o conteúdo do ficheiro `grafico.lsp`, deverá ser algo da forma:

```
grafico.lsp
(load "listas")
(load "grafico-utils")

(defun ...)
...
```

Naturalmente, o ficheiro `listas.lsp` será da forma:

```
listas.lsp
(load "utils")

(defun ...)
...
```

Deste modo, quando avaliamos (`(load "grafico")`) o Auto Lisp localiza o ficheiro em questão e, uma vez que a primeira expressão desse ficheiro é (`(load "listas")`), o Auto Lisp suspende momentaneamente o carregamento do ficheiro `grafico` e procura pelo ficheiro `listas`. Quando o encontra, começa a avaliar as suas expressões mas, mais uma vez, como a primeira dessas expressões é (`(load "utils")`) o Auto Lisp suspende momentaneamente o carregamento do ficheiro e procura pelo ficheiro `utils`. Quando o encontra, avalia todo o seu conteúdo, regressando então ao carregamento do ficheiro `listas`. Após ter avaliado todas as expressões deste ficheiro regressa finalmente ao primeiro ficheiro `grafico` para passar à avaliação da expressão seguinte: (`(load "grafico-utils")`). Novamente, o Auto Lisp suspende momentaneamente o carregamento do ficheiro `grafico`, para agora procurar pelo ficheiro `grafico-utils`, avaliar todas as suas expressões e, por fim, regressar ao ficheiro `grafico` para avaliar as restantes formas.

Esta relação de dependência entre os vários ficheiros pode ser representada graficamente num *grafo*, tal como se apresenta na Figura 56.

7.2 Módulos

A um conjunto de ficheiros que, juntos, providenciam uma determinada funcionalidade, chamamos um *modulo*. Assim, os ficheiros `grafico`, `listas`, `utils` e `grafico-utils` constituem um módulo. O mesmo se poderá dizer do subconjunto constituído pelos ficheiros `listas` e `utils` e, no limite, do subconjunto constituído pelo ficheiro `utils`. Assim, seria mais correcto dizer que existe um módulo de utilitários constituído pelo único ficheiro `utils`, existe um módulo de `listas` que depende do módulo de

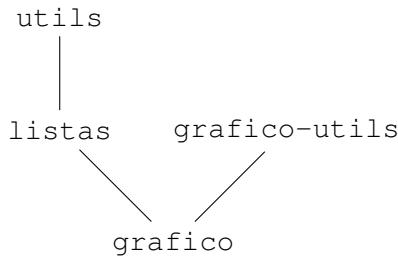


Figura 56: Dependências entre os ficheiros `grafico`, `listas`, `utils` e `grafico-utils`.

utilitários e que é composto pelo ficheiro `listas` e, finalmente, existe um módulo de gráficos que depende do módulo de listas e que é constituído pelos ficheiros `grafico` e `grafico-utils`.

O conceito de módulo é muito importante pois permite abstrair um conjunto de ficheiros numa só entidade. Do ponto de vista de quem usa o módulo, é irrelevante saber se ele é constituído por um, dois ou mil ficheiros, qual a ordem de carregamento desses ficheiros ou quais as dependências que esse módulo tem de outros módulos. Desde que não se reduza a funcionalidade de um módulo, esta abstracção permite que quem criou um módulo possa alterá-lo, acrescentando, retirando ou modificando ficheiros, sem afectar quem depende do módulo.

Consideremos agora uma situação mais complexa (mas mais frequente). Para além dos ficheiros `grafico-utils.lsp`, `grafico.lsp`, `listas.lsp` e `utils.lsp` que tínhamos no exemplo anterior, temos ainda, num outro ficheiro chamado `testes.lsp`, operações úteis para a realização de testes, operações essas cuja definição depende, por um lado, de algumas das operações de `utils` e, por outro, de algumas das operações de `listas`, tal como se apresenta em seguida:

testes.lsp
<pre>(load "utils") (load "listas") (defun ...) ...</pre>

Finalmente, temos um ficheiro `colunas.lsp` contendo operações para o desenho de colunas e que depende, quer de `testes`, quer de `utils`, quer de `graficos`:

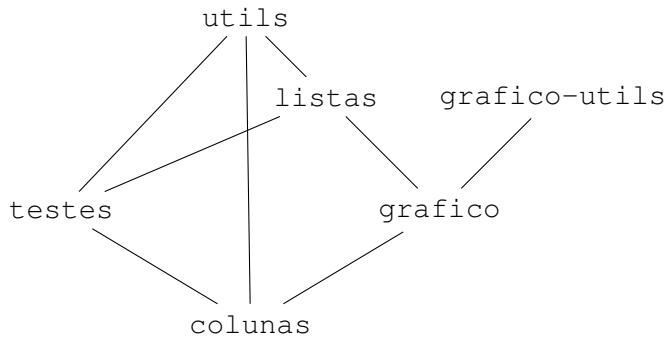


Figura 57: Dependências entre os ficheiros `grafico`, `listas`, `utils` e `grafico-utils`.

```

----- colunas.lsp -----
(load "testes")
(load "utils")
(load "graficos")

(defun ...)
...

```

Estas relações de dependências estão representadas no grafo presente na Figura 57.

Se `colunas` depende de `testes` e `testes` depende de `utils` então é redundante dizer que `colunas` depende de `utils`. De facto, a relação de dependência uma relação *transitiva*, i.e., se x depende de y e y depende de z , então automaticamente x depende de z . Isto sugere que, no ficheiro `colunas`, deveríamos remover a expressão que carrega o ficheiro `utils` pois ele já terá sido carregado automaticamente em consequência da avaliação da expressão anterior (`(load "testes")`).

Contudo, existe um pressuposto importante neste raciocínio: para que `colunas.lsp` possa dispensar o carregamento de `utils`, é preciso que o autor de `colunas.lsp` saiba que `utils` é uma dependência de `testes`. Ora isso implica que temos de conhecer a estrutura interna do módulo de `testes`, o que viola a abstracção desse módulo. Essa violação tem como consequência que quaisquer alterações que o autor do módulo de `testes` fizer na estrutura do seu módulo poderá ter implicações negativas para o módulo de `colunas`. Por exemplo, se o autor do módulo de `testes` decidir terminar a dependência que tinha de `utils`, então o módulo de `colunas` irá ter problemas pois o ficheiro `utils`, afinal, já não é carregado.

Por este motivo, é sempre preferível que cada módulo indique explicitamente as suas dependências de outros módulos e não assuma qualquer conhecimento sobre a estrutura interna ou dependências desses módulos.

Infelizmente, preservar a abstracção dos módulos acarreta um problema:

um ficheiro pode acabar por ser carregado múltiplas vezes. Para percebermos isto, pensemos na sequência de carregamentos que ocorre quando avaliamos (`(load "colunas")`):

1. `(load "testes")`
 - (a) `(load "utils")`
 - (b) `(load "listas")`
 - i. `(load "utils")`
2. `(load "utils")`
3. `(load "grafico")`
 - (a) `(load "listas")`
 - i. `(load "utils")`
4. `(load "grafico-utils")`

É fácil ver que, ao preservarmos a abstracção de cada módulo, i.e., ao não conhecermos as dependências de cada módulo, acabamos inadvertidamente por carregar múltiplas vezes vários ficheiros: `listas` é carregado duas vezes e `utils` é carregado quatro vezes.

Obviamente, este múltiplo carregamento de ficheiros é indesejável. Para resolver este problema, devemos complementar o processo de carregamento com um registo de tudo o que já tiver sido carregado para evitar carregar algum ficheiro mais do que uma vez.

Para isso, vamos começar por usar uma variável global para conter os nomes dos ficheiros já carregados e, em seguida, definimos uma função `usa` que, dado o nome de um ficheiro que se pretende usar, verifica se ele já foi carregado e, se não tiver sido, carrega-o. Estas duas formas serão colocadas no ficheiro `usa.lisp`:

```
usa.lsp
(setq ficheiros-carregados ())  
  
(defun usa (nome)
  (if (member nome ficheiros-carregados)
      nil
      (progn
        (load nome)
        (setq ficheiros-carregados
              (cons nome ficheiros-carregados)))
      t)))
```

Usando a função `usa` no lugar de `load`, passamos a ter os ficheiros:

```
grafico.lsp —  
(usa "listas")  
(usa "grafico-utils")
```

```
(defun ...)  
...
```

```
listas.lsp —  
(usa "utils")
```

```
(defun ...)  
...
```

```
testes.lsp —  
(usa "utils")  
(usa "listas")
```

```
(defun ...)  
...
```

```
colunas.lsp —  
(usa "testes")  
(usa "utils")  
(usa "graficos")
```

```
(defun ...)  
...
```

Agora, ao avaliarmos a expressão `(load "colunas")` obtemos o seguinte comportamento:

1. `(usa "testes")` Ainda não foi carregado.
2. `(load "testes")`
 - (a) `(usa "utils")` Ainda não foi carregado.
 - (b) `(load "utils")`
 - (c) `(usa "listas")` Ainda não foi carregado.
 - (d) `(load "listas")`
 - i. `(usa "utils")` Já foi carregado.
3. `(usa "utils")` Já foi carregado.
4. `(usa "grafico")` Ainda não foi carregado.
 - (a) `(usa "listas")` Já foi carregado.
5. `(usa "grafico-utils")` Ainda não foi carregado.

Como podemos ver, agora cada ficheiro é carregado uma única vez.

7.3 Carregamento Automático

Com a função `uso` já temos a possibilidade de carregar os módulos de que necessitamos sem nos preocuparmos com carregamentos duplicados. Infelizmente, esta função só está disponível depois de a carregarmos, o que é inconveniente pois temos de nos lembrar de carregar manualmente o ficheiro "usa" *sempre* que iniciamos o trabalho em AutoCad. Para obviar esta inconveniência, o AutoCad permite o carregamento *automático* de ficheiros segundo o seguinte esquema de ficheiros:

- O ficheiro `acad.lsp` é carregado automaticamente quando o AutoCad inicia.
- O ficheiro `acaddoc.lsp` é carregado automaticamente quando abrimos um desenho.

O AutoCad pesquisa por estes ficheiros usando as mesmas regras anunciadas anteriormente para localizar ficheiros sem a indicação do caminho.

Este esquema permite-nos ter um ficheiro de inicialização que carrega tudo aquilo que entendemos ser útil para a nossa sessão de trabalho. Se for útil independentemente do desenho em questão, deverá ser definido ou carregado no ficheiro `acad.lsp`. Se for útil para um só desenho, então o melhor é ter um ficheiro `acaddoc.lsp` contendo tudo o que for preciso e guardado na directória desse desenho. Desta forma, diferentes desenhos (guardados em diferentes directórias) podem ter diferentes ficheiros de inicialização `acaddoc.lsp`.

Há ainda outros ficheiros que o AutoCad carrega automaticamente mas que estão relacionados com outros recursos do AutoCad e não com o Auto Lisp.

Tendo esta informação em conta, deveremos criar um ficheiro `acad.lsp` numa das directórias presentes no caminho de pesquisa do AutoCad e, nesse ficheiro, devermos incluir a seguinte expressão:

```
----- acad.lsp -----  
(load "usa")
```

7.4 Interpretação e Compilação

Vimos que, quando o Auto Lisp avalia uma expressão, ele primeiro analisa-a, decompondo-a nas suas subexpressões e, recursivamente, avalia estas de acordo com as regras da linguagem, seguindo este processo até chegar às expressões mais simples de todas, para as quais o Auto Lisp sabe imediatamente qual é o valor, passando então a combinar os valores das subavaliações até produzir o valor da expressão original.

Este processo, que o Auto Lisp segue para a avaliação de qualquer expressão, denomina-se *interpretação* e o programa que o realiza diz-se um

interpretador. Quando estamos a usar o REPL do AutoCad estamos, na realidade, a usar um interpretador para a linguagem Auto Lisp.

Embora a interpretação seja um processo muito útil para a depuração dos programas, sofre de um problema: é computacionalmente pouco eficiente. Se pensarmos, por exemplo, na função `factorial`, é fácil vermos que a avaliação de uma invocação da função factorial implica uma sequência de análises e combinação de resultados que tem de ser repetida para cada uma das sucessivas invocações recursivas da função.

Existe um outro processo de avaliação que, embora menos flexível que a interpretação, tem o potencial para ser muito mais eficiente: a *compilação*.

A compilação é uma operação que é realizada por um *compilador* e que se baseia em analisar expressões sem as avaliar e em transformá-las noutras expressões equivalentes que, quando avaliadas, produzam os mesmos resultados mas de forma mais eficiente. A ideia fundamental da compilação é que é possível converter um programa escrito numa linguagem num programa escrito noutra linguagem que seja mais eficiente de interpretar ou, no limite, que seja directamente executável pelo computador.

Em geral, a compilação em Lisp transforma um ficheiro contendo expressões Lisp num outro ficheiro com as mesmas expressões traduzidas para outra linguagem muito mais eficiente de avaliar. Em geral, há duas vantagens em usar ficheiros compilados:

- É muito mais rápido carregar um ficheiro compilado do que um não compilado.
- A invocação das funções que foram definidas a partir de um ficheiro compilado é mais rápida, por vezes, muito mais rápida.

Para se compilar um ficheiro é necessário invocar a função `vlisp-compile`. Esta função recebe três argumentos:

1. Um símbolo indicando qual o grau de optimização desejado.
2. Uma *string* indicando o ficheiro a compilar. Se a extensão do ficheiro for omissa, considera-se “`.lsp`” como extensão. Se apenas se indicar o nome do ficheiro, sem o caminho até ele, o Visual Lisp usa o processo normal de procura de ficheiros para o encontrar.
3. Uma *string* opcional indicando o caminho para o ficheiro a criar para conter o resultado da compilação. Quando omissa, é idêntico ao argumento anterior, mas usando “`.fas`” como extensão. Quando não é omissa, então poderá ser apenas o nome do ficheiro (caso em que o Visual Lisp irá colocar o ficheiro na sua directória de instalação) ou o caminho completo para o ficheiro.

Em relação ao nível de optimização, há três possibilidades (indicadas pelos símbolos `st`, `lsm` e `lsa`) mas apenas a primeira preserva (quase) todas as características da linguagem Lisp. As seguintes alteram ligeiramente a semântica da linguagem para conseguirem obter maior *performance*. Por exemplo, para lá do nível `st`, a redefinição de funções deixa de ter efeito e a informação de depuração é cada vez menor, tornando os programas mais eficientes mas dificultando a vida ao programador quando ele tenta perceber qualquer erro que tenha ocorrido.⁵²

Como exemplo de utilização, a seguinte expressão mostra a compilação do ficheiro `listas.lsp`:⁵³

```
(vlisp-compile 'st "listas")
```

8 Listas de Coordenadas

As listas constituem um tipo de dados extraordinariamente útil. Através da utilização de listas é possível escrever programas mais simples, mais claros e mais genéricos.

Vimos anteriormente que sempre que pretendíamos indicar no Auto-Cad uma posição geométrica no plano bidimensional usávamos os construtores de coordenadas Cartesianas ou polares que, por sua vez, criavam lista com as coordenadas correspondentes, contendo dois elementos no caso bidimensional e três elementos no caso tri-dimensional.

Uma outra utilização frequente de listas ocorre quando se pretende agrupar um conjunto de entidades geométricas. Neste caso, contrariamente ao que acontecia com a utilização de listas para representar posições 2D ou 3D, a lista usualmente não possui um número fixo de elementos.

8.1 Polígonos

Imaginemos, a título de exemplo, que pretendemos representar um *polígono*. Por definição, um polígono é uma figura plana limitada por um caminho fechado composto por uma sequência de segmentos de recta. Cada segmento de recta é uma *aresta* (ou *lado*) do polígono. Cada ponto onde se encontram dois segmentos de recta é um *vértice* do polígono.

A partir da definição de polígono é fácil vermos que uma das formas mais simples de representar um polígono será através de uma sequência de coordenadas que indica qual a ordem pela qual devemos unir os vértices com uma aresta e onde se admite que o último elemento será unido ao primeiro. Para representar esta sequência podemos simplesmente usar

⁵²Essa dificuldade de compreensão pode ser útil quando se pretende ceder a terceiros um programa mas em que não se pretende que se saiba como está feito.

⁵³Note-se a utilização da plica para evitar a avaliação do símbolo `st`.

uma lista. Por exemplo, uma lista com as coordenadas de quatro vértices pode representar um quadrilátero, um octógono será representado por uma lista de oito vértices, um hentricontágono será representado por uma lista de trinta e um vértices, etc.

A representação de polígonos usando listas é apenas o primeiro passo para os podermos visualizar graficamente. O segundo passo é a invocação dos comandos do AutoCad que constroem o polígono em questão. Para isso, temos de invocar o comando `line` (ou, alternativamente, o comando `pline`) e, de seguida, passar os sucessivos pontos para o AutoCad. No final da sequencia de pontos temos ainda de passar a opção `close` para que o AutoCad “feche” o polígono, unindo o último ao primeiro ponto. Decompondo o problema em duas partes, podemos escrever:

```
(defun poligono (vertices)
  (command "_line")
  (passa-pontos vertices)
  (command "_close"))
```

Uma vez que a passagem dos pontos tem de ser feita percorrendo a lista com os vértices, vamos definir uma função recursiva para o fazer:

```
(defun passa-pontos (pontos)
  (if (null pontos)
      nil
      (progn
        (command (car pontos))
        (passa-pontos (cdr pontos))))))
```

Usando a função `poligono` podemos agora facilmente desenhar inúmeros polígonos. Por exemplo, as seguintes avaliações produzem o resultado apresentado na Figura 58.

```
(poligono (list (xy -2 -1) (xy 0 2) (xy 2 -1)))
(pолигоно (list (xy -2 1) (xy 0 -2) (xy 2 1)))
```

É importante notarmos que a função `poligono` é independente de quaisquer polígonos em particular. Ela tanto se aplica ao desenho de triângulos como quadrados como qualquer outro polígono. Esta independência tornou-se possível pela utilização de listas para representar as coordenadas dos vértices dos polígonos. Agora, apenas é necessário concentrarmo-nos na criação destas listas de coordenadas.

8.1.1 Estrelas Regulares

O polígono representado na Figura 58 foi gerado “manualmente” pela sobreposição de dois triângulos. Um polígono mais interessante é o famoso

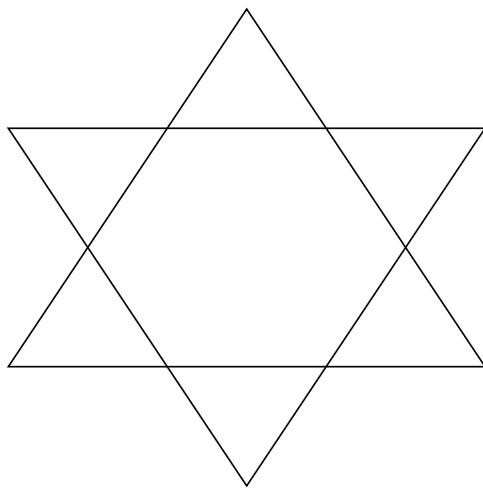


Figura 58: Polígonos em AutoCad.

pentagrama, ou estrela de cinco pontas, que tem sido usado com conotações simbólicas, mágicas ou decorativas desde os tempos da Babilónia.⁵⁴ A Figura 59 demonstra o uso do pentagrama (bem como do octograma, do octógono e do hexágono) como elemento decorativo e estrutural numa janela de pedra.

À parte as conotações extra-geométricas, o pentagrama é, antes de tudo, um polígono. Por este motivo, é desenhável pela função `poligono` desde que consigamos produzir uma lista com as coordenadas dos vértices. Para isso, vamos reportar-nos à Figura 60 onde é visível que os cinco vértices do pentagrama dividem o círculo em 5 partes, com arcos de $\frac{2\pi}{5}$ cada. Dado o centro do pentagrama, o seu vértice superior faz um ângulo de $\frac{\pi}{2}$ com o eixo das abcissas. Esse vértice deve ser unido, não com o vértice seguinte, mas sim com o imediatamente a seguir a esse, i.e., após uma rotação de dois arcos ou $\frac{2 \cdot 2\pi}{5} = \frac{4\pi}{5} = 0.8\pi$. Estamos agora em condições de definir a função `vertices-pentagrama` que constrói a lista com os vértices do pentagrama:

⁵⁴O pentagrama foi símbolo de perfeição matemática para os Pitagóricos, foi símbolo do domínio do espírito sobre os quatro elementos da matéria pelos ocultistas, representou as cinco chagas de Cristo crucificado para os Cristãos, foi associado às proporções do corpo humano, foi usado como símbolo maçônico e, quando invertido, até foi associado ao satanismo.

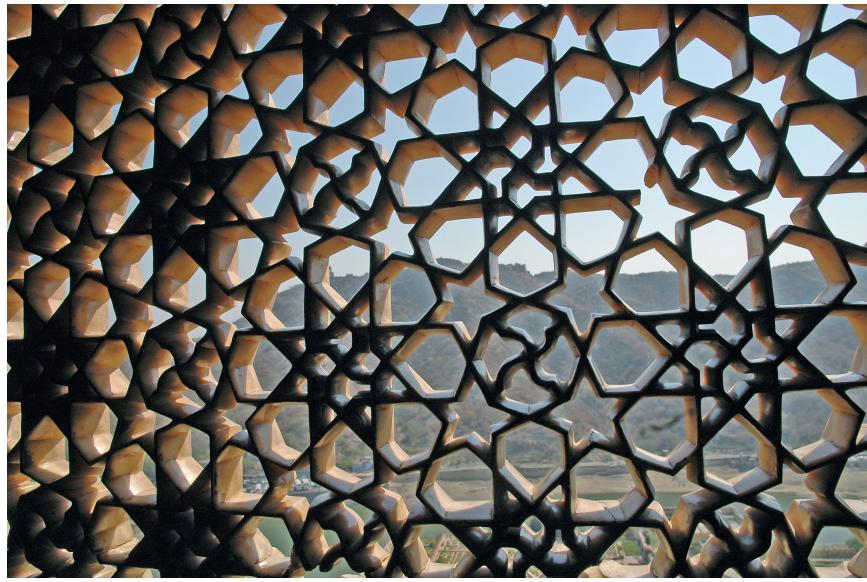


Figura 59: Variações de Estrelas numa janela do Forte de Amber, localizado no estado de Jaipur, na Índia. Fotografia de David Emmett Cooley.

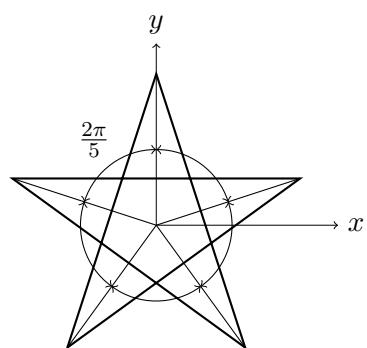


Figura 60: Construção do Pentagrama.

```
(defun vertices-pentagrama (centro raio)
  (list (+pol centro raio (+ (/ pi 2) (* 0 0.8 pi)))
        (+pol centro raio (+ (/ pi 2) (* 1 0.8 pi)))
        (+pol centro raio (+ (/ pi 2) (* 2 0.8 pi)))
        (+pol centro raio (+ (/ pi 2) (* 3 0.8 pi)))
        (+pol centro raio (+ (/ pi 2) (* 4 0.8 pi)))))

(polygono (vertices-pentagrama (xy 0 0) 1))
```

Como é óbvio, a função `vertices-pentagrama` possui excessiva repetição de código, pelo que seria preferível encontrarmos uma forma mais estruturada de gerarmos aqueles vértices. Para isso, vamos começar por pensar na generalização da função.

O caso geral de um pentagrama é a *estrela regular*, em que se faz variar o número de vértices e o número de arcos que separam um vértice do vértice a que ele se une. O pentagrama é um caso particular da estrela regular em que o número de vértices é cinco e o número de arcos de separação é dois. Matematicamente falando, uma estrela regular representa-se pelo símbolo de Schläfli⁵⁵ $\{ \frac{v}{a} \}$ em que v é o número de vértices e a é o número de arcos de separação. Nesta notação, um pentagrama escreve-se como $\{ \frac{5}{2} \}$.

Para desenarmos estrelas regulares vamos idealizar uma função que, dado o centro da estrela, o raio do círculo circunscrito, o número de vértices v (a que chamaremos `n-vertices`) e o número de arcos de separação a (a que chamaremos `n-arcs`), calcula o tamanho do arco $\Delta\phi$ que é preciso avançar a partir de cada vértice. Esse arco é, obviamente, $\Delta\phi = a \frac{2\pi}{v}$. Tal como no pentagrama, para primeiro vértice vamos considerar um ângulo inicial de $\phi = \frac{\pi}{2}$. A partir desse primeiro vértice não é preciso mais do que ir aumentando o ângulo ϕ de $\Delta\phi$ de cada vez. As seguintes funções implementam este raciocínio:

```
(defun vertices-estrela (p raio n-vertices n-arcs)
  (pontos-circulo p
    raio
    pi/2
    (* n-arcs (/ 2*pi n-vertices))
    n-vertices))

(defun pontos-circulo (p raio fi dfi n)
  (if (= n 0)
      (list)
      (cons (+pol p raio fi)
            (pontos-circulo p
              raio
              (+ fi dfi)
              dfi
              (- n 1)))))
```

Com a função `vertices-estrela` é agora trivial gerar os vértices de

⁵⁵Ludwig Schläfli foi um matemático e geómetra Suíço que fez importantes contribuições, em particular, na geometria multidimensional.

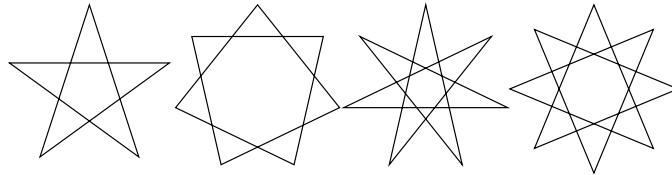


Figura 61: Estrelas regulares em AutoCad. Da esquerda para a direita, temos um pentagrama ($\{ \frac{5}{2} \}$), dois heptagramas ($\{ \frac{7}{2} \}$ e $\{ \frac{7}{3} \}$) e um octograma ($\{ \frac{8}{3} \}$).

qualquer estrela regular. A Figura 61 apresenta as estrelas regulares desenhadas a partir das seguintes expressões:

```
(poligono (vertices-estrela (xy 0 0) 1 5 2))
(poligono (vertices-estrela (xy 2 0) 1 7 2))
(poligono (vertices-estrela (xy 4 0) 1 7 3))
(poligono (vertices-estrela (xy 6 0) 1 8 3))
```

É muito importante salientarmos a forma como a representação gráfica de estrelas está separada em duas partes distintas. De um lado, o da função `vertices-estrela`, produzimos as coordenadas dos vértices das estrelas pela ordem em que estes devem ser unidos. No outro lado, o da função `poligono`, usamos essas coordenadas para criar uma representação gráfica dessa estrela baseada em linhas que ligam os seus vértices. A passagem das coordenadas de um lado para o outro é realizada através de uma lista que é “produzida” num lado e “consumida” no outro.

Esta utilização de listas para separar diferentes processos é fundamental e será por nós repetidamente explorada para simplificar os nossos programas.

8.1.2 Polígonos Regulares

Um *polígono regular* é um polígono que tem todos os lados de igual comprimento e todos os ângulos de igual amplitude. A Figura 62 ilustra exemplos de polígonos regulares. Como é óbvio, um *polígono regular* é um caso particular de uma estrela regular em que o número de arcos de separação é um.

Para criarmos polígonos regulares, vamos definir uma função denominada `vertices-poligono-regular` que gera uma lista de coordenadas correspondente às posições dos vértices de um polígono regular de n lados, inscrito numa circunferência de raio r centrada em p , cujo “primeiro” vértice faz um ângulo ϕ com o eixo dos X . Sendo um polígono regular um

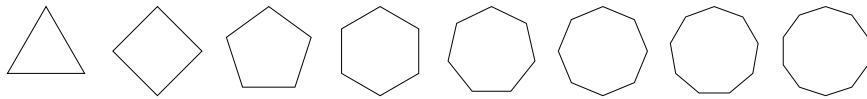


Figura 62: Polígonos regulares. Da esquerda para a direita temos um triângulo equilátero, um quadrado, um pentágono regular, um hexágono regular, um heptágono regular, um octógono regular, um eneágono regular e um decágono regular.

caso particular de uma estrela regular, basta-nos invocar a função que computa os vértices de uma estrela regular mas usando, para o parâmetro Δ_ϕ , apenas a divisão da circunferência 2π pelo número de lados n , i.e.:

```
(defun vertices-poligono-regular (p r fi n)
  (pontos-circulo p r fi (/ 2*pi n) n))
```

Finalmente, podemos encapsular a geração dos vértices com o seu uso para criar o polígono correspondente no AutoCad:

```
(defun poligono-regular (p r fi n)
  (poligono
    (vertices-poligono-regular p r fi n)))
```

8.2 Iteração em Listas

Até agora temos processado as listas usando apenas recursão. Sendo a lista um tipo de dados recursivo é natural que a forma mais simples de computar valores a partir de listas seja através de funções recursivas.

No entanto, existem casos, como o da função `passa-pontos`, em que o único resultado relevante da recursão é o conjunto de *efeitos secundários* que vão sendo realizados à medida que a recursão se vai desenrolando. De facto, se observarmos com atenção a definição da referida função constatamos que ocorre a seguinte redução:

$$\begin{array}{c} (\text{passa-pontos } l) \\ \downarrow \\ (\text{passa-pontos } (\text{cdr } l)) \end{array}$$

Contrariamente ao que acontecia nos casos tradicionais de recursão, neste nada é computado com o resultado da invocação recursiva, ou seja, a invocação recursiva nada deixa pendente. Esta forma de recursão denomina-se *iteração*.

Para estes casos em que apenas interessam os efeitos secundários do processamento dos vários elementos de uma lista o Auto Lisp providencia uma forma que simplifica substancialmente os programas: a forma `fforeach`. A sua sintaxe é a seguinte:

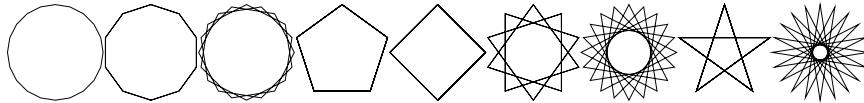


Figura 63: Estrelas regulares $\{\frac{p}{r}\}$ com $p = 20$ e r a variar desde 1 (à esquerda) até 9 (à direita).

```
(foreach nome expressão
  expressão1
  :
  expressãon)
```

A forma `foreach` recebe o nome de uma variável, uma expressão que deve avaliar para uma lista e várias expressões que serão avaliadas para cada elemento da lista, com o nome associado a esse elemento. Por exemplo, a seguinte expressão escreve todos os elementos de uma lista:

```
(foreach numero (list 1 2 3 4 5)
  (print numero))
```

Usando a forma `foreach` é agora mais fácil definir a função `passa-pontos`:

```
(defun passa-pontos (Pontos)
  (foreach ponto Pontos
    (command ponto)))
```

Na realidade, a função ficou tão simples que é preferível eliminá-la e juntar os vários comandos na função `poligono`:

```
(defun poligono (vertices)
  (command "_.line")
  (foreach vertice vertices
    (command vertice))
  (command "_close"))
```

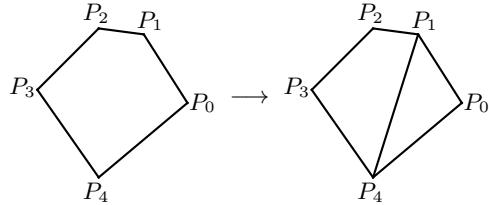
A iteração em listas é também muito útil para experimentarmos variações de figuras geométricas. Uma vez que a função enumera (definida na seção 6.6) gera uma sequência de números, podemos usar esses números em combinação com a forma `foreach` para desenhar uma sequência de figuras. Por exemplo, a seguinte expressão gera a Figura 63:

```
(foreach a (enumera 1 9 1)
  (poligono
    (vertices-estrela (xy (* 2 a) 0) 1 20 a)))
```

Exercício 8.1 Utilizando o operador `foreach`, defina a função denominada `rectangulo-envolvente`, que, dada uma lista de coordenadas bi-dimensionais, devolve uma lista com as coordenadas dos cantos inferior esquerdo e superior direito do menor retângulo capaz de incluir todas as coordenadas dadas. Por exemplo,

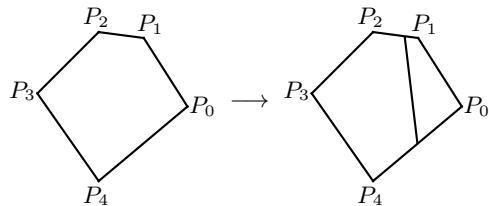
```
_§ (rectangulo-envolvente (list (xy 0 0) (xy 3 4) (xy -1 0) (xy 2 -3)))
((-1 -3) (3 4))
```

Exercicio 8.2 Considere um polígono representado pela lista dos seus vértices $(p_0 \ p_1 \ \dots \ p_n)$, tal como apresentamos no seguinte esquema, à esquerda:



Pretende-se dividir o polígono em dois sub-polígonos, tal como apresentado no esquema anterior, à direita. Defina a função `divide-poligono` que, dada a lista de vértices do polígono e dados dois índices i e j onde se deve fazer a divisão (com $i < j$), calcula as listas de vértices de cada um dos sub-polígonos resultantes e devolve-as numa lista. Por exemplo, no caso da imagem, temos que `(divide-poligono (list p0 p1 p2 p3 p4) 1 4)` produz a lista de listas `((p0 p1 p4) (p1 p2 p3 p4))`

Exercicio 8.3 Podemos considerar uma generalização do exercício anterior baseada na *bissecção* do polígono usando uma linha arbitrária, tal como podemos ver no esquema seguinte:



Para simplificar o processo, considere que cada extremidade da linha de bissecção está localizada a uma determinada fração $f_i \in [0, 1]$ da distância que vai do vértice P_i ao vértice seguinte P_{i+1} (com $P_{n+1} = P_0$). Por exemplo, no esquema anterior, os vértices em questão são P_1 e P_4 e a fracções são, respectivamente, de $f_1 = 0.3$ e $f_4 = 0.5$.

Defina a função `bisseccao-poligono` que, dada a lista de vértices do polígono e dados os dois índices i e j imediatamente anteriores aos extremos da linha de bissecção e dadas as fracções f_i e f_j da distância, respectivamente, entre os vértices (P_i, P_{i+1}) e entre (P_j, P_{j+1}) , calcula as listas de vértices de cada um dos sub-polígonos resultantes e devolve-as numa lista (i.e., devolve uma lista de listas de vértices).

Exercicio 8.4 Defina a função `bisseccao-aleatoria-poligono` que, dados os vértices de um polígono, devolve uma lista de listas de vértices dos sub-polígonos correspondentes a uma divisão aleatória do polígono. A seguinte imagem mostra exemplos de divisões aleatórias de um octógono.



Exercicio 8.5 Usando a função `bisseccao-aleatoria-poligono` definida no exercício anterior, defina a função `divisao-aleatoria-poligono` que, de forma aleatória, divide e subdivide recursivamente um polígono até que se atinja um determinado nível de recursão. A seguinte imagem mostra a divisão aleatória de um decágono para sucessivos níveis de recursão desde 0 até 10.



8.3 Linhas Poligonais e *Splines*

Vimos que as listas permitem armazenar um número variável de elementos e vimos como é possível usar listas para separar os programas que definem as coordenadas das figuras geométricas daqueles que usam essas coordenadas para as representarem graficamente.

No caso da função `poligono` discutida na secção 8.1, as listas de coordenadas são usadas para a criação de polígonos, i.e., figuras planas limitadas por um caminho fechado composto por uma sequência de segmentos de recta. Acontece que nem sempre queremos que as nossas figuras sejam limitadas por caminhos fechados, ou que esses caminhos sejam uma sequência de segmentos de recta ou, sequer, que as figuras sejam planas. Para resolver este problema, necessitamos de definir funções capazes de, a partir de listas de coordenadas, criarem outros tipos de figuras geométricas. Assim, vamos deixar de falar em vértices e vamos passar a falar genericamente em coordenadas dos pontos que definem as figuras geométricas.

O caso mais simples é o de uma linha poligonal (potencialmente aberta). Já vimos que o comando `line` permite criar linhas poligonais compostas por segmentos de recta independentes. Outra alternativa será não ter segmentos independentes mas sim uma única entidade geométrica. Para isso, podemos usar o comando `pline`:

```
(defun pline-pontos (pontos)
  (command "_.pline")
  (foreach p pontos (command p))
  (command ""))
```

Note-se que não usámos a opção "close" para não fecharmos a linha poligonal. Convém salientar que o comando `pline`, embora seja mais eficiente que o comando `line`, *não* consegue criar linhas poligonais que não sejam paralelas ao plano *XY*. No caso geral em que pretendemos linhas poligonais tridimensionais, este comando não pode ser usado e teremos de passar para uma variante menos eficiente mas mais versátil: o comando `3dpoly`. Para isso, vamos definir a função `3dpoly-pontos` de modo a usar linhas poligonais tridimensionais:

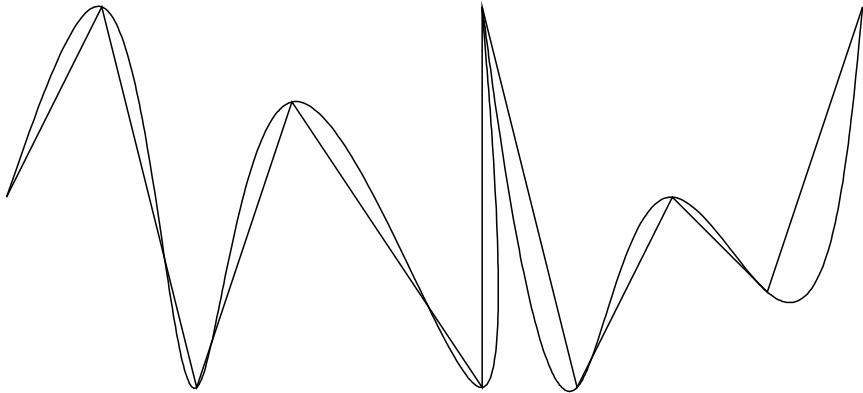


Figura 64: Comparação entre uma linha poligonal e uma *spline* que unem o mesmo conjunto de pontos.

```
(defun 3dpoly-pontos (pontos)
  (command "_.3dpoly")
  (foreach p pontos (command p))
  (command ""))
```

No caso de não pretendermos linhas poligonais mas sim curvas mais “suaves” podemos empregar uma *spline* que passe pelos pontos dados, através do comando *spline* do AutoCad:

```
(defun spline-pontos (pontos)
  (command "_.spline")
  (foreach p pontos (command p))
  (command " " " " "))
```

A diferença entre uma linha poligonal (produzida por *line*, *pline*, ou *3dpoly*) e uma *spline* é visível na Figura 64, onde comparamos uma sequência de pontos unida com uma linha poligonal com a mesma sequência unida com uma spline. O gráfico foi produzido pela avaliação das seguintes expressões:

```
(setq pontos
  (list (xy 0 2) (xy 1 4) (xy 2 0) (xy 3 3)
        (xy 5 0) (xy 5 4) (xy 6 0) (xy 7 2)
        (xy 8 1) (xy 9 4)))

(pline-pontos pontos)
(spline-pontos pontos)
```

Naturalmente, a especificação “manual” das coordenadas dos pontos é pouco conveniente, sendo preferível que essas coordenadas sejam computadas automaticamente de acordo com uma especificação matemática da

curva pretendida. Imaginemos, por exemplo, que pretendemos traçar uma curva sinusóide a partir de um ponto P . Infelizmente, de entre as figuras geométricas disponibilizadas pelo AutoCad—pontos, linhas rectas, rectângulos, polígonos, círculos, arcos de círculo, elipses, arcos de elipse, *donuts*, linhas poligonais e *splines*—não consta a sinusóide.

A alternativa é criarmos uma aproximação a uma sinusóide. Para isso, podemos calcular uma sequência de pontos pertencentes à curva da sinusóide e traçar rectas ou, ainda melhor, curvas que passem por esses pontos. Para calcular o conjunto de valores da função seno no intervalo $[x_0, x_1]$ basta-nos considerar um incremento Δ_x e, começando no ponto x_0 e passando do ponto x_i para o ponto x_{i+1} através de $x_{i+1} = x_i + \Delta_x$, vamos sucessivamente calculando o valor da expressão $\sin(x_i)$ até que x_i exceda x_1 . Para obtermos uma definição recursiva para este problema podemos pensar que quando $x_0 > x_1$ o resultado será uma lista vazia de coordenadas, caso contrário juntamos a coordenada $(x_0, \sin(x_0))$ à lista de coordenadas do seno para o intervalo $[x_0 + \Delta_x, x_1]$. Esta definição é traduzida directamente para a seguinte função:

```
(defun pontos-seno (x0 x1 dx)
  (if (> x0 x1)
      ()
      (list)
      (cons (xy x0 (sin x0))
            (pontos-seno (+ x0 dx) x1 dx))))
```

Para termos uma maior liberdade de posicionamento da curva sinusóide no espaço, podemos modificar a função anterior para incorporar um ponto P em relação ao qual se calcula a curva:

```
(defun pontos-seno (p x0 x1 dx)
  (if (> x0 x1)
      ()
      (list)
      (cons (+xy p x0 (sin x0))
            (pontos-seno p (+ x0 dx) x1 dx))))
```

A Figura 65 mostra as curvas traçadas pelas seguintes expressões que unem os pontos por intermédio de *plines*:

```
(pline-pontos (pontos-seno (xy 0.0 1.0) 0.0 6.0 1.0))
(pline-pontos (pontos-seno (xy 0.0 0.5) 0.0 6.5 0.5))
(pline-pontos (pontos-seno (xy 0.0 0.0) 0.0 6.4 0.2))
```

Note-se, na Figura 65, que demos um deslocamento vertical às curvas para melhor se perceber a diferença de precisão entre elas. Como é plenamente evidente, quantos mais pontos se usarem para calcular a curva, mais proxima será a linha poligonal da verdadeira curva. No entanto há também que ter em conta que o aumento do número de pontos obriga o AutoCad a maior esforço computacional.

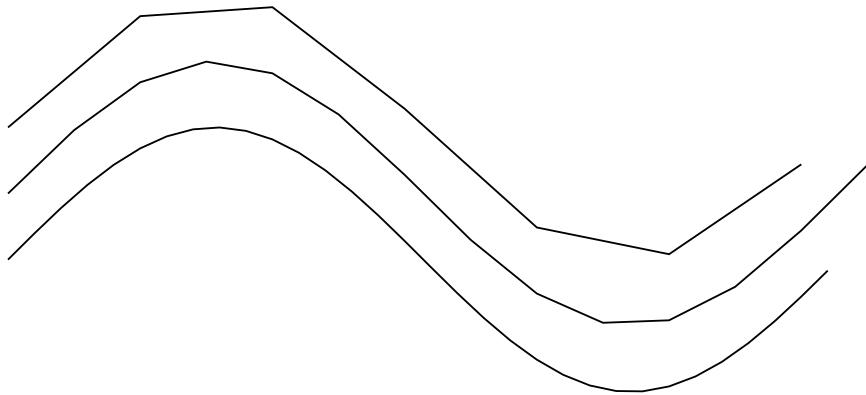


Figura 65: Senos desenhados usando *plines* com um número crescente de pontos.

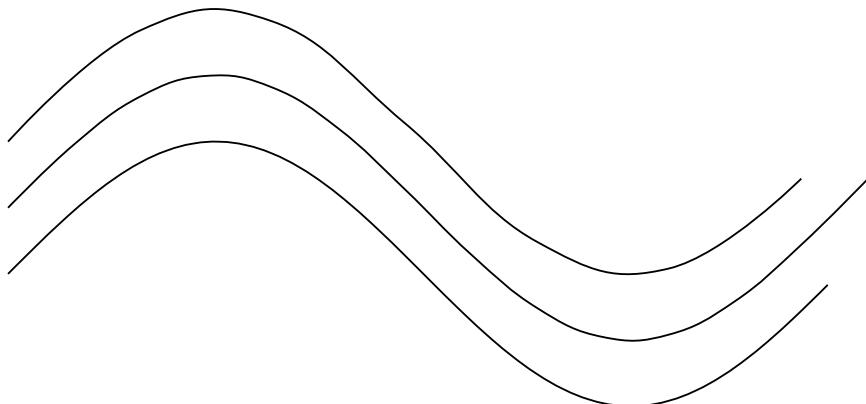
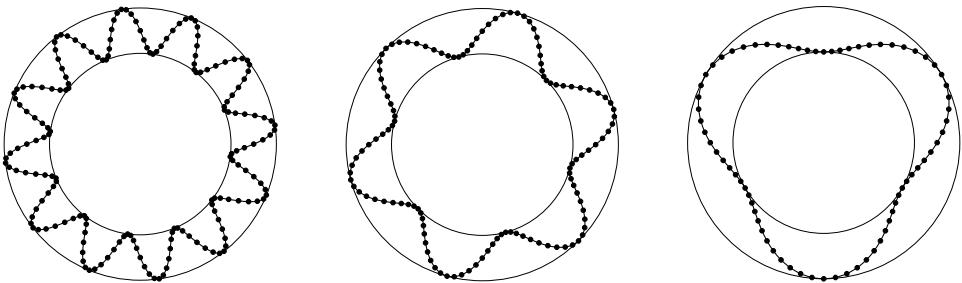


Figura 66: Senos desenhados usando *splines* com um número crescente de pontos.

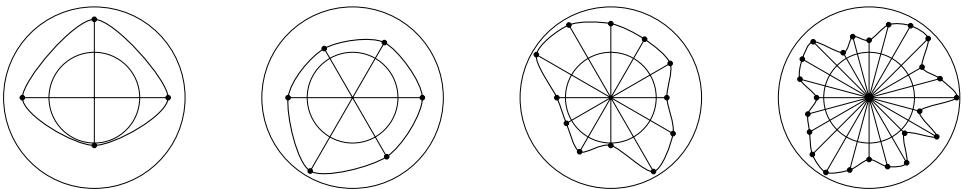
Para se obterem ainda melhores aproximações, embora à custa de ainda maior esforço computacional, podemos usar *splines*, simplesmente mudando as expressões anteriores para usarem a função `spline-pontos` no lugar de `pline-pontos`. O resultado está visível na Figura 66.

Exercício 8.6 Tal como as linhas poligonais podem dar lugar a polígonos, também as *splines* podem formar curvas fechadas. Para isso, basta indicar a opção `close` após o último ponto fornecido ao comando `spline`. Defina a função `spline-fechada-pontos` que recebe uma lista de coordenadas e cria uma spline fechada que passa por esses pontos:

Exercício 8.7 Defina a função `pontos-sinusoida-circular`, de parâmetros p, r_i, r_e, c e n que computa n pontos de uma curva fechada com a forma de uma sinusoide com c ciclos que se desenvolve num anel circular centrado no ponto p e delimitado pelos raios interior r_i e exterior r_e , tal como se pode ver nos vários exemplos apresentados na seguinte figura onde, da esquerda para a direita, o número de ciclos c é 12, 6, e 3.



Exercicio 8.8 Defina a função `pontos-circulo-raio-aleatorio`, de parâmetros p , r_0 , r_1 e n que computa n pontos de uma curva fechada de forma aleatória contida num anel circular centrado no ponto p e delimitado pelos raios interior r_i e exterior r_e , tal como se pode ver nos vários exemplos apresentados na seguinte figura onde, da esquerda para a direita, fomos aumentando progressivamente o número de pontos usados, assim aumentando a irregularidade da curva.



Sugestão: para a computação dos pontos, considere a utilização de coordenadas polares para distribuir uniformemente os pontos em torno de um círculo mas com a distância ao centro desse círculo a variar aleatoriamente entre r_i e r_e . A título de exemplo, considere que a curva mais à esquerda na figura anterior foi gerada pela expressão

```
(spline-fechada-pontos
  (pontos-circulo-raio-aleatorio
    (xy 0 0) 1 2 6))
```

8.4 Treliças

Uma treliça é uma estrutura composta por barras rígidas que se unem em nós, formando unidades triangulares. Sendo o triângulo o único polígono intrinsecamente estável, a utilização de triângulos convenientemente interligados permite que as treliças sejam estruturas indeformáveis. Apesar da simplicidade dos elementos triangulares, diferentes arranjos destes elementos permitem diferentes tipos de treliças.

É conhecido o uso de treliças desde a Grécia antiga, em que eram utilizadas para o suporte dos telhados. No século dezasseis, nos seus *Quatro Livros de Arquitectura*, Andrea Palladio ilustra pontes de treliças. No século dezanove, com o uso extensivo de metal e a necessidade de ultrapassar vãos cada vez maiores, inventaram-se vários tipos de treliças que se distinguem pelos diferentes arranjos de barras verticais, horizontais e diagonais e que, frequentemente, se denominam de acordo com os seus inventores. Temos assim treliças Pratt, treliças Howe, treliças Town, treliças Warren,



Figura 67: A esfera geodésica de Buckminster Füller. Fotografia de Glen Fraser.

etc. Nas últimas décadas as treliças começaram a ser intensivamente utilizadas como elemento artístico ou para a construção de superfícies elaboradas. No conjunto de exemplos mais famosos incluem-se a esfera geodésica de Buckminster Füller para a Exposição Universal de 1967, apresentada (reconstruída) na Figura 67 e as treliças em forma de banana de Nicolas Grimshaw para o terminal de Waterloo, visíveis na Figura 68.

As treliças apresentam um conjunto de propriedades que as tornam particularmente interessantes do ponto de vista arquitectónico:

- É possível construir treliças muito grandes a partir de elementos relativamente pequenos, facilitando a produção, transporte e erecção.
- Desde que as cargas sejam aplicadas apenas nos nós da treliça, as barras ficam apenas sujeitas a forças axiais, i.e., trabalham apenas à tracção ou à compressão, permitindo formas estruturais de grande eficiência.
- Uma vez que os elementos básicos de construção são barras e nós, é fácil adaptar as dimensões destes às cargas previstas, permitindo assim grande flexibilidade.

8.4.1 Desenho de Treliças

O passo fundamental para o desenho de treliças é a construção dos elementos triangulares fundamentais. Embora frequentemente se conside-



Figura 68: Treliças em forma de banana para o terminal de Waterloo, por Nicolas Grimshaw. Fotografia de Thomas Hayes.

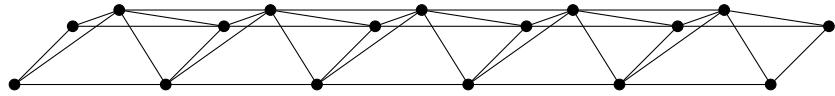


Figura 69: Treliça composta por elementos triangulares iguais.

rem apenas treliças bi-dimensionais (também chamadas *treliças planas*), iremos tratar o caso geral de uma treliça tri-dimensional composta por semi-octaedros. Esta forma de treliça denomina-se de *space frame*. Cada semi-octaedro é denominado de *módulo*.

A Figura 69 apresenta o esquema de uma treliça. Embora nesta figura os nós da treliça estejam igualmente espaçados ao longo de rectas paralelas, nada obriga a que assim seja. A Figura 70 mostra uma outra treliça em que tal não se verifica.

Assim, para o desenho de uma treliça, vamos considerar, como base de

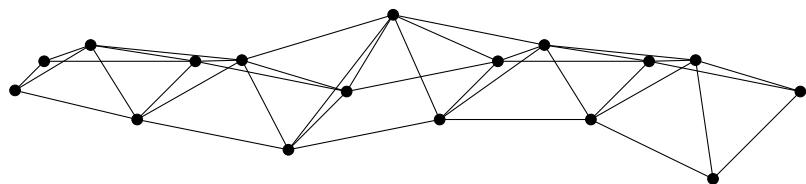


Figura 70: Treliça cujos elementos triangulares não são idênticos entre si.

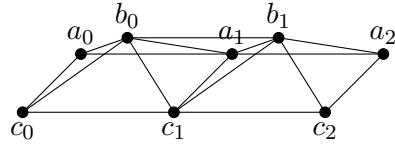


Figura 71: Esquema de ligação de barras de uma treliça em *space frame*.

trabalho, três sequências arbitrárias de pontos em que cada ponto define um nó da treliça. A partir destas três sequências podemos criar as ligações que é necessário estabelecer entre cada par de nós. A Figura 72 apresenta o esquema de ligação a partir de três sequências de pontos (a_0, a_1, a_2) , (b_0, b_1) e (c_0, c_1, c_2) . Note-se que a sequência de topo estabelecida pelos pontos b_i da sequência intermédia tem sempre menos um elemento que as sequências a_i e c_i .

Para a construção da treliça precisamos de encontrar um processo que, a partir das listas de pontos a_i , b_i e c_i , não só crie os nós correspondentes aos vários pontos, como os interligue da forma correcta. Comecemos por tratar da criação dos nós:

```
(defun nos-trelica (ps)
  (if (null ps)
    nil
    (progn
      (no-trelica (car ps))
      (nos-trelica (cdr ps)))))
```

ou, empregando o operador `foreach`,

```
(defun nos-trelica (ps)
  (foreach p ps
    (no-trelica p)))
```

A função `no-trelica` (notemos o singular, por oposição ao plural empregue na função `nos-trelica`) recebe as coordenadas de um ponto e é responsável por criar o modelo tridimensional que representa o nó da treliça centrado nesse ponto. Uma hipótese simples será esta função criar uma esfera onde se encaixarão as barras mas, por agora, vamos deixar a decisão sobre qual o modelo em concreto para mais tarde e vamos simplesmente admitir que a função `no-trelica` fará algo apropriado para criar o nó. Assim, podemos começar a idealizar a função que constrói a treliça completa a partir das listas de pontos `as`, `bs` e `cs`:

```
(defun trelica (as bs cs)
  (nos-trelica as)
  (nos-trelica bs)
  (nos-trelica cs)
  ...)
```

De seguida, vamos tratar de estabelecer as barras entre os nós. Da aná-

lise da Figura 72 ficamos a saber que temos uma ligação entre cada a_i e cada c_i , outra entre a_i e b_i , outra entre c_i e b_i , outra entre b_i e a_{i+1} , outra entre b_i e c_{i+1} , outra entre a_i e a_{i+1} , outra entre b_i e b_{i+1} e, finalmente, outra entre c_i e c_{i+1} . Admitindo que a função `barra-trelica` cria o modelo tridimensional dessa barra (por exemplo, um cilindro, ou uma barra prismática), podemos começar por definir uma função denominada `barras-trelica` (notemos o plural) que, dadas duas listas de pontos `ps` e `qs`, crie barras de ligação ao longo dos sucessivos pares de pontos. Para criar uma barra, a função necessita de um elemento dos `ps` e outro dos `qs`, o que implica que a função deve terminar assim que uma destas listas estiver vazia. A definição fica então:

```
(defun barras-trelica (ps qs)
  (if (or (null ps) (null qs))
      nil
      (progn
        (barra-trelica (car ps) (car qs))
        (barras-trelica (cdr ps) (cdr qs)))))
```

Para interligar cada nó a_i ao correspondente nó c_i , apenas temos de avaliar `(barras-trelica as cs)`. O mesmo poderemos dizer para interligar cada nó b_i ao nó a_i correspondente e para interligar cada b_i a cada c_i . Assim, temos:

```
(defun trelica (as bs cs)
  (nos-trelica as)
  (nos-trelica bs)
  (nos-trelica cs)
  (barras-trelica as cs)
  (barras-trelica bs as)
  (barras-trelica bs cs)
  ...)
```

Para ligar os nós b_i aos nós a_{i+1} podemos simplesmente subtrair o primeiro nó da lista `as` e estabelecer a ligação como anteriormente. O mesmo podemos fazer para ligar cada b_i a cada c_{i+1} . Finalmente, para ligar cada a_i a cada a_{i+1} podemos usar a mesma ideia mas aplicando-a apenas à lista `as`. O mesmo podemos fazer para a lista `cs`. A função completa fica, então:

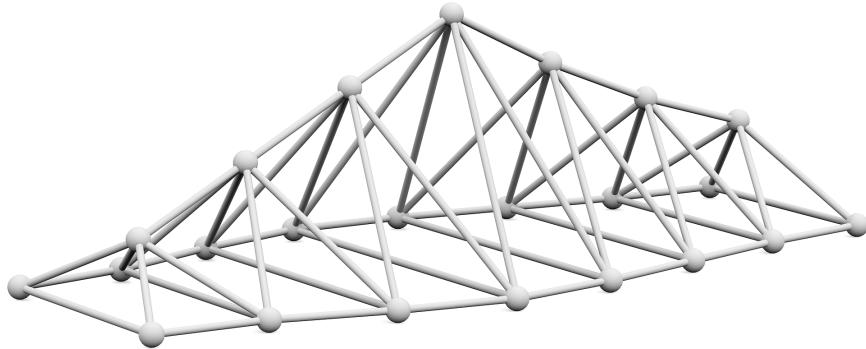


Figura 72: Treliça construída a partir de pontos especificados arbitrariamente.

```
(defun trelica (as bs cs)
  (nos-trelica as)
  (nos-trelica bs)
  (nos-trelica cs)
  (barras-trelica as cs)
  (barras-trelica bs as)
  (barras-trelica bs cs)
  (barras-trelica bs (cdr as))
  (barras-trelica bs (cdr cs))
  (barras-trelica as (cdr as))
  (barras-trelica cs (cdr cs))
  (barras-trelica bs (cdr bs)))
```

As funções anteriores constroem treliças com base nas funções “elementares” no-trelica e barra-trelica. Embora o seu significado seja óbvio, ainda não definimos estas funções e existem várias possibilidades. Numa primeira abordagem, vamos considerar que cada nó da treliça será constituído por uma esfera onde se irão unir as barras, barras essas que serão definidas por cilindros. O raio das esferas e da base dos cilindros será determinado por uma variável global, para que possamos facilmente alterar o seu valor. Assim, temos:

```
(setq raio-no-trelica 0.1)

(defun no-trelica (p)
  (esfera p raio-no-trelica))

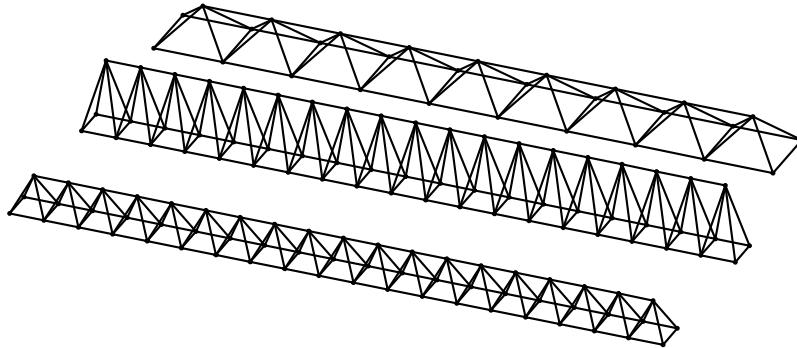
(setq raio-barra-trelica 0.03)

(defun barra-trelica (p0 p1)
  (cilindro p0 raio-barra-trelica p1))
```

Podemos agora criar as treliças com as formas que entendemos. A Figura 72 mostra uma treliça desenhada a partir da expressão:

```
(trelica
  (list (xyz 0 -1 0) (xyz 1 -1.1 0) (xyz 2 -1.4 0) (xyz 3 -1.6 0)
        (xyz 4 -1.5 0) (xyz 5 -1.3 0) (xyz 6 -1.1 0) (xyz 7 -1 0))
  (list (xyz 0.5 0 0.5) (xyz 1.5 0 1) (xyz 2.5 0 1.5) (xyz 3.5 0 2)
        (xyz 4.5 0 1.5) (xyz 5.5 0 1.1) (xyz 6.5 0 0.8)))
  (list (xyz 0 +1 0) (xyz 1 +1.1 0) (xyz 2 +1.4 0) (xyz 3 +1.6 0)
        (xyz 4 +1.5 0) (xyz 5 +1.3 0) (xyz 6 +1.1 0) (xyz 7 +1 0)))
```

Exercício 8.9 Defina uma função denominada `trelica-recta` capaz de construir qualquer uma das treliças que se apresentam na imagem seguinte.



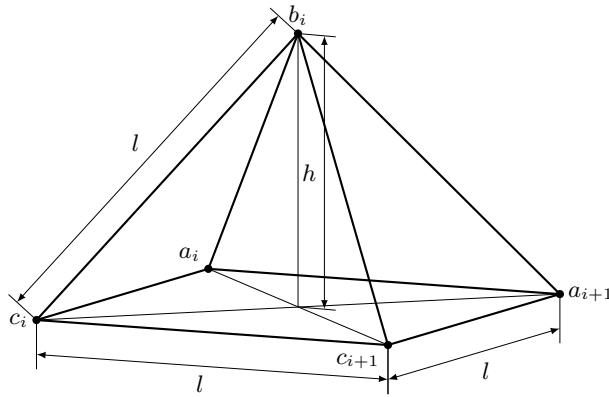
Para simplificar, considere que as treliças se desenvolvem segundo o eixo X . A função `trelica-recta` deverá receber o ponto inicial da treliça, a altura e largura da treliça e o número de nós das fileiras laterais. Com esses valores, a função deverá produzir três listas de coordenadas que passará como argumentos à função `trelica`. Como exemplo, considere que as três treliças apresentadas na imagem anterior foram o resultado da avaliação das expressões:

```
(trelica-recta (xyz 0 0 0) 1.0 1.0 20)
(trelica-recta (xyz 0 5 0) 2.0 1.0 20)
(trelica-recta (xyz 0 10 0) 1.0 2.0 10)
```

Sugestão: comece por definir a função `coordenadas-linha` que, dado um ponto inicial p , um afastamento l entre pontos e um número n de pontos, devolve uma lista com as coordenadas dos n pontos dispostos ao longo do eixo X .

Exercício 8.10 O custo total de uma treliça é muito dependente do número de diferentes comprimentos que as barras podem ter: quanto menor for esse número, maiores economias de escala se conseguem obter e, consequentemente, mais económica fica a treliça. O caso ideal é aquele em que existe um único comprimento igual para todas as barras.

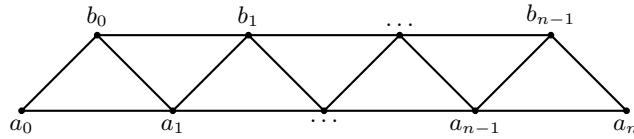
Atendendo ao seguinte esquema, determine a altura h da treliça em função da largura l do módulo de modo a que todas as barras tenham o mesmo comprimento.



Defina ainda a função `trelica-modulo` que constrói uma treliça com barras todas do mesmo comprimento, orientada segundo o eixo X. A função deverá receber o ponto inicial da treliça, a largura da treliça e o número de nós das fileiras laterais.

Exercício 8.11 Dadas as coordenadas dos quatro vértices da base de uma pirâmide quadrangular e a altura dessa pirâmide, é possível calcular as coordenadas do vértice do topo dessa pirâmide determinando o centro da base e o vector normal à base.

Exercício 8.12 Considere o desenho de uma treliça plana, tal como se apresenta na seguinte figura:

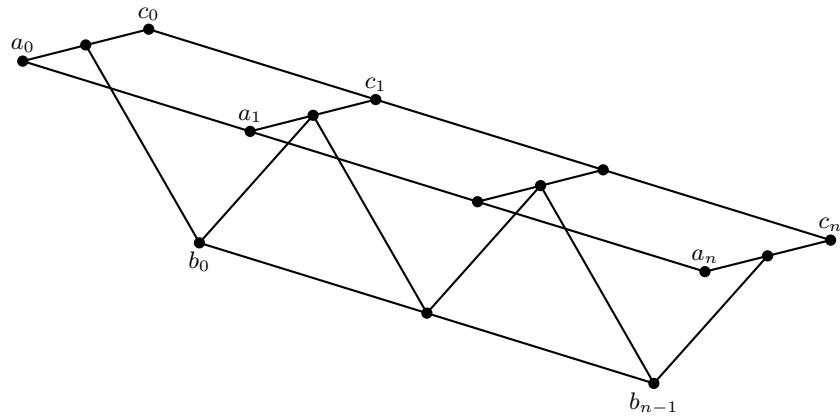


Defina uma função `trelica-plana` que recebe, como parâmetros, duas listas de pontos correspondentes aos pontos desde a_0 até a_n e desde b_0 até b_{n-1} e que cria os nós nesses pontos e as barras que os unem. Considere, como pré-definidas, as funções `nos-trelica`, que recebe uma lista de pontos como argumento e `barras-trelica` que recebe duas listas de pontos como argumentos.

Teste a função de definir com a seguinte expressão:

```
(trelica-plana
  (coordenadas-linha (xyz 0 0 0) 2.0 20)
  (coordenadas-linha (xyz 1 0 1) 2.0 19))
```

Exercício 8.13 Considere o desenho da treliça especial apresentada na seguinte figura:



Defina uma função `trelica-especial` que recebe, como parâmetros, três listas de pontos correspondentes aos pontos desde a_0 até a_n , desde b_0 até b_{n-1} e desde c_0 até c_n e que cria os nós nesses pontos e as barras que os unem. Considere, como pré-definidas, as funções `nos-trelica` que recebe uma lista de ponto como argumento e `barras-trelica` que recebe duas listas de pontos como argumentos.

8.4.2 Geração de Coordenadas

Como vimos na secção anterior, podemos idealizar um processo de criação de uma treliça a partir das listas de coordenadas dos seus nós. Estas listas, naturalmente, podem ser especificadas manualmente mas esta abordagem só será realizável para treliças muito pequenas. Ora sendo uma treliça uma estrutura capaz de vencer vãos muito grandes, no caso geral, o número de nós da treliça é demasiado elevado para que possamos produzir manualmente as suas listas de coordenadas. Para resolver este problema, temos de pensar em processos automatizados para criar essas listas, processos esses que tenham em conta a geometria pretendida para a treliça.

A título de exemplo, idealizemos um processo de criação de treliças em arco, em que as sequências de nós a_i , b_i e c_i formam arcos de circunferência. A Figura 73 mostra uma versão de uma destas treliças, definida pelos arcos de circunferência de raio r_0 e r_1 .

Para tornar a treliça uniforme, os nós encontram-se igualmente espaçados ao longo do arco. O ângulo Δ_ψ corresponde a esse espaçamento e calcula-se trivialmente pela divisão da amplitude angular do arco pelo número de nós pretendidos n . Atendendo a que o arco intermédio tem sempre menos um nó do que os arcos laterais, temos de dividir o ângulo Δ_ψ pelas duas extremidades do arco intermédio, de modo a centrar os nós desse arco entre os nós dos arcos laterais, tal como se pode verificar na Figura 73.

Uma vez que o arco é circular, a forma mais simples de calcularmos as posições dos nós será empregando coordenadas esféricas (ρ, ϕ, ψ) . Esta decisão leva-nos a considerar que os ângulos inicial e final dos arcos devem ser medidos relativamente ao eixo Z , tal como é visível na Figura 73. Para

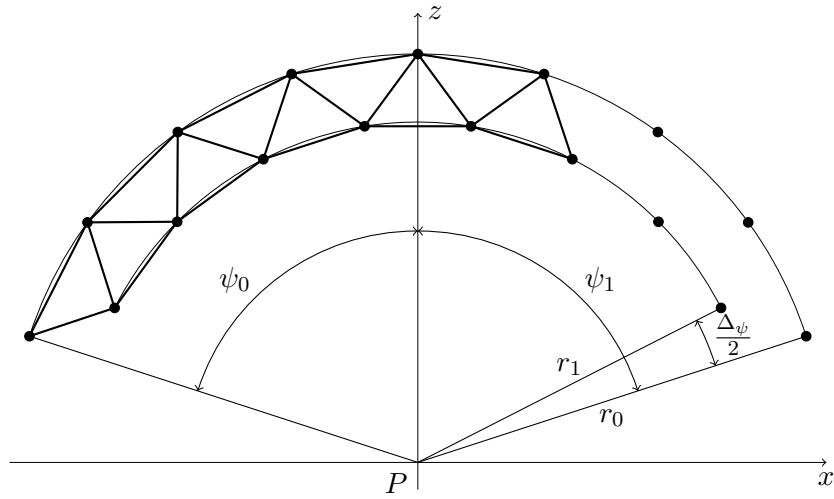


Figura 73: Alçado frontal de uma treliça em forma de arco de círculo.

flexibilizar a produção das coordenadas dos nós do arco vamos definir uma função que recebe o centro P do arco, o raio r desse arco, o ângulo ϕ , os ângulos inicial ψ_0 e final ψ_1 e, finalmente, o incremento de ângulo Δ_ψ . Assim, temos:

```
(defun pontos-arco (p r fi psi0 psil dpsi)
  (if (> psi0 psil)
      (list)
      (cons (+esf p r fi psi0)
            (pontos-arco p r fi (+ psi0 dpsi) psil dpsi))))
```

Para construirmos a treliça em arco podemos agora definir uma função que cria três dos arcos anteriores. Para isso, a função terá de receber o centro P do arco central, o raio r_{ac} dos arcos laterais, o raio r_b do arco central, o ângulo ϕ , os ângulos inicial ψ_0 e final ψ_1 e, ainda, a separação e entre os arcos laterais e o número n de nós dos arcos laterais. A função irá calcular o incremento $\Delta_\psi = \frac{\psi_1 - \psi_0}{n}$ e, de seguida, invoca a função `trelica` com os parâmetros apropriados:

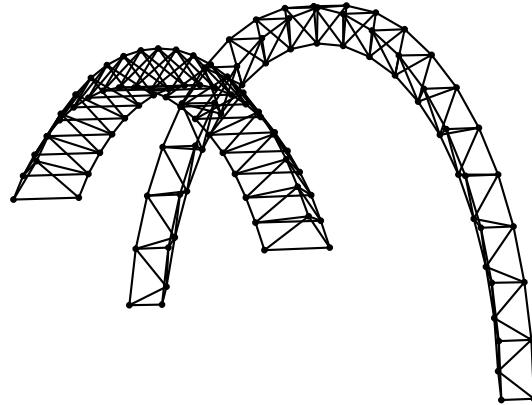


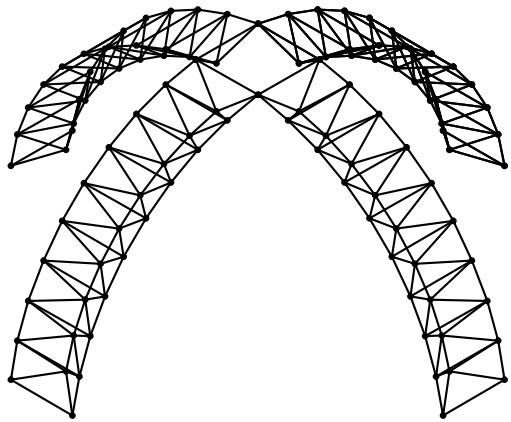
Figura 74: Treliças em arco criadas com parâmetros diferentes.

```
(defun trelica-arco (p rac rb fi psi0 psil e n / dpsi)
  (setq dpsi (/ (- psil psi0) n))
  (trelica
    (pontos-arco (+pol p (/ e 2.0) (+ fi pi/2))
      rac
      fi
      psi0 psil
      dpsi)
    (pontos-arco p
      rb
      fi
      (+ psi0 (/ dpsi 2.0)) (- psil (/ dpsi 2.0))
      dpsi)
    (pontos-arco (+pol p (/ e 2.0) (- fi pi/2))
      rac
      fi
      psi0 psil
      dpsi)))
  
```

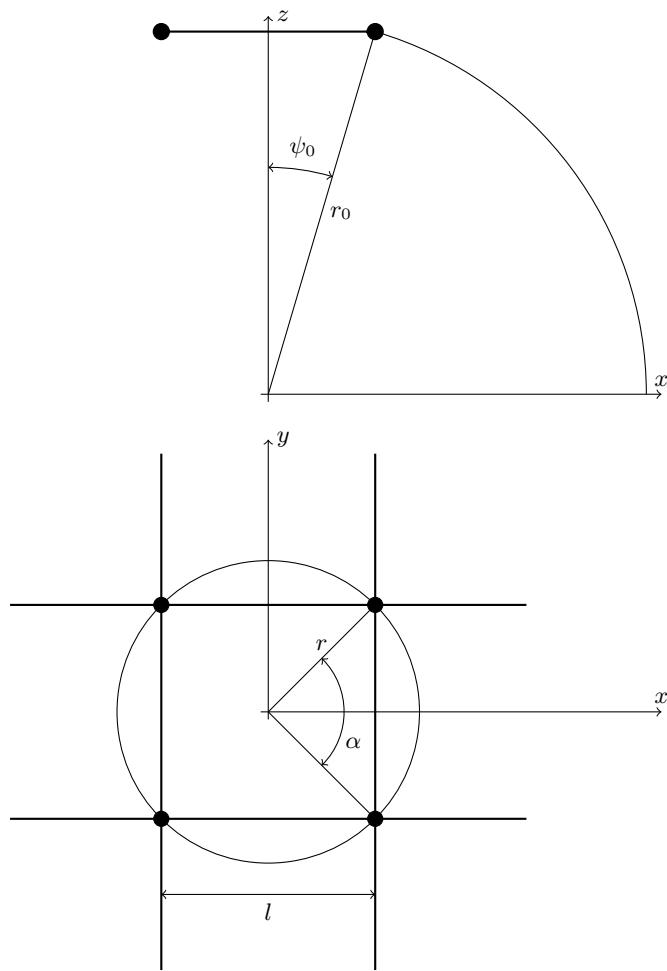
A Figura 74 mostra as treliças construídas a partir das expressões:

```
(trelica-arco (xyz 0 0 0) 10 9 0 -pi/2 pi/2 1.0 20)
(trelica-arco (xyz 0 5 0) 8 9 0 -pi/3 pi/3 2.0 20)
```

Exercicio 8.14 Considere a construção de abóbadas apoiadas em treliças distribuidas radialmente, tal como a que se apresenta na imagem seguinte:



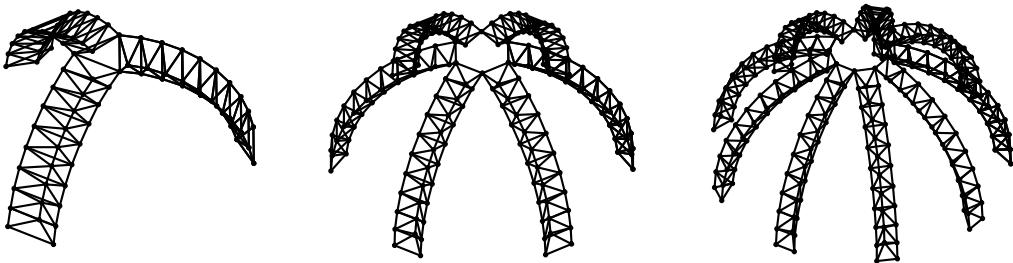
Esta abóbada é constituída por um determinado número de tréliças de arco circular. A largura l de cada trélica e o ângulo inicial ψ_0 a que se dá o arranque de cada trélica são tais que os nós dos topos das tréliças são coincidentes dois a dois e estão dispostos ao longo de um círculo de raio r , tal como se apresenta no esquema seguinte:



Defina a função `abobada-trelicas` que constrói uma abóbada de trelicas a partir do centro da abóbada P , do raio r_{ac} dos arcos laterais de cada treliça, do raio r_b do arco central de cada treliça, do raio r do “fecho” das treliças, do número de nós n em cada treliça e, finalmente, do número de treliças n_ϕ .

A título de exemplo, considere a figura seguinte que foi produzida pela avaliação das seguintes expressões:

```
(abobada-trelicas (xyz 0 0 0) 10 9 2.0 10 3)
(abobada-trelicas (xyz 25 0 0) 10 9 2.0 10 6)
(abobada-trelicas (xyz 50 0 0) 10 9 2.0 10 9)
```



8.4.3 Treliças Espaciais

Vimos como é possível definir treliças a partir de três listas cada uma contendo as coordenadas dos nós a que as barras das treliças se ligam. Ligando entre si várias destas treliças é possível produzir uma estrutura ainda maior a que se dá o nome de *treliça espacial*. A Figura 75 mostra um exemplo onde são visíveis três treliças espaciais.

Para podermos definir um algoritmo que gere treliças espaciais é importante termos em conta que embora este tipo de treliças aglomere várias treliças simples, estas estão interligadas de tal modo que cada treliça partilha um conjunto de nós e barras com a treliça que lhe é adjacente, tal como é visível na Figura 76 que apresenta um esquema de uma treliça espacial. Assim, se uma treliça espacial for constituída por duas treliças simples interligadas, a treliça espacial será gerada, não por seis listas de coordenadas, mas apenas por cinco listas de coordenadas. No caso geral, uma treliça espacial constituída por n treliças simples, será definida por um $2n + 1$ listas de pontos, i.e., por um número ímpar de listas de pontos (no mínimo, três listas).

A definição da função que contrói uma treliça espacial segue a mesma lógica da função que constrói uma treliça simples só que agora, em vez de operar com apenas três listas, opera com um número ímpar delas. Assim, a partir de uma lista contendo um número ímpar de listas de coordenadas, iremos processar essas listas de coordenadas duas a duas, sabendo que a “terceira” lista de coordenadas $c_{i,j}$ da treliça i é também a “primeira” lista de coordenadas $a_{i+1,j}$ da treliça seguinte $i + 1$.



Figura 75: Treliças espaciais no estádio Al Ain nos Emirados Árabes Unidos.
Fotografia de Klaus Knebel.

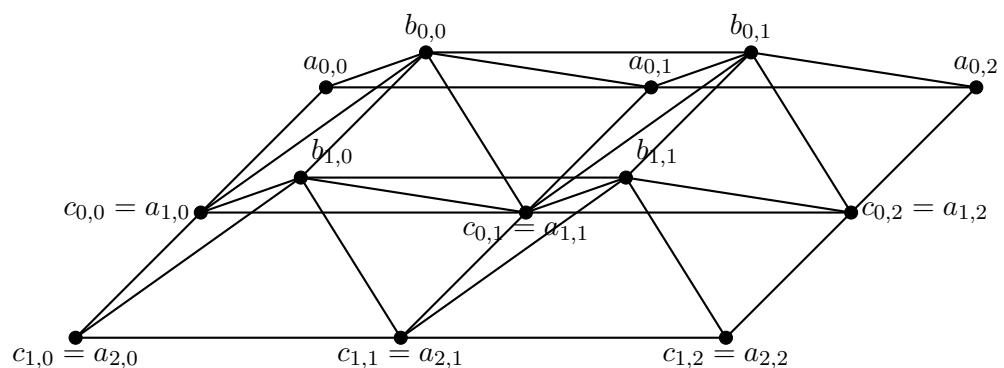


Figura 76: Esquema de ligação de barras de uma treliça espacial.

Uma vez que processamos duas listas de cada vez e partimos de um número ímpar de listas, no caso “final” restará apenas uma lista de coordenadas que deverá “fechar” a construção da treliça.

Há, no entanto, uma dificuldade adicional: para que a treliça plana tenha rigidez transversal é ainda necessário interligar entre si os nós centrais das várias treliças. Estes travamentos correspondem a ligar cada nó $b_{i,j}$ ao nó $b_{i+1,j}$. Assim, à medida que formos processando as listas de coordenadas, iremos também estabelecer os travamentos entre as listas correspondentes. Todo este processo é implementado pela seguinte função:

```
(defun trelica-espacial (curvas / as bs cs)
  (setq as (car curvas)
        bs (cadr curvas)
        cs (caddr curvas))
  (nos-trelica as)
  (nos-trelica bs)
  (barras-trelica as cs)
  (barras-trelica bs as)
  (barras-trelica bs cs)
  (barras-trelica bs (cdr as))
  (barras-trelica bs (cdr cs))
  (barras-trelica (cdr as) as)
  (barras-trelica (cdr bs) bs)
  (if (null (cdddr curvas))
      (progn
        (nos-trelica cs)
        (barras-trelica (cdr cs) cs))
      (progn
        (barras-trelica bs (cadddr curvas))
        (trelica-espacial (cddr curvas))))
```

Exercício 8.15 Na realidade, uma treliça simples é um caso particular de uma treliça espacial. Redefina a função `trelica` de modo a que ela use a função `trelica-espacial`.

Agora que já sabemos construir treliças espaciais a partir de uma lista de listas de coordenadas, podemos pensar em mecanismos para gerar esta lista de listas. Um exemplo simples é o de uma treliça espacial horizontal, tal como a que se apresenta na Figura 77.

Para gerar as coordenadas dos nós desta treliça, podemos definir uma função que, com base no número de pirâmides pretendidas e na largura da base da pirâmide, gera os nós ao longo de uma das dimensões, por exemplo, a dimensão X :

```
(defun coordenadas-linha (p l n)
  (if (= n 0)
      (list p)
      (cons p
            (coordenadas-linha (+x p 1) l (- n 1)))))
```

Em seguida, basta-nos definir uma outra função que itera a anterior ao longo da outra dimensão Y , de modo a gerar uma linha de nós a_i , seguida

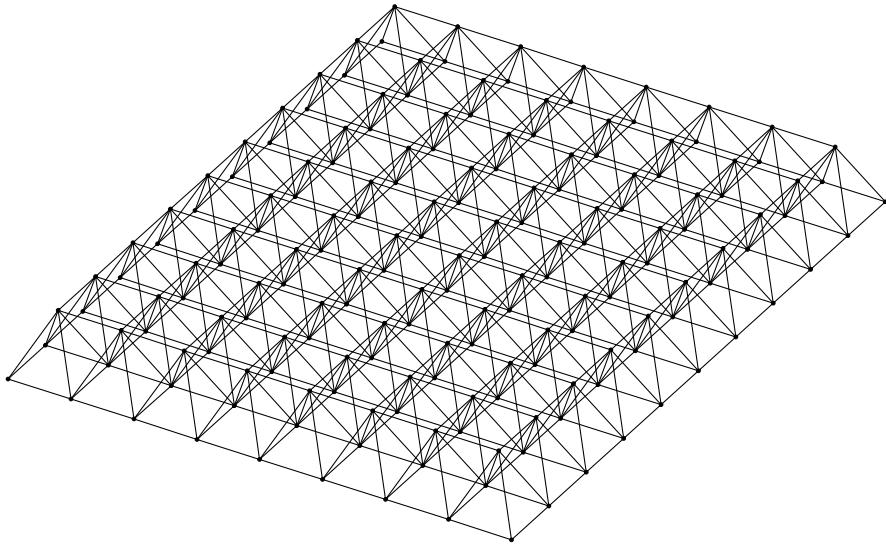


Figura 77: Uma treliça espacial horizontal, composta por oito treliças simples com dez pirâmides cada uma.

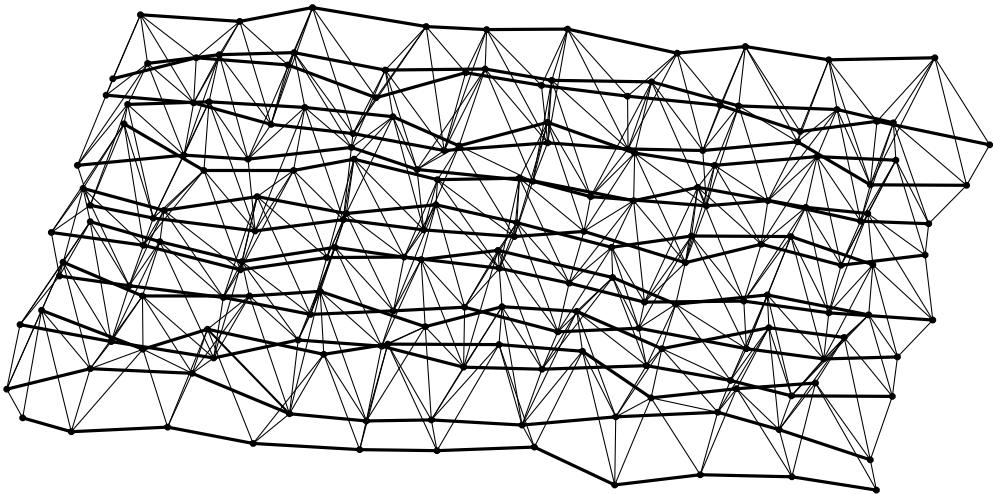
de outra linha b_i deslocada para o centro da pirâmide e à altura desta, seguida das restantes linhas, até o final, em que teremos de produzir mais uma linha a_i . É ainda necessário termos em conta que as linhas b_i têm menos um nó do que as linhas a_i . Com base nestas considerações, podemos escrever:

```
(defun coordenadas-piramides-horizontais (p h l m n)
  (if (= m 0)
    (list (coordenadas-linha p l n))
    (cons
      (coordenadas-linha p l n)
      (cons
        (coordenadas-linha
          (+xyz p (/ l 2.0) (/ l 2.0) h) l (- n 1))
        (coordenadas-piramides-horizontais
          (+y p 1) h l (- m 1) n))))
```

Podemos agora combinar a lista de listas de coordenadas produzidas pela função anterior com a que constrói uma treliça espacial. A título de exemplo, a seguinte expressão produz a treliça apresentada na Figura 77:

```
(trelica-espacial
  (coordenadas-piramides-horizontais (xyz 0 0 0) 1 1 8 10))
```

Exercício 8.16 Considere a construção de uma treliça espacial *aleatória*. Esta treliça caracteriza-se por as coordenadas dos seus nós estarem posicionados a uma distância aleatória das coordenadas dos nós correspondentes de uma treliça espacial horizontal, tal como é exemplificado pela seguinte figura onde, para facilitar a visualização, se assinalou a traço mais grosso as barras que unem os nós a_i , b_i e c_i do esquema apresentado na Figura 76.

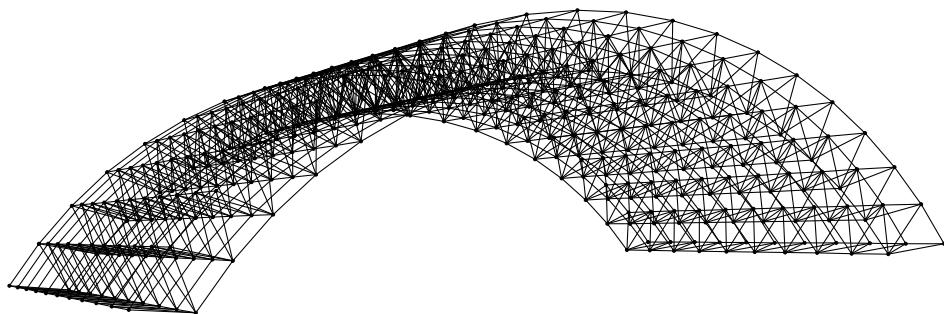


Defina a função `coordenadas-trelica-aleatoria` que, para além dos parâmetros da função `coordenadas-piramides-horizontais`, recebe ainda a distância máxima r a que cada nó da treliça aleatória pode ser colocado relativamente ao nó correspondente da treliça horizontal. A título de exemplo, considere que a treliça apresentada na figura anterior foi gerada pela avaliação da expressão:

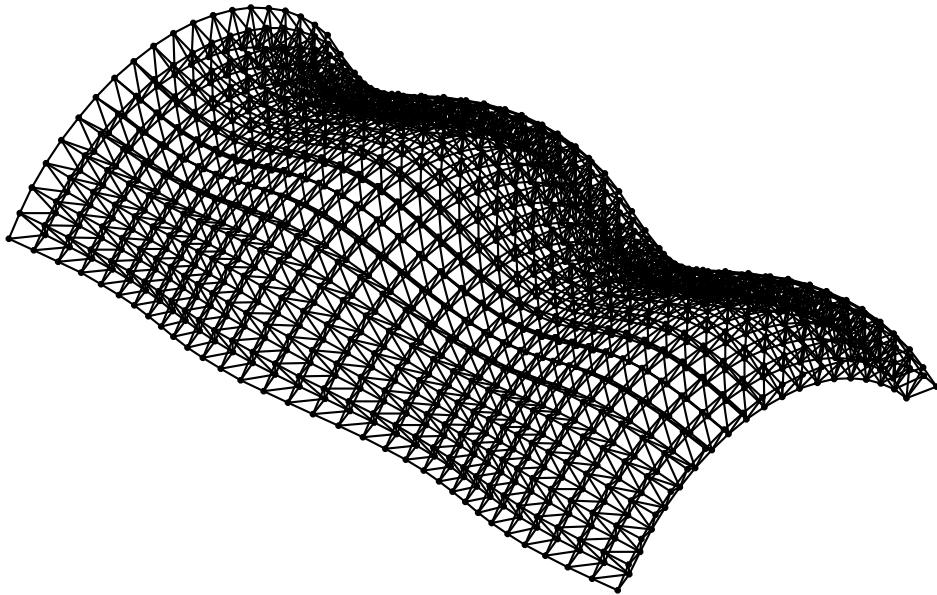
```
(trelica-espacial
  (coordenadas-trelica-aleatoria (xyz 0 0 0) 1 1 8 10 0.2))
```

Exercicio 8.17 Considere a construção de uma treliça espacial em arco, tal como a que se apresenta na imagem seguinte (em perspectiva). Defina a função `trelica-espacial-arco`, que, para além dos parâmetros da função `trelica-arco`, possui ainda um parâmetro adicional que indica o número de treliças simples que constituem a treliça espacial. A título de exemplo, considere que a treliça apresentada na imagem seguinte foi gerada pela expressão:

```
(trelica-espacial-arco (xyz 0 0 0) 10 9 1.0 20 0 -pi/3 pi/3 10)
```



Exercicio 8.18 Considere a treliça apresentada na imagem seguinte:



Esta treliça é semelhante à treliça espacial em arco mas com uma *nuance*: os raios exterior rac e interior rb variam ao longo do eixo do arco. Esta variação corresponde a uma sinusoide de amplitude Δ_r a variar desde um valor inicial α_0 até um valor final α_1 , em incrementos de Δ_α .

Defina a função `trelica-ondulada` que constrói este tipo de treliças, a partir dos mesmos parâmetros da função `trelica-espacial-arco` e ainda dos parâmetros α_0 , α_1 , Δ_α e Δ_r . Como exemplo, considere que a figura anterior foi gerada pela invocação seguinte:

```
(trelica-ondulada (xyz 0 0 0) 10 9 1.0 20 0 -pi/3 pi/3 0 4*pi (/ pi 8) 1)
```

9 Manipulação de Entidades

O AutoCad não é só um programa de desenho. Ele é também uma base de dados sobre figuras geométricas. De facto, sempre que desenhamos algo o AutoCad regista na sua base de dados a entidade gráfica criada, bem como algumas informações adicionais relacionadas com essa entidade como, por exemplo, a sua cor.

Existem várias maneiras de se aceder às entidades criadas. Uma das mais simples para um utilizador “normal” do AutoCad será através do uso do rato, simplesmente “clicando” na entidade gráfica a que pretendemos aceder. Outra, mais útil para quem pretende programar, será através da invocação de funções do Auto Lisp que devolvem, como resultados, as entidades geométricas criadas. Há várias destas funções à nossa disposição mas, por agora, vamos limitar-nos a uma das mais simples: a função `entlast`.

A função `entlast` não recebe quaisquer argumentos e devolve a *última* entidade criada no AutoCad. Para além de registar todas as entidades

geométricas criadas durante uma sessão de desenho, o AutoCad mantém ainda um registo (que vai sendo continuamente actualizado) da última dessas entidades, de modo a que o programador a ela possa aceder simplesmente invocando a função `entlast`. Se ainda não tiver sido criada nenhuma entidade, a função devolve `nil`.

A seguinte interacção demonstra o comportamento desta função após a criação de um círculo:

```
_\$ (command "_circle" (xy 10 20) 30)
nil
_\$ (entlast)
<Entity name: 7ef91ed8>
```

O valor retornado pela avaliação de `(entlast)` tem, como representação externa, o texto `<Entity name: ???>`, em que `???` é um *identificador* da entidade, destinado apenas a permitir ao leitor distinguir diferentes entidades entre si pois cada entidade terá necessariamente um identificador diferente.

Dada uma entidade, podemos estar interessados em conhecer as suas *propriedades*. Para isso, o Auto Lisp disponibiliza a função `entget`. Esta função recebe uma entidade como argumento e devolve uma lista com as propriedades da entidade. Eis um exemplo:⁵⁶

```
_\$ (entget (entlast))
((-1 . <Entity name: 7ef91ed8>)
 (0 . "CIRCLE")
 (330 . <Entity name: 7ef91db8>)
 (5 . "5B")
 (100 . "AcDbEntity")
 (67 . 0)
 (410 . "Model")
 (8 . "0")
 (100 . "AcDbCircle")
 (10 10.0 20.0 0.0)
 (40 . 30.0)
 (210 0.0 0.0 1.0))
```

Como podemos ver, o valor devolvido é uma lista de *pares*. Isto é válido mesmo para os último e o antepenúltimo elementos da lista, i.e., `(210 0.0 0.0 1.0)` e `(10 10.0 20.0 0.0)`, que não aparecem ter a forma de pares simplesmente porque o segundo elemento de cada um desses pares é, ele próprio uma lista. Como já vimos aquando da relação entre pares e listas, aqueles elementos, na realidade, correspondem, respectivamente, aos pares `(210 . (0.0 0.0 1.0))` e `(10 . (10.0 20.0 0.0))`.

⁵⁶Indentámos o resultado da avaliação para melhor se compreender o valor devolvido.

Código	Significado
-1	Nome (uma <i>string</i> que pode mudar de sessão para sessão)
0	Tipo
1	Texto primário
2	Nome dado pelo utilizador (presente apenas nalgumas entidades)
3-4	Outros textos (presente apenas nalgumas entidades)
5	Referência(uma <i>string</i> que nunca muda)
6	Tipo de linha (presente apenas nalgumas entidades)
7	Estilo de texto (presente apenas nalgumas entidades)
8	Nome do <i>layer</i>
10	Ponto primário (o centro para um círculo, ponto inicial para uma linha, etc)
11-18	Outros pontos (presente apenas nalgumas entidades)
38	Elevação (apenas para entidades com elevação diferente de zero)
39	Espessura (apenas para entidades com espessura diferente de zero)
48	Escala do tipo de linha (presente apenas nalgumas entidades)
50-58	Ângulos
60	Visibilidade (opcional, 0 para visível, 1 para invisível)
62	Índice da cor
100	Marcador de subclasse
210	Direcção de extrusão

Tabela 5: Alguns dos códigos DXF

9.1 Listas de Associações

Este tipo de lista cujos elementos são pares denomina-se uma *lista de associações* ou, abreviadamente, uma *assoc-list* ou, ainda mais abreviadamente, uma *alist*.

No resultado devolvido pela última expressão há alguns elementos facilmente relacionáveis com o círculo que tínhamos criado, nomeadamente, a *string* "CIRCLE", as coordenadas (10.0 20.0 0.0) e o raio 30. Menos claro é o significado dos números que aparecem como primeiro elemento de cada par. Esses números designam, na realidade, uma codificação numérica da propriedade correspondente e encontram-se definidos pela especificação DXF (*Drawing eXchange Format*) inventada pela AutoDesk para permitir troca de informação entre aplicações de CAD. Assim, o número 0 designa o *tipo* da entidade, o número 10 designa o *ponto primário* da entidade (o centro do círculo, neste caso), etc. A Tabela 5 descreve o significado de alguns dos códigos definidos pela especificação DXF.

Quando se pretende saber apenas uma das propriedades de uma entidade, por exemplo, o seu tipo, é pouco prático ter de a procurar visualmente numa lista razoavelmente grande como a que resulta de `entget`. Note-se que não basta aceder à posição onde julgamos estar a propriedade pois essa posição não é a mesma para todos os objectos (pois nem todos os objectos possuem as mesmas propriedades) nem é necessariamente a mesma para diferentes versões do AutoCad. Naturalmente, sendo o resultado da função `entget` uma lista de pares, nada nos impede de definir uma função que processe essa lista à procura da propriedade em questão.

Para concretizarmos, consideremos que pretendemos definir uma função que, a partir do código DXF de uma propriedade que procuramos e da lista de propriedades de uma entidade, nos diz qual é o seu valor para aquela entidade. Para isso, o mais simples é aplicarmos recursão:

- Se o primeiro elemento do primeiro par da lista tem o código DXF que procuramos então o valor correspondente é o segundo elemento desse par.
- Se o primeiro elemento do primeiro par da lista não tem o código DXF pretendido, então podemos recursivamente procurar no resto da lista.

Traduzindo para Auto Lisp, temos:

```
(defun valor-propriedade (propriedade lista-assoc)
  (cond ((= propriedade (car (car lista-assoc)))
         (cdr (car lista-assoc)))
        (t
         (valor-propriedade propriedade (cdr lista-assoc))))))
```

Um pequeno teste assegura-nos que a função tem o comportamento desejado:

```
_ $ (valor-propriedade 0 (entget (entlast)))
"IRCLE"
```

A função `valor-propriedade`, infelizmente, contém um problema: tal como se pode ver na Tabela 5, nem todas as propriedades existem para todas as entidades. Isto implica que é possível a função `valor-propriedade` ser invocada para procurar uma propriedade que não existe numa dada lista de associações. Neste caso, a função teria um comportamento incorrecto pois, uma vez que as invocações recursivas vão operando sobre uma lista de propriedades sucessivamente mais pequena, chegará o momento em que essa lista fica vazia. Como a função não detecta essa situação continuará a tentar aceder ao primeiro elemento da lista, numa altura em que a lista já não tem quaisquer elementos. Para resolver este problema basta

detectarmos este caso e, caso ocorra, devolver um valor que signifique que a propriedade não existe naquela lista. Em geral, a linguagem Lisp convenciona que, para essas situações, o valor a devolver é `nil`. Tendo em conta este raciocínio, a função fica então com a seguinte forma:

```
(defun valor-propriedade (propriedade lista-assoc)
  (cond ((null lista-assoc)
         nil)
        ((= propriedade (car (car lista-assoc)))
         (cdr (car lista-assoc)))
        (t
         (valor-propriedade propriedade (cdr lista-assoc)))))

_ $ (valor-propriedade 0 (entget (entlast)))
"IRCLE"
_ $ (valor-propriedade 10 (entget (entlast)))
(10.0 20.0 0.0)
_ $ (valor-propriedade 123456789 (entget (entlast)))
nil
```

9.2 A função assoc

A função `valor-propriedade` é tão útil que até já se encontra pré-definida em Auto Lisp com o nome `assoc` mas com uma pequena diferença destinada a facilitar a distinção entre *não* existir uma propriedade numa lista de associações e existir essa propriedade mas com o valor `nil`: ao invés de devolver o valor da propriedade, a função `assoc` devolve o par propriedade/valor, quando existe, ou `nil` quando não existe. Como um par é necessariamente diferente de `nil`, é então possível distinguir os dois casos. O reverso da moeda é que, para aceder ao valor da propriedade torna-se ainda necessário usar a função `cdr`. Este comportamento é visível no seguinte exemplo:

```
_ $ (assoc 0 (entget (entlast)))
(0 . "IRCLE")
_ $ (cdr (assoc 0 (entget (entlast))))
"IRCLE"
```

Finalmente, podemos usar esta função como base para a construção de selectores que, ao invés de operarem com a lista de propriedades de uma entidade, operam sobre a própria entidade:

```
(defun tipo-entidade (entidade)
  (cdr (assoc 0 (entget entidade)))))

(defun ponto-primario (entidade)
  (cdr (assoc 10 (entget entidade)))))

_ $ (tipo-entidade (entlast))
"IRCLE"
_ $ (ponto-primario (entlast))
(10.0 20.0 0.0)
```

9.3 Modificação de Entidades

Vimos como aceder às propriedades de uma entidade. Vamos agora discutir a forma de modificarmos essas propriedades.

A função `entmod` permite a modificação de entidades. Para isso, a função recebe como argumento uma lista de associações contendo, para além dos pares propriedade/valor que pretendemos modificar, um par nome/entidade contendo a entidade cujas propriedades pretendemos modificar. Por exemplo, se pretendermos mudar a origem do círculo criado anteriormente para as coordenadas (0, 0, 0) basta-nos avaliar a expressão:

```
_\$ (entmod (list (cons -1 (entlast))
                    (cons 10 (xyz 0 0 0))))
     ((-1 . <Entity name: 7ef91f30>) (10 0 0 0))
```

Quando é bem sucedida, a invocação da função `entmod` devolve a lista de associações usada. Se ocorrer algum erro, a função devolve `nil`.

Como é óbvio no exemplo anterior, o uso da função `entmod` é pouco claro, sendo preferível criarmos um modificador que “esconda” o seu uso.

```
(defun muda-ponto-primario (entidade novo-ponto-primario)
  (entmod (list (cons -1 entidade)
                 (cons 10 novo-ponto-primario))))
```

Naturalmente, cada modificador deve corresponder a um selector. Por exemplo, para aceder e modificar a cor de um objecto, podemos definir as funções:

```
(defun cor (entidade)
  (cdr (assoc 62 (entget entidade)))))

(defun muda-cor (entidade nova-cor)
  (entmod (list (cons -1 entidade)
                 (cons 62 nova-cor))))
```

De igual modo, para aceder e modificar o raio de um círculo, podemos usar:

```
(defun raio (entidade)
  (cdr (assoc 40 (entget entidade))))

(defun muda-raio (entidade novo-raio)
  (entmod (list (cons -1 entidade)
                 (cons 40 novo-raio))))
```

9.4 Criação de Entidades

Há duas maneiras fundamentais de se criarem entidades em Auto Lisp.⁵⁷

⁵⁷Na realidade, há *três* maneiras fundamentais de se criarem objectos em AutoCad mas, por agora, apenas vamos lidar com duas.

- Através da pseudo-função `command`. Esta função recebe como argumentos as expressões necessárias que serão avaliadas apenas se e quando o AutoCad precisar dos valores correspondentes para criar a entidade.

A criação da entidade é ainda afectada por um conjunto de parâmetros globais, como o *layer* em que se está a trabalhar, o UCS actual, o modo OSNAP, a língua, etc. Para se usarem valores diferentes destes parâmetros é necessário fazer as correspondentes alterações previamente à criação da entidade ou é necessário fazer posteriores alterações à entidade.

O uso desta função é ainda dependente das alterações que a Auto-Desk vai fazendo à sintaxe e parâmetros dos comandos em sucessivas versões do AutoCad.

- Através da função `entmake`. Esta função recebe uma lista de associações idêntica à que é devolvida pela função `entget` e usa os valores associados às propriedades correspondentes para criar e inicializar a entidade. As propriedades não obrigatórias ficam com valores por omissão.

A criação da entidade não é afectada por quaisquer outros parâmetros. Em particular, não existe dependência do UCS actual, o que implica que as coordenadas das entidades criadas são interpretadas em termos do WCS.

O uso desta função é ainda dependente das alterações que a Auto-Desk vai fazendo ao formato DXF em sucessivas versões do Auto-Cad.

Há ainda mais uma pequena diferença entre estas duas funções: em geral, usar `entmake` é computacionalmente mais eficiente (por vezes, *muito* mais eficiente) que usar `command`, embora esta última possa ser mais simples para o programador.

Uma vez que já vimos no passado a pseudo-função `command` vamos agora discutir a `entmake`. Para exemplificar, consideremos a criação de um círculo de centro em (4, 4), raio 10 e côr 3 (verde). Como referimos anteriormente, temos de usar como argumento uma lista de associações com os pares propriedade/valor pretendidos:

```
_\$ (entmake (list (cons 0 "CIRCLE")
                     (cons 62 3)
                     (cons 10 (xy 4 4))
                     (cons 40 10)))
((0 . "CIRCLE") (62 . 3) (10 4 4 0) (40 . 10))
```

Há duas importantes considerações a fazer relativamente à criação de entidades:

- O par que contém o tipo da entidade tem de ser o primeiro ou o segundo elemento da lista. Se for o segundo, o primeiro terá de ser o nome da entidade que será simplesmente ignorado.
- Embora não o tenhamos feito no exemplo anterior, a especificação do marcador de subclasse tornou-se obrigatória em todas as versões recentes do AutoCad. O marcador de subclasse corresponde a duas ocorrências do código DXF 100, uma com a *string* AcDbEntity e outra com a classe de entidade (AcDbCircle, AcDbText, AcDbArc, AcDbLine, AcDbPoint, etc). Isso quer dizer que a expressão anterior deveria ter a seguinte forma:

```
(entmake (list (cons 0 "CIRCLE")
               (cons 100 "AcDbEntity")
               (cons 100 "AcDbCircle")
               (cons 62 3)
               (cons 10 (xy 4 4))
               (cons 40 10)))
```

Contudo, para preservar a compatibilidade dos programas anteriores à introdução do marcador de subclasse, existem vários tipos de entidades em relação aos quais a função `entmake` ignora a presença do marcador na lista de associações.⁵⁸ A prática, nestes casos, é nem sequer incluir as associações com esse marcador. Seguindo esta prática, apenas faremos referência a este marcador quando tal for absolutamente necessário.

Se a operação `entmake` for bem sucedida, devolve a lista de associações usada. Se for mal sucedida, devolve `nil`.

Tal como acontecia com a função `entmod`, a utilização “crua” da função `entmake` é complexa, produzindo programas muito pouco legíveis. Particularmente difícil de perceber é o significado dos números (códigos DXF) que representam propriedades. Para resolver este problema, à semelhança do que fizemos com os selectores e modificadores, podemos definir um construtor que aceite todos os parâmetros da criação do círculo, escondendo assim a utilização dos códigos DXF. Infelizmente, no caso do construtor existem demasiados parâmetros e vários deles são opcionais. É lógico que tenhamos de especificar o centro e o raio de um círculo mas a cor, o *layer*, o tipo de linha, etc, só serão necessários em casos raros. Seria um incômodo para o utilizador do construtor ter de passar todos esses parâmetros.

⁵⁸A lista de marcadores ignorados inclui, entre outros, AcDbText, AcDb2dVertex, AcDb3dPolylineVertex, AcDbPolygonMeshVertex, AcDbPolyFaceMeshVertex, AcDbFaceRecord, AcDb2dPolyline, AcDb3dPolyline, AcDbArc, AcDbCircle, AcDbLine, AcDbPoint, AcDbFace, AcDbPolyFaceMesh, AcDbPolygonMesh, AcDbSolid.

9.5 Listas de Propriedades

Para resolver estes problemas vamos seguir uma abordagem diferente. Ao invés de aceitar um número fixo de argumentos, o construtor de círculos aceitará uma lista de argumentos de tamanho variável. A ideia será poder criar um círculo da seguinte forma:⁵⁹

```
(cria-circulo
  (list centro: (xy 10 5)
        raio: 25
        cor: 5))
```

Há dois detalhes importantes a salientar do exemplo anterior. O primeiro detalhe é que, ao invés de usarmos os códigos numéricos DXF, usámos nomes lógicos facilmente comprehensíveis, como `centro:`, `raio:` e `cor:`. O segundo detalhe é que, ao invés de usarmos uma lista de associações em que cada associação é um par propriedade/valor, da forma $((p_0 . v_0) (p_1 . v_1) \dots (p_n . v_n))$, usámos uma lista simples contendo uma sucessão de propriedades e valores, da forma $(p_0 \ v_0 \ p_1 \ v_1 \ \dots \ p_n \ v_n)$. A este tipo de lista dá-se o nome de *lista de propriedades* ou, abreviadamente, *plist*.

Uma lista de propriedades *plist* é muito semelhante a uma lista de associações *alist*. A diferença fundamental está no facto de as associações estarem “espalmadas” na própria lista. Isso faz com que as listas de propriedades sejam mais legíveis do que as listas de associações.

No entanto, é trivial converter listas de propriedades em listas de associações. Mais uma vez, o pensamento recursivo ajuda: se a lista de propriedades é uma lista vazia, então a lista de associações correspondente é também uma lista vazia. Caso contrário, formamos um par com os dois primeiros elementos da lista de propriedades (para formar uma associação) e juntamos esse par ao resultado de converter a lista sem esses dois elementos.

```
(defun alist->-plist (plist)
  (if (null plist)
      ()
      (cons (cons (car plist) (cadr plist))
            (alist->-plist (cddr plist))))
```

Do mesmo modo, é trivial converter listas de associações em listas de propriedades. O raciocínio é idêntico:

⁵⁹Noutros dialectos de Lisp (mas não no Auto Lisp) é possível dispensar a invocação da função `list` pois é possível definir funções com um número variável de argumentos que constroem a lista de argumentos automaticamente.

```
(defun plist<-alist (alist)
  (if (null alist)
      nil
      (cons (caar alist)
            (cons (cdar alist)
                  (plist<-alist (cdr alist)))))))
```

A partir de uma lista de propriedades com os argumentos pretendidos para a criação da figura geométrica temos agora de lhe juntar o tipo pretendido: (0 . "CIRCLE") no caso de círculos, (0 . "LINE") no caso de linhas, etc. Para evitar ter de escrever sempre o mesmo programa, vamos primeiro generalizar um pouco definindo uma função que recebe o tipo de entidade (i.e., a *string* correspondente) e a lista de propriedades e invoca a função `entmake` com a conversão dessa lista de propriedades para o formato de lista de associações requerido:

```
(defun cria-entidade (tipo params)
  (entmake (cons (cons 0 tipo)
                 (alist<-plist params))))
```

A função anterior permite criar entidades arbitrárias. É agora trivial definir a função que cria círculos:

```
(defun cria-circulo (params)
  (cria-entidade "CIRCLE" params))
```

Finalmente, apenas falta definir os nomes lógicos correspondentes às propriedades. Obviamente, o valor associado a cada nome lógico é o código DXF correspondente. Para um círculo, temos:

```
(setq centro: 10
      raio: 40)
```

Outras propriedades são de uso mais geral:

```
(setq tipo-linha: 6
      cor: 62
      layer: 8
      espessura: 39)
```

Para algumas propriedades faz também sentido dar nomes lógicos aos possíveis valores. Por exemplo, no caso da *visibilidade*, cujo código DXF é 60, os possíveis valores são 0 quando é *visível* e 1 quando é *invisível*:

```
(setq visibilidade: 60
      visivel 0
      invisivel 1)
```

Desta forma, podemos criar figuras geométricas de forma muito mais legível. Por exemplo, para criar um círculo invisível basta-nos escrever:

```
(cria-circulo
  (list centro: (xy 1 2)
        raio: 5
        visibilidade: invisivel))
```

Podemos agora definir mais umas figuras geométricas, bem como as propriedades que lhes estão associadas. Por exemplo, para um arco de círculo podemos reutilizar as propriedades do círculo `centro:` e `raio:` e acrescentar mais algumas específicas dos arcos:

```
(defun cria-arco (params)
  (cria-entidade "ARC" params))

(setq arco-angulo-inicial: 50
      arco-angulo-final: 51)
```

Para uma linha temos:

```
(defun cria-linha (params)
  (cria-entidade "LINE" params))

(setq ponto-inicial: 10
      ponto-final: 11)
```

Note-se, no programa anterior, que o nome `ponto-inicial` tem o mesmo valor que o nome `centro`. Obviamente, ambos designam aquilo que anteriormente denominámos de *ponto primário* mas é muito mais claro distinguir os diferentes usos.

Uma outra entidade que pode vir a ser útil é a *ellipse*. Para além da necessidade de definição de constantes para as suas propriedades específicas, a entidade *ellipse* não dispensa a utilização do marcador de classe `AcDbEllipse`, tal como referimos na secção 9.4. Assim, temos:

```
(setq marcador-subclasse: 100
      extremidade-eixo-maior: 11
      eixo-menor/eixo-maior: 40
      ellipse-angulo-inicial: 41
      ellipse-angulo-final: 42)

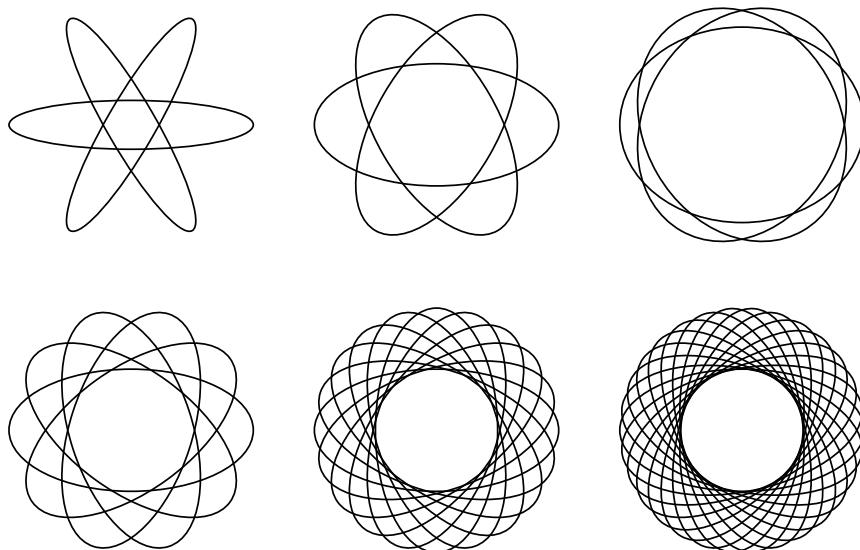
(defun cria-ellipse (params)
  (cria-entidade
    "ELLIPSE"
    (append (list marcador-subclasse: "AcDbEntity"
                  marcador-subclasse: "AcDbEllipse")
            params)))
```

Note-se que acrescentámos (na posição correcta) as duas ocorrências da propriedade 100 relativa ao marcador de subclasse.

Existem dois detalhes importantes relacionados com as propriedades da *ellipse*. O primeiro é que a propriedade `extremidade-eixo-maior:`

pressupõe um ponto relativamente ao referencial do centro da elipse. Isto quer dizer que as coordenadas da extremidade do eixo maior da elipse são dadas como se a origem do sistema de coordenadas estivesse colocada no centro da elipse. O segundo é que a propriedade eixo-menor/eixo-maior representa a razão de proporção entre o comprimento do eixo menor e o do eixo maior. Por exemplo, se o eixo menor tiver 2m de comprimento e o maior tiver 3m, a propriedade eixo-menor/eixo-maior deverá ter associado o valor $2/3 = 0.66666$.

Exercicio 9.1 Defina uma função `flor` capaz de desenhar uma “flor” à custa da sobreposição de elipses rodadas, tal como se apresenta na seguinte imagem:



A função `flor` deverá receber, como parâmetros, as coordenadas do centro da elipse, o comprimento do semi-eixo maior e do semi-eixo menor e o número de pétalas a desenhar de forma a completar uma revolução. Por exemplo, deverá ser possível gerar as imagens anteriores através da avaliação das seguintes expressões:

```
(flor (xy 0.0 0.0) 1 0.5 10)
(flor (xy 2.5 0.0) 1 0.5 20)
(flor (xy 5.0 0.0) 1 0.5 30)
(flor (xy 0.0 2.5) 1 0.2 6)
(flor (xy 2.5 2.5) 1 0.5 6)
(flor (xy 5.0 2.5) 1 0.8 6)
```

Finalmente, para vermos um exemplo de uma entidade bastante parametrizável, consideremos a criação de textos. Uma consulta à documentação DXF do AutoCad revela várias propriedades que podemos definir da seguinte forma:

```

(defun cria-texto (params)
  (cria-entidade "TEXT" params))

  (setq texto: 1
        estilo: 7
        altura: 40
        rotacao: 50
        escala-x: 41
        inclinacao: 51

        geracao: 71
        normal 0
        contrario 2
        invertida 4

        alinhamento-horizontal: 72
        ah-esquerda 0
        ah-centro 1
        ah-direita 2
        ah-alinhado 3
        ah-meio 4
        ah-justificado 5

        alinhamento-vertical: 73
        av-linha-base 0
        av-baixo 1
        av-meio 2
        av-topo 3)

```

Os parâmetros `alinhamento-horizontal` e `alinhamento-vertical` permitem produzir variações na forma como um dado texto se apresenta em relação a dois pontos. A Figura 78 mostra essas possíveis variações.

Para se visualizar a utilização desta última função, a Figura 79 representa a imagem gerada pela seguinte expressão:

```

(foreach fraccao (enumera -0.5 0.5 0.1)
  (cria-texto
    (list ponto-inicial: (xy 0 (* 200 fraccao))
          inclinacao: (* -2 fraccao)
          texto: "Inclinado"
          altura: 10)))

```

Uma outra imagem interessante é a apresentada na Figura 80 que demonstra a capacidade do AutoCad para aplicar rotações ao texto a inserir:

```

(foreach fraccao (enumera 0.0 1.0 0.1)
  (cria-texto
    (list ponto-inicial: (xy 0 0)
          rotacao: (* fraccao 2 pi)
          texto: "Rodado"
          altura: 10)))

```

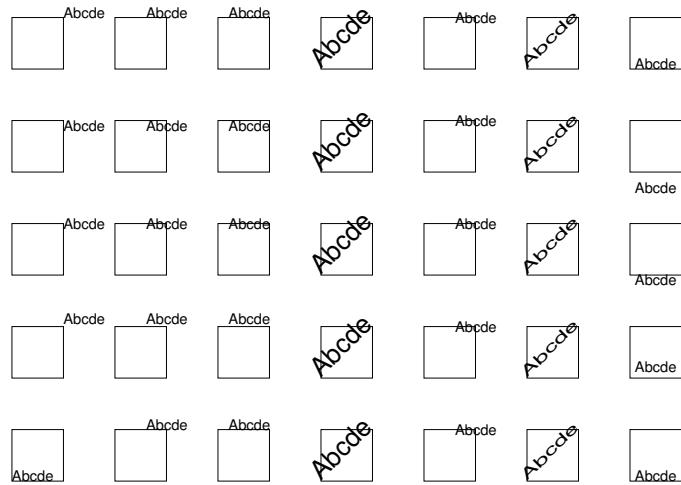


Figura 78: Possibilidades de posicionamento de texto em relação a dois pontos. De baixo para cima o parâmetro alinhamento-vertical toma os valores de 0 a 4. Da esquerda para a direita, o parâmetro alinhamento-horizontal toma os valores de 0 a 6.

Inclinado
Inclinado
Inclinado
Inclinado
Inclinado
Inclinado
Inclinado
Inclinado
Inclinado
Inclinado

Figura 79: Um texto com diferentes inclinações.

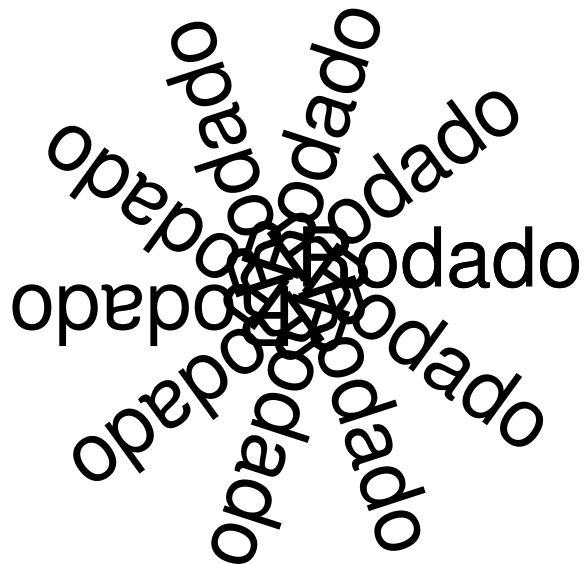


Figura 80: Um texto com diferentes rotações.

Finalmente, na Figura 81 apresentamos uma combinação entre sucessivas rotações do texto e reduções no tamanho da letra. A imagem foi gerada pela expressão:

```
(foreach fraccão (enumera 0.0 1.0 0.1)
  (cria-texto
    (list ponto-inicial: (xy 0 0)
          rotacao: (* fraccão 2 pi)
          texto: "Rodado"
          altura: (* (- 1 fraccão) 10))))
```

Exercicio 9.2 Embora os exemplos anteriores mostrem que a criação de entidades é razoavelmente simples, nem todos os tipos de entidades possuem esta característica. A criação de um texto multilinha (tipo MTEXT) é um exemplo de uma entidade cuja especificação DXF é complexa. Para além de obrigar a lista de associações a incluir a presença do marcador de subclasse para textos multilinha (AcDbMText), existe ainda a necessidade de partir o texto em subtextos com, no máximo 250 caracteres. Isso é claro a partir da especificação desta entidade que, para os códigos DXF 1 e 3, afirma:

- 1 *String* de texto. Se a *string* tiver 250 (ou menos) caracteres, todos aparecem no código
 1. Caso contrário, a *string* é dividida em porções de 250 caracteres que aparecem associadas a um ou mais códigos 3. Se for usado algum código 3, a última porção fica associada ao código 1 e possui menos de 250 caracteres.⁶⁰
- 3 Texto adicional (em porções de 250 caracteres) (opcional)

⁶⁰Na verdade, a documentação DXF do AutoCad apenas refere os casos de *strings* com menos de 250 caracteres e de *strings* com mais de 250 caracteres, deixando na dúvida o que acontece no caso de *strings* com *exatamente* 250 caracteres. Por experimentação verifica-se que este caso também fica associado ao código DXF 1.



Figura 81: Um texto com diferentes rotações e escalas.

Esta documentação mostra que a criação de um texto multilinha implica partitionar o texto em subfragmentos que deverão ser agrupados na lista de associações conjuntamente com os códigos 1 e 3. Tendo este comportamento em conta, defina a função `cria-texto-multilinha` que, a partir de uma *string* e de uma lista de propriedades, partitiona a string e acrescenta as associações necessárias à lista de propriedades para a correcta criação da entidade.

Exercicio 9.3 Consulte a documentação DXF do AutoCad para *splines* e defina uma função `cria-spline` bem como as constantes que considerar relevantes para permitir a fácil criação de curvas *spline*.

9.6 Leitura e Modificação de Entidades

A partir do momento em que associámos nomes às propriedades podemos tirar partido disso para redefinir as operações que obtêm e modificam o valor das propriedades das entidades gráficas:

```
(defun obtem-propriedade (propriedade entidade)
  (cdr (assoc proprietade (entget entidade)))))

(defun muda-propriedade (propriedade novo-valor entidade)
  (entmod (list (cons -1 entidade)
                (cons proprietade novo-valor))))
```

Usando estas operações, podemos também redefinir alguns dos selectores e modificadores que tínhamos introduzido anteriormente. Por exemplo, para a cor, temos:

```
(defun cor (entidade)
  (obtem-propriedade cor: entidade))

(defun muda-cor (entidade nova-cor)
  (muda-propriedade cor: nova-cor entidade))
```

Exercicio 9.4 Defina a função `area-entidade-circulo` que, dada uma entidade do tipo circulo com um raio r , calcula a sua área através da fórmula $A = \pi r^2$.

Exercicio 9.5 Defina a função `area-entidade-elipse` que, dada uma entidade do tipo elipse, calcula a sua área. Dada uma elipse de semi-eixos a e b , a sua área A é dada por $A = \pi ab$.

9.7 Eliminação de Entidades

Para se eliminar uma entidade, o AutoCad disponibiliza a função `entdel` que recebe, como argumento, a entidade a eliminar. Se a entidade já tiver sido eliminada e o desenho a que ela pertencia ainda não tiver sido encerrado, a invocação desta função repõe a entidade no desenho.

Para se determinar se uma entidade está eliminada, podemos usar a função `entget` com a entidade em questão como argumento. Se essa entidade estiver eliminada, a função `entget` devolverá `nil`. Este comportamento é visível na seguinte interacção com o Auto Lisp:

```
_\$ (cria-circulo          ;; criamos um circulo
     (list centro: (xy 1 2) raio: 10))
((0 . "CIRCLE") (10 1 2) (40 . 10))
_\$ (setq circulo (entlast)) ;; guardamo-lo numa variavel
<Entity name: 7efa26d8>
_\$ (entget circulo)        ;; obtemos as suas propriedades
((-1 . <Entity name: 7efa26d8>) (0 . "CIRCLE") ...)
_\$ (entdel circulo)       ;; eliminamo-lo
<Entity name: 7efa26d8>
_\$ (entget circulo)        ;; obtemos as suas propriedades de novo
nil                         ;; e nada recebemos
_\$ (entdel circulo)       ;; "deseliminamo-lo"
<Entity name: 7efa26d8>
_\$ (entget circulo)        ;; obtemos as suas propriedade de novo
((-1 . <Entity name: 7efa26d8>) (0 . "CIRCLE") ...)
```

9.8 Última Entidade

No passado dissemos que a função `entlast` devolve a última entidade criada. Podemos agora, mais correctamente, dizer que esta função devolve a última entidade não eliminada, tal como se pode ver no seguinte exemplo:

```

_§ (cria-circulo
    (list centro: (xy 1 2) raio: 10))
((0 . "CIRCLE") (10 1 2) (40 . 10))
_§ (tipo-entidade (entlast))
"CIRCLE"
_§ (cria-linha
    (list ponto-inicial: (xy 3 4) ponto-final: (xy 5 6)))
((0 . "LINE") (10 3 4) (11 5 6))
_§ (tipo-entidade (entlast))
"LINE"
_§ (entdel (entlast))
<Entity name: 7efcaab8>
_§ (tipo-entidade (entlast))
"CIRCLE"

```

9.9 Iteração de Entidades

O AutoCad disponibiliza ainda uma operação para *iterar* as entidades de um desenho: `entnext`.

Quando invocada sem argumentos, a função devolve a *primeira* entidade não eliminada (ou `nil`, se não existir nenhuma). Quando invocada com uma entidade como argumento, a função devolve a entidade (não eliminada) *seguinte* (ou `nil` se o argumento for a última entidade não eliminada). Tal como se vê no seguinte exemplo, este comportamento permite iterar ao longo de todas as entidades do desenho:

```

_§ (command "_.erase" "_all" "")
nil
_§ (entnext)
nil
_§ (cria-circulo
    (list centro: (xy 1 2)
          raio: 10))
((0 . "CIRCLE") (10 1 2) (40 . 10))
_§ (cria-linha
    (list ponto-inicial: (xy 3 4)
          ponto-final: (xy 5 6)))
((0 . "LINE") (10 3 4) (11 5 6))
_§ (entnext)
<Entity name: 7efcaac0>
_§ (entnext (entnext))
<Entity name: 7efcaac8>
_§ (entnext (entnext (entnext)))
nil

```

Como base nesta função podemos construir outras bastante mais úteis. Por exemplo, dada uma entidade podemos coleccionar numa lista todas as entidades que foram criadas depois dela:

```
(defun entidades-desde (actual)
  (if (null actual)
      ()
      (cons actual
            (entidades-desde (entnext actual)))))
```

Tendo em conta que a invocação (`entnext`) (i.e., sem argumentos) devolve a primeira entidade de todas, podemos agora trivialmente colecionar todas as entidades presentes no desenho actual:

```
(defun todas-entidades ()
  (entidades-desde (entnext)))
```

Continuando o exemplo anterior, temos:

```
_\$ (todas-entidades)
(<Entity name: 7efcaac0> <Entity name: 7efcaac8>)
```

9.10 Entidades Compostas

Até agora vimos que cada figura geométrica que criamos no AutoCad possui uma entidade associada. No entanto, algumas das figuras geométricas disponibilizadas pelo AutoCad são demasiadamente complexas para poderem ser representadas por uma única entidade. Por exemplo, uma linha poligonal a três dimensões⁶¹ necessita não só de uma entidade para representar as características gerais da linha como também necessita de uma sucessão de entidades para representar a sucessão dos vértices da linha. Por ser composta por várias subentidades, uma linha poligonal diz-se uma *entidade composta*.

Após a criação de uma linha poligonal, embora a entidade que a representa seja aceitável usando a função `entlast`, para se aceder às entidades que representam os vértices é necessário iterar a partir da primeira, usando a função `entnext`, tal como é perceptível na seguinte interacção:⁶²

⁶¹Versões recentes do AutoCad permitem um outro tipo de linha poligonal denominada linha poligonal *leve* que é representada por uma única entidade. Esta linha, contudo, é estritamente bidimensional e não permite o mesmo grau de sofisticação das linhas poligonais 3D.

⁶²Para facilitar a leitura, omitimos alguma da informação produzida.

```

_> $ (command "_.3DPOLY"
              (xy -1 -1 0) (xy 1 0 1) (xy 0 1 -1)
              "")
nil
_> $ (entget (entlast))
((-1 . <Entity name: 7efcb1f0>)
 (0 . "POLYLINE") ...
 (10 0.0 0.0 0.0) ...)
_> $ (entget (entnext (entlast)))
((-1 . <Entity name: 7efcb1f8>)
 (0 . "VERTEX") ...
 (10 -1.0 -1.0 0.0) ...)
_> $ (entget (entnext (entnext (entlast))))
((-1 . <Entity name: 7efcb200>)
 (0 . "VERTEX") ...
 (10 1.0 0.0 1.0) ...)
_> $ (entget (entnext (entnext (entnext (entlast)))))
((-1 . <Entity name: 7efcb208>)
 (0 . "VERTEX") ...
 (10 0.0 1.0 -1.0) ...)
_> $ (entget (entnext (entnext (entnext (entnext (entlast))))))
((-1 . <Entity name: 7efcb210>)
 (0 . "SEQEND") ...)
_> $ (entnext (entnext (entnext (entnext (entlast))))))
nil

```

Repare-se nos tipos das entidades que são acedidas: embora a primeira seja do tipo POLYLINE, as seguintes são do tipo VERTEX e, finalmente, a última é do tipo SEQEND. Esta última entidade é usada para “marcar” o fim da sequência de entidades que constituem a entidade composta.

Com base nesta informação estamos em condições de definir funções que processem as linhas poligonais. Por exemplo, para sabermos quais são as coordenadas dos vértices de uma linha poligonal, podemos simplesmente começar com a entidade que representa a linha e ir recursivamente recolhendo as coordenadas das entidades seguintes até atingir a entidade cujo tipo é SEQEND:

```

(defun junta-vertices (entidade)
  (if (= (tipo-entidade entidade) "SEQEND")
      (list)
      (cons (ponto-primario entidade)
            (junta-vertices (entnext entidade)))))

(defun vertices-linha-3d (linha)
  (junta-vertices (entnext linha)))

```

É agora bastante mais simples obter as coordenadas que definem a linha poligonal:

```

_> $ (vertices-linha-3d (entlast))
((-1.0 -1.0 0.0) (1.0 0.0 1.0) (0.0 1.0 -1.0))

```

Para se criar uma linha poligonal, é necessário usar o processo inverso.

Dada a lista de coordenadas dos vértices da linha a criar, criamos a entidade que representa a linha poligonal e iteramos ao longo da lista, criando um vértice para cada elemento. Finalmente, terminamos com a entidade que marca o fim da sequência. Para tornar a especificação das propriedades mais clara, é conveniente definirmos previamente algumas constantes:

```
(setq tipo-linha: 70
      linha-aberta 0
      linha-fechada 1
      linha-3d 8
      malha-poligonal-3d 16

      tipo-vertice: 70
      vertice-extra 1
      vertice-tangente 2
      vertice-spline 8
      vertice-controlo 16
      vertice-3d 32
      vertice-malha 64)
```

A linha poligonal pode agora ser criada usando a seguinte função:

```
(defun cria-linha-3d (vertices)
  (cria-entidade "POLYLINE"
    (list tipo-linha: polilinha-3d))
  (foreach vertice vertices
    (cria-entidade "VERTEX"
      (list centro: vertice
        tipo-vertice: vertice-3d)))
  (cria-entidade "SEQEND" (list)))
```

Note-se que a função anterior apenas cria um tipo específico de linha poligonal.

9.11 Redesenho de Entidades

Há uma função relacionada com entidades que embora não seja geralmente necessária, pode vir a ser útil: `entupd`. Esta função recebe uma entidade como argumento e redesenha-a no *ecran*.

Em geral, nunca é preciso usar esta função pois o AutoCad encarrega-se de redesenhar automaticamente as entidades que mudam. No entanto, no caso de entidades cujo processo de modificação é moroso (como os vértices de uma linha poligonal a três dimensões), o AutoCad não faz o redesenho automaticamente. Nestes casos, fica à responsabilidade do programador invocar a função `entupd` sobre a entidade modificada.

9.12 Criação de Linhas Poligonais “Leves”

As linhas poligonais descritas na secção 9.10 são entidades complexas compostas por várias subentidades. Nesta secção vamos ver uma variante mais

“ligeira” dessas linhas poligonais que é usada pelo AutoCad sempre que lida com linhas poligonais bidimensionais.

Contrariamente a uma linha poligonal genérica, uma linha poligonal leve é uma única entidade. Os vértices da linha poligonal são representados como múltiplas ocorrências de uma mesma propriedade cujo código DXF é o número 10. Estes vértices são relativos a um sistema de coordenadas próprio de cada entidade, denominado *sistema de coordenadas do objecto* (do Inglês *Object Coordinate System* ou, abreviadamente, *OCS*). Sempre que se pretender conhecer as coordenadas dos vértices da linha poligonal relativamente ao *sistema de coordenadas do mundo* (do Inglês *World Coordinate System* ou, abreviadamente, *WCS*) é necessário fazer uma transformação de coordenadas, implementada pela função `trans`.

Para construirmos uma destas linhas vamos definir a seguinte função:

```
(defun cria-polilinha-leve (params)
  (cria-entidade "LWPOLYLINE" params))
```

Infelizmente, no caso de polilinhas leves, a construção é mais complexa do que anteriormente pois, por um lado, não é possível omitir o marcador de subclasse (ver secção 9.4) e, por outro, é necessário associar um código DXF a cada vértice. Por exemplo, para criarmos um rectângulo com os vértices nas coordenadas (10,20), (20,20), (20,40) e (10,40), teremos de avaliar a seguinte expressão:

```
(cria-polilinha-leve
  (list 100 "AcDbEntity"      ;;marcador de subclasse
        100 "AcDbPolyline"   ;;idem
        70 1                  ;;tipo de linha fechada
        90 4                  ;;numero de vertices
        10 (xy 10 20)         ;;vertice
        10 (xy 20 20)         ;;idem
        10 (xy 20 40)         ;;idem
        10 (xy 10 40)))      ;;idem
```

Uma vez que é incômodo termos de especificar sistematicamente os marcadores de subclasse e o número de vértices, vamos redefinir a função anterior de modo a que ela aceite também a seguinte sintaxe para criar o mesmo quadrado:

```
(cria-polilinha-leve
  (list vertices: (list (xy 10 20)
                        (xy 20 20)
                        (xy 20 40)
                        (xy 10 40))
    tipo-linha: linha-fechada))
```

A ideia consiste em inventar uma nova opção `vertices:` (com um código que não interfira com os códigos DXF pré-definidos) e, sempre que

ele aparece, processamos a lista que lhe está associada de modo a gerar o código DXF correcto para cada vértice. Note-se que os restantes parâmetros não devem ser afectados, permitindo misturar as várias formas de especificar os vértices. Por exemplo, se tivermos os vértices p_0, \dots, p_4 e a lista de parâmetros for $(10\ p_0\ 70\ 1\ -1000\ (p_1\ p_2\ p_3)\ 10\ p_4)$, a sua conversão irá produzir a lista $(10\ p_0\ 70\ 1\ 10\ p_1\ 10\ p_2\ 10\ p_3\ 10\ p_4)$. Este comportamento está incorporado no seguinte programa onde começamos por definir as constantes relevantes:

```
(setq vertices: -1000
      numero-vertices: 90
      vertice: 10)

(defun converte-parametros (params)
  (cond ((null params)
         (list))
        ((= (car params) vertices:)
         (append (converte-vertices (cadr params))
                 (converte-parametros (cddr params))))
        (t
         (cons (car params)
               (cons (cadr params)
                     (converte-parametros (cddr params)))))))

(defun converte-vertices (vertices)
  (if (null vertices)
      (list)
      (cons vertice:
            (cons (car vertices)
                  (converte-vertices (cdr vertices))))))
```

Para sabermos o número de vértices precisamos agora de contabilizar as ocorrências do código associado a `vertice`:

```
(defun conta-parametros-vertice (params)
  (cond ((null params)
         0)
        ((= (car params) vertice:)
         (1+ (conta-parametros-vertice (cddr params))))
        (t
         (conta-parametros-vertice (cddr params)))))
```

Finalmente, podemos combinar tudo numa só função:

```
(defun cria-polilinha-leve (params)
  (setq params (converte-parametros params))
  (cria-entidade
   "LWPOLYLINE"
   (append (list marcador-subclasse: "AcDbEntity"
                 marcador-subclasse: "AcDbPolyline"
                 numero-vertices:
                 (conta-parametros-vertice params))
          params)))
```

Para vermos um exemplo da utilização desta função, consideremos o

desenho de polígonos que foi descrito na secção 8. Na altura, definimos uma função que, a partir das coordenadas dos vértices do polígono, desenhava-o como uma sucessão de linhas:

```
(defun poligono (vertices)
  (command "_.line")
  (foreach vertice vertices
    (command vertice))
  (command "_close"))
```

Admitindo que todas as coordenadas são bi-dimensionais, podemos re-definir a função de modo a usar a criação de polilinhas leves:

```
(defun poligono (vertices)
  (cria-polilinha-leve
    (list tipo-linha: linha-fechada
      vertices: vertices)))
```

Obviamente, com esta nova definição todos os exemplos que foram apresentados na secção 8 continuam a produzir os mesmos resultados. A diferença estará apenas na *performance* e no facto de a utilização de entidades não obedecer aos parâmetros que afectam o `command`, nomeadamente, o UCS, o SNAP, etc.

9.13 Manipulação de Linhas Poligonais “Leves”

Para além de podermos construir linhas poligonais leves, podemos ter a necessidade de as manipular.

Para a maioria das propriedades como, por exemplo, o `tipo-linha:`, podemos continuar a usar o selector `obtem-propriedade`. No entanto, para obtermos os vértices da linha poligonal leve (no referencial OCS) temos de empregar uma função diferente da `obtem-propriedade` pois esta apenas devolve o valor associado à *primeira* ocorrência da propriedade na lista de associações enquanto que agora precisamos de construir uma lista com *todas* as ocorrências da propriedade `vertice:`. Assim, temos:

```
(defun valores-propriedade (propriedade lista-assoc)
  (cond ((null lista-assoc)
         (list))
        ((= propriedade (caar lista-assoc))
         (cons (cdr lista-assoc)
               (valores-propriedade propriedade
                                     (cdr lista-assoc)))))
        (t
         (valores-propriedade propriedade
                               (cdr lista-assoc))))))
```

Usando esta função é agora trivial definir:

```
(defun vertices-polilinha-leve (entidade)
  (valores-propriedade vertice: (entget entidade)))
```

É claro que se quisermos ter uma só função que produz a lista de vértices de uma linha poligonal qualquer, seja ela uma POLYLINE ou uma LWPOLYLINE, basta-nos testar o tipo de entidade e usar a função apropriada:

```
(defun vertices-polilinha (entidade / tipo)
  (setq tipo (tipo-entidade entidade))
  (cond ((= tipo "POLYLINE")
         (vertices-polilinha-3d entidade))
        ((= tipo "LWPOLYLINE")
         (vertices-polilinha-leve entidade))))
```

9.14 Selecção Interactiva de Entidades

Vimos que a função `entlast` permite aceder à última entidade criada e que a função `entnext` permite iterar todas as entidades. Infelizmente, estas duas funções não são suficientes para permitir identificar uma entidade escolhida pelo utilizador.

Quando é necessário pedir ao utilizador que seleccione uma entidade presente num desenho pode-se empregar a função `entsel` que recebe, opcionalmente, um texto a apresentar ao utilizador. Quando invocada, a função começa por apresentar o texto e, de seguida, aguarda que o utilizador use o rato. Quando o utilizador clica num objecto gráfico qualquer, a função devolve uma lista contendo, como primeiro elemento, a entidade seleccionada e, como segundo elemento, as coordenadas do ponto onde o utilizador clicou.

```
_ $ (entsel "Escolha uma entidade")
(<Entity name: 7efcaac0> (7.04109 -5.97352 0.0))
```

A razão porque é devolvido também o ponto clicado prende-se com o facto de podermos usar essa informação para, de alguma forma, modificarmos a entidade. Por exemplo, se pretendermos partir uma linha em duas num dado ponto, podemos definir a seguinte função:

```
(defun parte-linha (linha ponto)
  (cria-linha
   (list
    ponto-inicial: (obtem-propriedade ponto-inicial: linha)
    ponto-final: ponto)))
  (cria-linha
   (list
    ponto-inicial: ponto
    ponto-final: (obtem-propriedade ponto-final: linha)))
  (entdel linha))
```

Podemos agora usar a função `entsel` para obter a linha que pretendemos partir em duas e o ponto nessa linha onde a queremos partir, passando

esses valores separadamente para a função parte-linha. É isso que a função seguinte faz:

```
(defun comando-parte-linha (/ seleccao)
  (setq seleccao (entsel))
  (parte-linha (car seleccao) (cadr seleccao)))
```

9.15 Definição de Comandos AutoCad

Até agora, todas as operações que definimos apenas podiam ser invocadas a partir do Auto Lisp. Para quem gosta de alisp, isso é suficiente, mas para o utilizador menos experiente de AutoCad isso pode rapidamente tornar-se inconveniente: na maioria dos casos será preferível poder invocar funções sem abandonar o ambiente do AutoCad e sem sequer ter se saber a sintaxe do Lisp.

Para isso, o Auto Lisp permite a definição de comandos AutoCad. Essa definição é feita de forma idêntica à definição de funções, sendo a distinção feita apenas por uma convenção de nomes: para se definir o comando *nome*, basta definir uma função cujo nome será *c:nome* e cujo corpo irá conter as acções que se pretende que o comando realize. Uma vez que a função se destina a ser invocada como se de um comando AutoCad se tratasse, ela não pode receber argumentos e, consequentemente, não pode ter parâmetros. No entanto, embora se destine fundamentalmente a ser usada no AutoCad, o comando definido não deixa de estar implementado por uma função Lisp e, por isso, pode também ser invocado a partir do Lisp. À semelhança de qualquer outra invocação de função, o comando *nome* assim definido pode ser invocado como (*c:nome*). Note-se que o nome da função, em Auto Lisp, é aquele que foi empregue na definição. É apenas para o AutoCad que o nome perde o prefixo *c*:

Na definição de comandos, é importante termos em conta que se o comando a definir tiver o mesmo nome que um comando pré-definido, ele só ficará activo quando indefinirmos o comando pré-definido.

Usando a definição de comandos AutoCad podemos tornar mais prática a operação de partir uma linha:

```
(defun c:parte-linha (/ seleccao)
  (setq seleccao (entsel))
  (parte-linha (car seleccao) (cadr seleccao)))
```

Note-se que a função anterior não é recursiva: a função *c:parte-linha* invoca a função *parte-linha* mas trata-se de duas funções diferentes. No entanto, com a definição anterior fica disponível no AutoCad um novo comando denominado *parte-linha* que, quando invocado, provoca a invocação da função *c:parte-linha* que, por sua vez, invoca a função *parte-linha*.

Função	Tipo de Resultado
getint	Um inteiro.
getreal	Um real.
getstring	Uma <i>string</i> .
getpoint	Um ponto obtido na área de desenho ou na linha de comandos.
getcorner	Um ponto relativo a um outro ponto dado, obtido na área de desenho ou na linha de comandos.
getdist	Uma distância obtida na área de desenho ou na linha de comandos.
getangle	Um ângulo (em radianos) obtido na área de desenho ou na linha de comandos. O ângulo zero é dado pela variável ANGBASE e cresce segundo ANGDIR.
getorient	Um ângulo (em radianos) obtido na área de desenho ou na linha de comandos. O ângulo zero é sempre para a direita e cresce segundo ANGDIR.

Tabela 6: Funções de obtenção de informação pré-definidas do Auto Lisp.

9.16 Obtenção de Informação

Vimos que a função `entsel` permite pedir ao utilizador para seleccionar uma entidade, entidade essa que será devolvida juntamente com as coordenadas do ponto clicado. Em muitos casos, contudo, podemos necessitar também de outras informações. Por exemplo, podemos pedir ao utilizador que indique pontos na sua área de desenho. Outra possibilidade será pedirmos números ou textos ou nomes de ficheiros.

Para cada um destes fins, o Auto Lisp disponibiliza uma função pré-definida: `getpoint`, `getcorner`, `getint`, `getreal`, `getstring`, `getfiled`, etc. Este conjunto de funções está descrito na Tabela 6. Qualquer delas recebe um argumento opcional que é uma *string* a apresentar ao utilizador para o informar do que se pretende.

É ainda possível restringir os valores que se podem obter a partir das funções anteriores através da utilização da função `initget`, por exemplo, proibindo os valores negativos ou limitando os pontos obtidos a terem uma coordenada *z* nula.

9.17 Áreas de Polígonos

Vimos que a definição de funções Auto Lisp pode ser usada também para definir comandos AutoCad. Nesta secção vamos combinar a definição de comandos com a utilização de funções que pedem informações ao utilizador.

A título de exemplo, consideremos um comando para criar uma divisão

rectangular em plantas de habitações. O comando irá pedir a posição dos cantos da divisão e, com esses valores, irá construir o rectângulo correspondente e colocará no interior desse rectângulo uma linha de texto com a área da divisão.

Para implementarmos este comando vamos começar por definir a função `divisao-rectangular` que, a partir do canto inferior esquerdo e do canto superior direito, constrói o rectângulo em questão e insere o texto pretendido:

```
(defun divisao-rectangular (p0 p1 / meia-altura)
  (cria-polilinha-leve
    (list tipo-linha: linha-fechada
          vertices: (list p0
                           (xy (cx p1) (cy p0))
                           p1
                           (xy (cx p0) (cy p1))))
    (setq meia-altura (/ (+ (cy p0) (cy p1)) 2.0))
    (cria-texto
      (list ponto-inicial: (xy (cx p0) meia-altura)
            ponto-final: (xy (cx p1) meia-altura)
            altura: 1
            texto: (strcat " Area:"
                           (rtos (* (- (cx p1) (cx p0))
                                     (- (cy p1) (cy p0))))
                           " "))
            alinhamento-horizontal: ah-alinhado
            alinhamento-vertical: av-meio))))
```

Uma vez que tem parâmetros, esta função apenas pode ser invocada a partir do Auto Lisp. Isso é suficiente para testarmos o seu funcionamento mas não é suficiente para o seu uso a partir do AutoCad. O objectivo, agora, é criar um comando que recolhe do utilizador os dados suficientes para conseguir invocar a função anterior. Para evitar estarmos sempre a definir novos nomes, vamos reutilizar o nome da função `divisao-rectangular` para baptizar o comando:

```
(defun c:divisao-rectangular (/ p0 p1)
  (setq p0 (getpoint "\nEspecifique o primeiro canto")
        p1 (getcorner "\nEspecifique o segundo canto" p0))
  (divisao-rectangular p0 p1))
```

Infelizmente, a definição anterior tem um problema: apenas funciona correctamente se o segundo canto ficar acima e à direita do primeiro canto. Esse problema é visível na Figura 82 onde acrescentámos também uma seta desde o primeiro ponto fornecido até o segundo ponto fornecido.

Para corrigirmos o problema, necessitamos de processar os pontos recolhidos de forma a invocarmos a função `divisao-rectangular` com o canto inferior esquerdo e com o canto superior direito do rectângulo “virtual” recolhido. É precisamente isso que faz a seguinte redefinição:

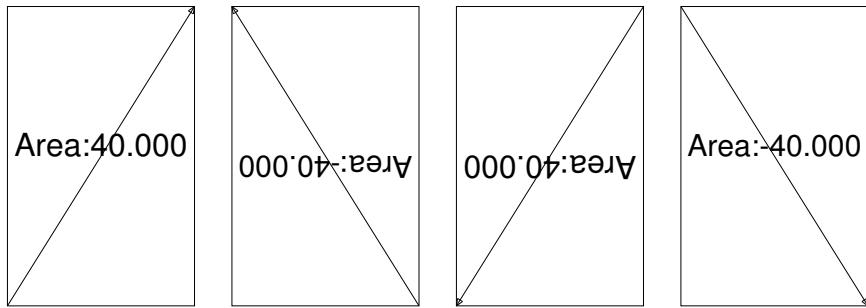


Figura 82: Divisões rectangulares com a área ou a orientação do texto incorrectamente calculadas.



Figura 83: Divisões rectangulares criadas com o comando AutoCad divisao-rectangular.

```
(defun c:divisao-rectangular (/ p0 p1)
  (setq p0 (getpoint "\nEspecifique o primeiro canto")
        p1 (getcorner "\nEspecifique o segundo canto" p0))
  (divisao-rectangular (xy (min (cx p0) (cx p1))
                           (min (cy p0) (cy p1)))
                       (xy (max (cx p0) (cx p1))
                           (max (cy p0) (cy p1)))))
```

Com esta correção é agora possível criar, interactivamente, a planta de uma habitação e anotar cada divisão criada com a informação da sua área, tal como ilustramos na Figura 83.

Embora o comando `divisao-rectangular` permita criar divisões onde é automaticamente calculada e escrita a sua área, existem duas importantes limitações:

- Apenas permite criar áreas rectangulares alinhadas com os eixos x e y .

- Não permite inserir a área em figuras geométricas já existentes num desenho.

Estas limitações sugerem que repensem o comando. A primeira limitação resolve-se de forma trivial: não vale a pena ser o comando a criar o rectângulo pois o AutoCad já possui uma operação para isso, denominada `rectangle`. Na realidade, o AutoCad já disponibiliza muitas operações capazes de criar as mais variadas figuras geométricas, pelo que é inútil estarmos a tentar replicar essa funcionalidade. A segunda limitação é um pouco mais difícil de resolver pois o cálculo da área e do ponto de inserção dessa área depende da figura geométrica em questão, o que nos vai obrigar a considerar várias formas diferentes de o fazermos.⁶³

Comecemos por analisar o cálculo da área de uma entidade geométrica. Se a entidade for um círculo de raio r , a área é πr^2 mas se for uma elipse de semi-eixos a e b , a área é πab . Isto permite-nos definir as funções:

```
(defun area-entidade-circulo (entidade)
  (* pi (quadrado (obtem-propriedade raio: entidade)))))

(defun area-entidade-elipse (entidade / eixo-maior)
  (setq eixo-maior
        (pol-ro (obtem-propriedade extremidade-eixo-maior: entidade)))
  (* pi
     eixo-maior
     (obtem-propriedade eixo-menor/eixo-maior: entidade)
     eixo-maior))
```

Para podermos tratar de forma genérica qualquer das duas entidades anteriores, podemos definir uma função que distingue os dois casos:

```
(defun area-entidade (entidade / tipo)
  (setq tipo (tipo-entidade entidade))
  (cond ((= tipo "CIRCLE")
         (area-entidade-circulo entidade))
        ((= tipo "ELLIPSE")
         (area-entidade-elipse entidade))))
```

Quanto ao ponto de inserção, vamos considerar que o texto irá ficar centrado na entidade, pelo que podemos definir uma operação genérica que distingue os vários tipos de entidade:⁶⁴

⁶³Versões recentes do AutoCad são capazes de determinar a área e o centro de um vasto conjunto de entidades. No entanto, para efeitos pedagógicos, vamos admitir que essa funcionalidade não existe.

⁶⁴No caso dos círculos e elipses, seria possível combinar as duas situações numa só pois a propriedade para aceder ao centro da entidade é a mesma. No entanto, para tornar o programa mais claro, vamos manter a separação.

```
(defun centro-entidade (entidade / tipo)
  (setq tipo (tipo-entidade entidade))
  (cond ((= tipo "CIRCLE")
         (obtem-propriedade centro: entidade))
        ((= tipo "ELLIPSE")
         (obtem-propriedade centro: entidade))))
```

A partir do momento em que temos funções genéricas que nos dão a área e o centro de uma entidade, podemos definir a função (igualmente genérica) que, a partir de uma entidade, insere no seu centro um texto a descrever a sua área:

```
(defun insere-area-entidade (entidade / centro)
  (setq centro (centro-entidade entidade))
  (cria-texto
    (list ponto-inicial: centro
          ponto-final: centro
          altura: 1.5
          texto: (strcat "Area:"
                         (rtos (area-entidade entidade)))
          alinhamento-horizontal: ah-centro
          alinhamento-vertical: av-meio)))
```

Por fim, podemos definir um comando no AutoCad que, após pedir ao utilizador que seleccione uma entidade, invoca a função anterior usando a entidade seleccionada como argumento:

```
(defun c:insere-area ()
  (insere-area-entidade
    (car (entsel "Seleccione a entidade pretendida"))))
```

Como é óbvio, o comando anterior apenas pode ser aplicado a círculos e elipses, o que limita excessivamente a sua utilidade. Vamos, portanto, modificá-lo para lidar com outras entidades, por exemplo, rectângulos, pentâgonos, etc.

Uma característica interessante do AutoCad que nos simplifica muito o problema é o facto de a maioria das figuras geométricas poligonais estarem implementadas usando o mesmo tipo de linhas poligonais que discutimos na secção 9.12. De facto, o AutoCad não possui um tipo de entidade específico para rectângulos ou pentâgonos, optando por implementá-los como sequências de linhas poligonais leves, tal como é verificável nas seguintes interacções:

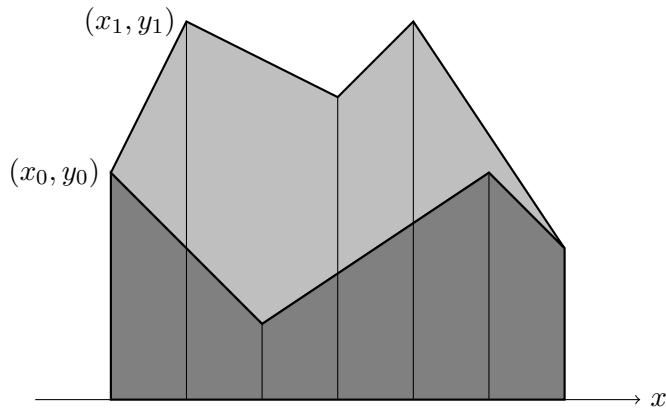


Figura 84: Decomposição de um polígono arbitrário em trapézios rectangulares.

```

_ $ (command "_rectangle" (xy 1 2) (xy 3 4))
nil
_ $ (tipo-entidade (entlast))
"LWPOLYLINE"
_ $ (command "_polygon" 5 (xy 1 2) "_inscribed" 3)
nil
_ $ (tipo-entidade (entlast))
"LWPOLYLINE"

```

Assim, podemos preocupar-nos apenas em lidar com este tipo de entidades e, automaticamente, ficaremos habilitados a lidar com uma enorme variedade de figuras geométricas. Temos, portanto, de encontrar uma forma de calcular a área e o centro de um qualquer polígono delimitado por linhas poligonais leves.

Para calcular a área de um polígono podemos empregar uma abordagem simples que foi inventada independentemente pelos matemáticos Meister e Gauss em finais do século dezoito. A ideia consiste em decompor o polígono num conjunto de trapézios rectangulares, tal como se apresenta na Figura 84. Se for $[(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_{n-1}, y_{n-1}), (x_n, y_n)]$ a sequência de coordenadas dos vértices do polígono, o primeiro trapézio terá os vértices em $(x_0, 0)$, (x_0, y_0) , (x_1, y_1) e $(x_1, 0)$, o segundo em $(x_1, 0)$, (x_1, y_1) , (x_2, y_2) e $(x_2, 0)$, o penúltimo em $(x_{n-1}, 0)$, (x_{n-1}, y_0) , (x_n, y_n) e $(x_n, 0)$ e o último trapézio terá os vértices em $(x_n, 0)$, (x_n, y_n) , (x_0, y_0) e $(x_0, 0)$.

A área de cada um destes trapézios pode ser calculada pela fórmula

$$A = \frac{1}{2}(x_{i+1} - x_i)(y_{i+1} + y_i)$$

sendo que a diferença $x_{i+1} - x_i$ será positiva ou negativa consoante x_i é menor ou maior que x_{i+1} , implicando que a área do trapézio respectivo será

também positiva ou negativa. Ora sendo o polígono fechado, se percorrermos os vértices por ordem, por exemplo, segundo os ponteiros do relógio, necessariamente que alguns dos vértices terão coordenadas x_i sucessivamente crescente enquanto outros a terão sucessivamente decrescente. Não é difícil vermos que ao somarmos os trapézios com área positiva aos trapézios com área negativa, iremos obter a área total do polígono. Essa área poderá ser positiva ou negativa consoante a ordem escolhida para percorrer os vértices do polígono mas, em qualquer caso, será a área total do polígono.

Traduzindo este raciocínio para Lisp, podemos definir as seguintes funções que, a partir de uma lista com as coordenadas dos vértices do polígono, calculam a sua área:

```
(defun area-vertices (vertices)
  (if (null (cdr vertices))
      0
      (+ (area-trapezio (car vertices) (cadr vertices))
          (area-vertices (cdr vertices)))))

(defun area-trapezio (p0 p1)
  (* 0.5
     (- (cx p1) (cx p0))
     (+ (cy p0) (cy p1))))
```

Exercício 9.6 Embora correcto, o algoritmo anterior realiza trabalho desnecessário. Isso torna-se óbvio quando “desenvolvemos” o cálculo para um polígono qualquer. Por exemplo, num polígono de três vértices, temos:

$$A = \frac{1}{2}(x_1 - x_0)(y_1 + y_0) + \frac{1}{2}(x_2 - x_1)(y_2 + y_1) + \frac{1}{2}(x_0 - x_2)(y_0 + y_2)$$

ou seja

$$\begin{aligned} A &= \frac{1}{2}(x_1y_1 + x_1y_0 - x_0y_1 - x_0y_0) + \\ &\quad \frac{1}{2}(x_2y_2 + x_2y_1 - x_1y_2 - x_1y_1) + \\ &\quad \frac{1}{2}(x_0y_0 + x_0y_2 - x_2y_0 - x_2y_2) \end{aligned}$$

É evidente, na fórmula anterior, que há cálculos que são feitos repetidamente mas com sinais opostos. Eliminando os termos que se anulam, obtemos:

$$\begin{aligned} A &= \frac{1}{2}(x_1y_0 - x_0y_1) + \\ &\quad \frac{1}{2}(x_2y_1 - x_1y_2) + \\ &\quad \frac{1}{2}(x_0y_2 - x_2y_0) \end{aligned}$$

Não é difícil generalizar a fórmula anterior para uma sequência arbitrária de vértices, permitindo-nos obter uma simplificação do processo de cálculo:

$$A = \frac{1}{2} \sum_{i=0}^{n-1} x_{i+1}y_i - x_iy_{i+1}$$

Redefina a função `area-vertices` de forma a tirar partido desta simplificação.

Há um detalhe do algoritmo anterior que ainda não está tratado: o último trapézio tem vértices em $(x_n, 0)$, (x_n, y_n) , (x_0, y_0) e $(x_0, 0)$, mas a lista de vértices termina em (x_n, y_n) . Uma forma trivial de implementarmos este detalhe consiste em acrescentarmos o vértice (x_0, y_0) ao fim da lista de vértices. É precisamente isso que a seguinte função faz onde aproveitamos também para obter o valor absoluto para termos sempre uma área positiva:

```
(defun area-poligono (vertices)
  (abs (area-vertices (append vertices (list (car vertices))))))
```

Finalmente, podemos aplicar este cálculo a qualquer polígono fechado criado à custa de linhas poligonais leves:

```
(defun area-polilinha-leve (entidade)
  (area-poligono (vertices-polilinha-leve entidade)))
```

Uma vez calculada a área de um polígono arbitrário, apenas nos falta definir o *centro* (\bar{x}, \bar{y}) do polígono para podermos inserir o texto que descreve essa área.⁶⁵ O centro de um polígono pode-se definir por um processo idêntico ao do cálculo da área, somando, de forma ponderada, os centros dos vários trapézios:

$$\bar{x} = \frac{\sum_i \bar{x}_i A_i}{\sum_i A_i}$$

$$\bar{y} = \frac{\sum_i \bar{y}_i A_i}{\sum_i A_i}$$

Uma maneira simples de calcularmos simultaneamente \bar{x} e \bar{y} consiste em tratá-los como coordenadas (\bar{x}, \bar{y}) e empregarmos as operações de coordenadas `xy` e `+xy`.

Para começar, precisamos de definir o centro de um trapézio de lados a e b e base d . A fórmula matemática deste centro é

$$\bar{x} = \frac{d(2a + b)}{3(a + b)}$$

$$\bar{y} = \frac{a^2 + ab + b^2}{3(a + b)}$$

As fórmulas anteriores são válidas para trapézios com o canto inferior na origem e com lados a ou b não nulos. Para lidarmos com as situações em que tal não acontece, vamos definir uma função Lisp mais sofisticada:

⁶⁵Quando falamos de polígonos arbitrários é usual designar-se o seu centro pelo termo *centróide*.

```
(defun centro-trapezio (p0 p1 / a b d)
  (setq a (cy p0)
        b (cy p1)
        d (- (cx p1) (cx p0)))
  (if (= a b 0)
      (xy (+ (cx p0) (/ d 2))
           0)
      (xy (+ (cx p0)
              (/ (* d (+ (* 2 a) b))
                 (* 3 (+ a b))))
          (/ (+ (* a a) (* a b) (* b b))
             (* 3 (+ a b))))))
```

Para calcular cada um dos produtos $\bar{x}_i A_i$ e $\bar{y}_i A_i$ que ocorrem como termos do numerador das frações anteriores vamos definir a função centro-area-trapezio:

```
(defun centro-area-trapezio (p0 p1 / centro area)
  (setq centro (centro-trapezio p0 p1)
        area (area-trapezio p0 p1))
  (xy (* (cx centro) area)
      (* (cy centro) area)))
```

A função anterior trata de um só trapézio. Para fazer o somatório de todos os trapézios, definimos:

```
(defun centro-area-vertices (vertices)
  (if (null (cdr vertices))
      (xy 0 0)
      (+c (centro-area-trapezio (car vertices) (cadr vertices))
           (centro-area-vertices (cdr vertices)))))
```

Finalmente, para determinar o centro do polígono, temos de dividir o resultado calculado pela função anterior pela área total do polígono:

```
(defun centro-poligono (vertices / centro-area area)
  (setq vertices (append vertices (list (car vertices))))
  (setq centro-area (centro-area-vertices vertices)
        area (area-vertices vertices))
  (xy (/ (cx centro-area) area)
      (/ (cy centro-area) area)))
```

Uma vez que estamos particularmente interessados em tratar polígonos que, sabemos, o AutoCad implementa com linhas poligonais leves, vamos criar uma nova função que, a partir de uma entidade construída com uma linha poligonal leve, calcula o seu centro a partir dos seus vértices:

```
(defun centro-polilinha-leve (entidade)
  (centro-poligono (vertices-polilinha-leve entidade)))
```

Finalmente, falta apenas generalizar as funções area-entidade e centro-entidade para tratarem também as linhas poligonais leves:

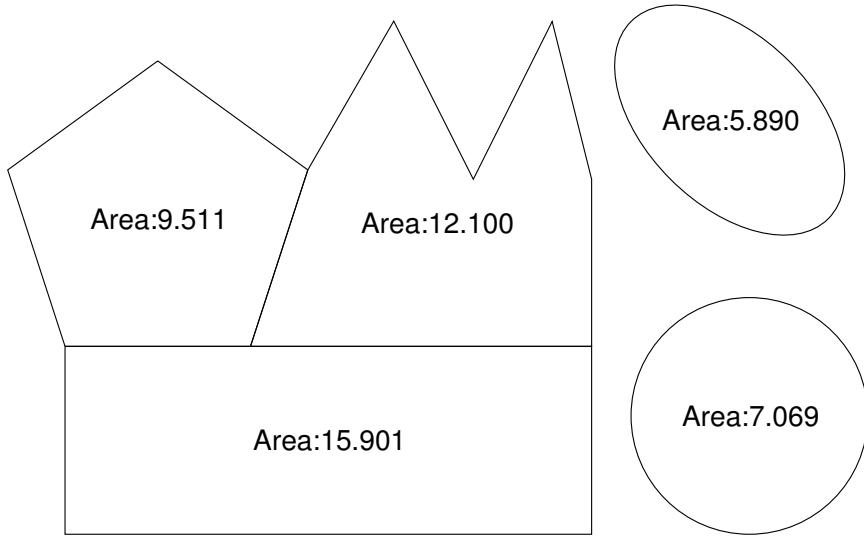


Figura 85: Inserção de áreas de entidades.

```
(defun area-entidade (entidade / tipo)
  (setq tipo (tipo-entidade entidade))
  (cond ((= tipo "CIRCLE")
         (area-entidade-circulo entidade))
        ((= tipo "ELLIPSE")
         (area-entidade-elipse entidade))
        ((= tipo "LWPOLYLINE")
         (area-polilinha-leve entidade)))))

(defun centro-entidade (entidade / tipo)
  (setq tipo (tipo-entidade entidade))
  (cond ((= tipo "CIRCLE")
         (obtem-propriedade centro: entidade))
        ((= tipo "ELLIPSE")
         (obtem-propriedade centro: entidade))
        ((= tipo "LWPOLYLINE")
         (centro-polilinha-leve entidade))))
```

É agora possível seleccionar entidades de tipos variados e inserir automaticamente a sua área, tal como se apresenta na Figura 85

Exercicio 9.7 Um semi-octaedro é uma pirâmide de base quadrangular que se define por cinco pontos: quatro para a base e um para o topo ou zénite.

Considere a criação de um comando AutoCad denominado `semi-octaedro` que pede ao utilizador as cinco coordenadas necessárias para definir um semi-octaedro e crie as linhas correspondentes às arestas do sólido.

9.18 Entidades com Informação Adicional

Vimos que o AutoCad nos permite obter um conjunto vasto de informação sobre as entidades que foram criadas, incluindo os pontos primários e secundários, as dimensões, o tipo de linha, a cor, etc. Vamos agora ver que o AutoCad também permite acrescentar informações adicionais a essas entidades, informações essas que são guardadas no próprio documento em que estamos a trabalhar, permitindo a sua recuperação numa sessão posterior.

Esta possibilidade é particularmente interessante para programas Auto Lisp que pretendam associar algum significado às entidades geométricas que estão a ser criadas. A título de exemplo, consideremos a planta de uma habitação onde se empregaram diferentes entidades geométricas para fazer a divisão do espaço em diferentes áreas. Do ponto de vista do AutoCad, essas entidades são meras figuras geométricas sem significado mas, para o arquitecto que idealiza a planta, cada uma representa um uso específico, como seja, um quarto, uma sala, uma cozinha. Para que esta informação não se perca, é necessária a associação deste uso a cada uma das entidades permitindo, por exemplo, que um programa Auto Lisp possa verificar se uma dada planta cumpre as normas legais relativas às dimensões mínimas que cada uso implica.

Antes de vermos como é que esta associação de informação pode ser feita é importante termos em conta que o AutoCad é agnóstico quanto ao que cada programa Auto Lisp possa querer associar a cada entidade, permitindo que diferentes programas o possam fazer em simultâneo. Este agnosticismo dá aos programas a liberdade de incluirem as informações que pretendem mas tem o potencial de permitir que um programa possa inadvertidamente usar e manipular informação que foi adicionada por outro programa diferente. Para evitar este problema, o AutoCad exige que a informação adicionada a uma entidade seja particionada de acordo com o programa que a está a adicionar. Para isso, temos de *registar* o programa, i.e., temos de lhe dar um nome que o distinga de outros programas e temos de informar o AutoCad acerca desse nome. Isso é feito através da função `regapp`.⁶⁶ Note-se que o nome é uma *string* que não pode ter mais de 31 caracteres e cuja sintaxe é semelhante à de um símbolo Lisp.

Para exemplificarmos, consideremos um programa Auto Lisp destinado a associar um determinado uso a cada entidade geométrica representada numa planta. Comecemos por baptizar este programa:

```
(regapp "USO")
```

A partir deste momento, podemos usar o nome "USO" para associar informações a qualquer entidade. Para isso, o AutoCad reserva um conjunto

⁶⁶Cada programa Auto Lisp é considerado pelo AutoCad como uma *aplicação*. Esta terminologia reflecte-se no nome da função `regapp` pois ele constitui uma contracção de *register application*.

Código	Significado
1000	Uma <i>string</i> até 255 caracteres.
1001	O nome de uma aplicação.
1002	O nome de um <i>layer</i> .
1010	Uma coordenada tridimensional.
1040	Um valor real.
1070	Um valor inteiro de 16 bits.
1071	Um valor inteiro de 32 bits.

Tabela 7: Códigos DXF para informações adicionais. Cada código representa um tipo particular de dados.

de códigos DXF que podem ser usados repetidamente para categorizar a informação associada. A Tabela 7 explica alguns desses códigos. Da observação da tabela concluímos, por exemplo, que para associarmos uma *string* a uma entidade temos de empregar o código DXF 1000. Vários outros códigos existem, incluindo códigos que permitem estruturar a informação em vários níveis de detalhe.

Uma vez que estes códigos e os valores associados têm de ser particionados por aplicação, a lista de associações tem de ser agrupada numa outra lista que contém o código DXF –3, indicador de que há informações adicionais associadas à entidade. Por exemplo, para atribuirmos o uso “sala” à última entidade criada temos de avaliar a seguinte expressão:

```
(entmod
  (list (cons -1 (entlast))
        (cons -3 (list
                   (cons "USO"
                         (list (cons 1000 "sala"))))))))
```

Para recuperar aquela informação, temos de ter em conta que, por omisão, a função `entget` não devolve a informação adicional e, para a obtermos, temos de especificar num parâmetro adicional uma lista contendo as aplicações cuja informação também pretendemos.

```
_S (assoc -3 (entget (entlast) (list "USO")))
(-3 ("USO" (1000 . "sala"))))
```

Obviamente, aquelas formas de associar e obter informações são muito pouco claras. Felizmente, é fácil escondermos a complexidade do processo dentro de duas funções que denominaremos `adiciona-informacao` e `obtem-informacao`.

Para facilitar ainda mais o uso da função `adiciona-informacao`, vamos adoptar uma abordagem idêntica à que descrevemos na Secção 9.5, em que empregámos uma lista de propriedades para conter os parâmetros da

função, lista essa que depois convertemos para uma lista de associações tal como a função `entmod` pretende:

```
(defun adiciona-informacao (entidade aplicacao plist)
  (entmod
    (list (cons -1 entidade)
          (cons -3 (list
                      (cons aplicacao (alist<-plist plist)))))))
```

Assim, a atribuição do uso “sala” à última entidade criada passa a ser feita da seguinte forma:

```
(adiciona-informacao (entlast) "USO" (list 1000 "sala"))
```

Para recuperarmos a informação que adicionámos a uma entidade vamos definir a função `obtem-informacao`. Esta função precisa de saber qual a aplicação e qual a propriedade específica dessa aplicação que foram usadas para guardar a informação. Tendo em conta a estrutura de listas de associação encadeadas que é usada para guardar informação adicional em entidades, a definição da função fica:

```
(defun obtem-informacao (entidade aplicacao propriedade)
  (cdr (assoc propriedade
               (cdadr (assoc -3
                             (entget entidade (list aplicacao)))))))
```

Assim, para obter o uso da última entidade criada, temos de escrever:

```
_ $ (obtem-informacao (entlast) "USO" 1000)
" sala"
```

Como o uso explícito de códigos DXF não ajuda à compreensão do que se está a fazer, podemos tornar os programas mais claros se “escondermos” esses códigos em variáveis com nomes apropriados. A seguinte atribuição define alguns desses códigos:

```
(setq tipo-string 1000
      tipo-xyz 1010
      tipo-real 1040
      tipo-inteiro 1071)
```

É claro que podemos simplificar ainda mais os nossos programas se criarmos funções ainda mais específicas, desenhadas apenas para atribuir e recuperar usos:

```
(defun atribui-uso (entidade uso)
  (adiciona-informacao entidade "USO" (list tipo-string uso)))
```

```
(defun obtem-uso (entidade)
  (obtem-informacao entidade "USO" tipo-string))
```

Para permitir a sua invocação a partir do AutoCad, podemos também definir um comando equivalente:

```
(defun c:atribui-uso ()
  (atribui-uso
    (car (entsel "Seleccione a entidade"))
    (getstring "Qual o uso da entidade seleccionada?")))
```

Através deste comando, o utilizador pode definir (e redefinir) o uso de qualquer entidade. Naturalmente, a definição do uso de pouco serve se não for possível recuperar essa informação. Para exemplificarmos a utilidade desta informação, podemos definir um comando AutoCad que itere por todas as entidades e escreva uma listagem dos seus usos e áreas:

```
(defun c:usos-e-areas (/ posicao-texto uso)
  (setq posicao-texto (getpoint "Posicione o texto..."))
  (foreach entidade (todas-entidades)
    (setq uso (obtem-uso entidade))
    (if uso
        (progn
          (cria-texto
            (list ponto-inicial: posicao-texto
                  texto: (strcat uso ":" (rtos (area-entidade entidade)))
                  altura: 1))
          (setq posicao-texto (+y posicao-texto -1.5))))))
```

A Figura 86 mostra um exemplo da utilização desta função numa planta com vários tipos de áreas.

9.19 Escadas

Uma escada é uma sucessão de degraus que estabelece uma ligação entre dois níveis. Os degraus são agrupados em *lanços* e os lanços são interligados por *patamares*. Cada degrau é caracterizado pelo comprimento c do seu *cobertor* e pela altura e do seu *espelho*, tal como se representa na Figura 88. Notemos, nessa figura, que uma escada com n cobertores tem $n + 1$ espelhos. Assim, dado um número de cobertores n , o comprimento l é dada por $l = nc$ mas a altura h vencida pelos correspondentes $n + 1$ espelhos é $h = (n + 1)e$.

Imaginemos agora que pretendemos definir um comando que nos permita criar escadas facilmente. Para isso, é recomendável dividirmos o problema em dois subproblemas:

1. Primeiro, definimos o programa Auto Lisp que, a partir dos parâmetros geométricos do problema, produz a sua solução. Este programa

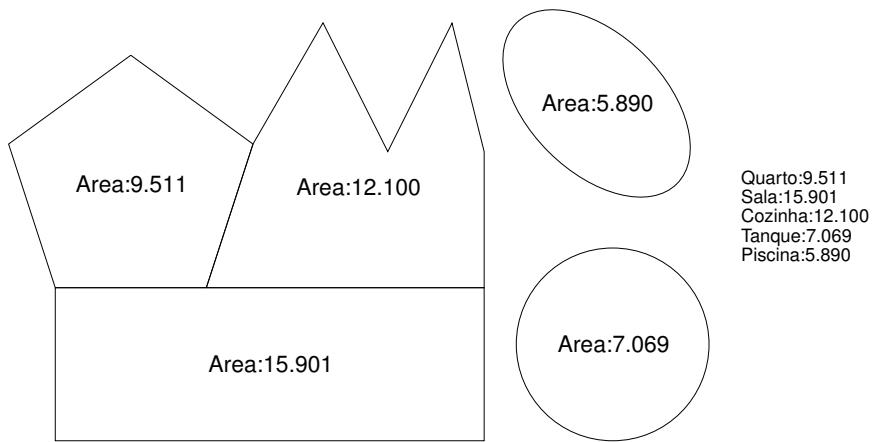


Figura 86: Uma planta com uma legenda produzida automaticamente a partir dos usos e áreas das entidades geométricas.



Figura 87: Uma escada na ilha de Santorini, na Grécia. Fotografia de Christopher Riess.

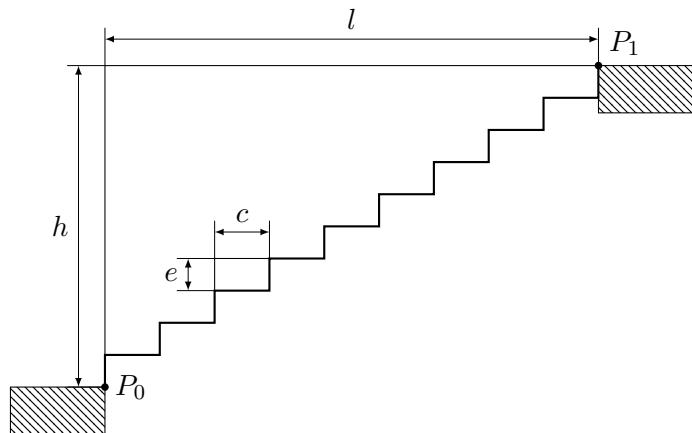


Figura 88: Parametrização de uma escada.

deve ser desenvolvido segundo uma perspectiva matemática e sem preocupações com o modo como o programa vai ser usado pelo utilizador.

2. Segundo, observamos o programa desenvolvido no ponto anterior e, a partir da determinação dos parâmetros necessários para o usar, definimos um comando AutoCad que pede esses parâmetros ao utilizador (segundo uma ordem que faça sentido) e, de seguida, invoca o programa anterior, passando-lhe os valores fornecidos pelo utilizador.

Seguindo esta metodologia, vamos começar por definir uma função que, a partir de um ponto P , do comprimento e altura de cada degrau e do número de degraus, desenha um lanço de escada. Como habitualmente, esta função pode ser decomposta recursivamente no desenho de um degrau seguido do desenho da restante escada. No caso de paragem desenhamos apenas o espelho do último degrau pois o cobertor correspondente já pertence ao patamar ou andar onde a escada termina.

Uma vez que o final de cada elemento da escada é ponto de partida para o elemento seguinte, será útil saber qual foi o ponto onde terminou o elemento anterior (seja degrau ou apenas espelho). Por este motivo, vamos definir as funções `degrau` e `espelho` de modo a devolverem o ponto final onde terminaram o desenho.

```
(defun degrau (p c e / pm pf)
  (setq pm (+xy p 0 e)
        pf (+xy p c e))
  (command "_line" p pm pf ""))
  pf)
```

```
(defun espelho (p c e / pf)
  (setq pf (+xy p 0 e))
  (command "_.line" p pf ""))
  pf)
```

Estas duas funções serão agora usadas para definir o caso recursivo e o caso final da função `lanco-escada`: um lanço de escada com n degraus é obtido através da criação de um degrau seguido da criação de um lanço de escada com $n - 1$ degraus a começar do ponto onde terminou o degrau anterior; um lanço de escada com 0 degraus é meramente um espelho.

```
(defun lanco-escada (p c e n)
  (if (= n 0)
    (espelho p c e)
    (lanco-escada (degrau p c e) c e (- n 1))))
```

A função `lanco-escada` resolve o problema geométrico mas não é prática para quem está no AutoCad e precisa de criar uma escada entre dois pontos. Para tornarmos a sua utilização mais simples, vamos criar um comando AutoCad que, a partir de dois pontos P_0 e P_1 e do número n de cobertores pretendidos, calcula a dimensão de cada degrau e invoca aquela função com os parâmetros adequados. Para isso, temos de calcular o tamanho do cobertor e do espelho a partir das distâncias horizontal l e vertical h entre aqueles pontos, o que podemos fazer a partir das fórmulas

$$\begin{cases} c = \frac{l}{n} \\ e = \frac{h}{n+1} \end{cases}$$

Assim, temos:

```
(defun c:lanco-escada (/ p0 p1 l h n)
  (setq p0 (getpoint "\nPonto inicial da escada?")
        p1 (getpoint "\nPonto final da escada?")
        l (- (cx p1) (cx p0))
        h (- (cy p1) (cy p0))
        n (getint "\nNúmero de cobertores?"))
  (lanco-escada
    p0
    (/ l n)
    (/ h (+ n 1))
    n))
```

Podemos agora invocar no AutoCad o comando `lanco-escada`, o que levará o AutoCad a pedir-nos que indiquemos com o cursor gráfico o ponto inicial e o ponto final da escada e, finalmente, o número de cobertores desejado. Em relação a este último parâmetro, o AutoCad irá responsabilizar-se por verificar e garantir que o utilizador fornece, de facto, um número inteiro.⁶⁷ A Figura 89 apresenta um conjunto de escadas criadas a partir de múltiplas invocações do comando AutoCad `lanco-escada`.

⁶⁷Para este problema, seria útil que se conseguisse garantir, não só que a resposta do uti-

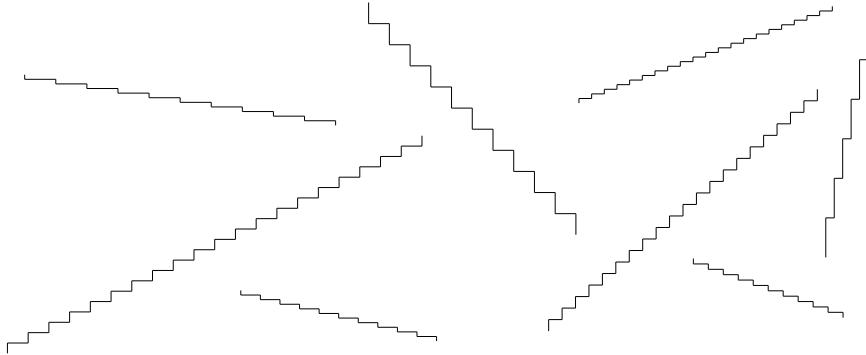


Figura 89: Escadas criadas por múltiplas invocações do comando AutoCad lanco-escada.

Obviamente, podemos usar a função `lanco-escada` para construir escadas mais complexas, compostas por vários lanços de escadas separados por patamares, tal como se apresenta na Figura 90. Para isso, mais uma vez, vamos começar por resolver o problema geométrico.

Em primeiro lugar, vamos definir a função que desenha um patamar a partir de um ponto (que deverá corresponder ao final de um espelho) e que devolve o ponto final do patamar:

```
(defun patamar (p lpatamar / pf)
  (setq pf (+xy p lpatamar 0))
  (command "_.line" p pf "")
  pf)
```

Em segundo lugar, vamos definir a função `lancos-escada` que, a partir de um ponto de partida p , de um comprimento de cobertor c , de uma altura de espelho e , de um número de cobertores n por lance, de um número total de lances e do comprimento de cada patamar, desenha a escada separando-a em vários lances. A função terá de distinguir o caso em que só tem de desenhar um lance de escadas daquele em que tem de desenhar mais do que um. Neste último caso, a função terá que desenhar também um patamar:

```
(defun lancos-escada (p c e n lances lpatamar)
  (if (= lances 1)
      (lanco-escada p c e n)
      (lancos-escada
        (patamar (lanco-escada p c e n) lpatamar)
        c e n (- lances 1) lpatamar)))
```

A Figura 91 ilustra o uso desta função. As escadas aí representadas foram geradas pelas seguintes expressões:

lizador corresponde a um número inteiro, mas também que esse número inteiro é *positivo*. A função `getint` apenas garante a primeira parte.

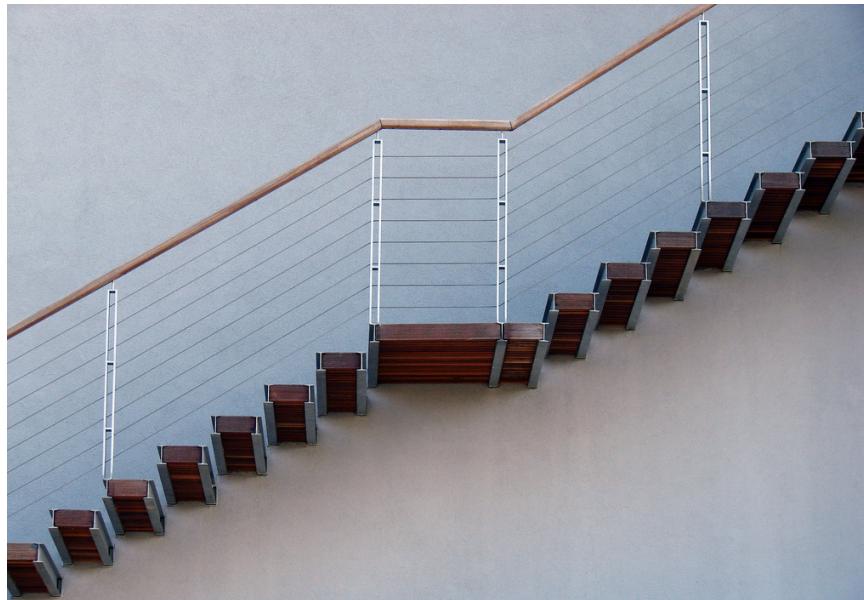


Figura 90: Uma escada composta por dois lanços com um patamar intermédio. Fotografia de Mateusz Szczerbinski.

```
(lancos-escada (xy 0.0 0.0) 0.26 0.16 10 4 1.0)
(lancos-escada (xy 10.0 0.0) 0.26 0.36 10 2 2.0)
(lancos-escada (xy 13.0 0.0) 0.16 0.12 8 7 0.5)
```

Tal como aconteceu com a função `lanco-escada`, podemos tornar a função `lancos-escada` mais acessível ao utilizador de AutoCad através da definição de um comando que pede os parâmetros geométricos necessários, calcula os outros que deles dependem e, finalmente, invoca a função:

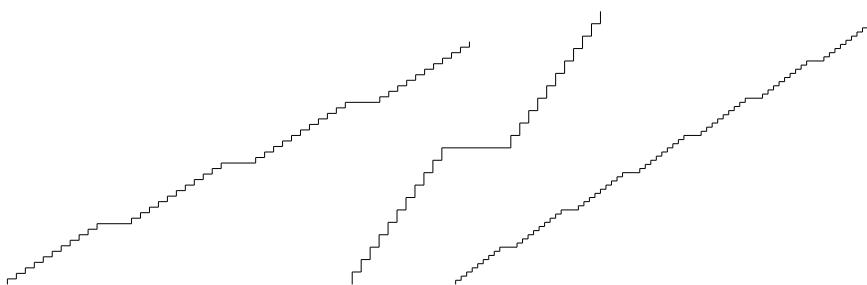
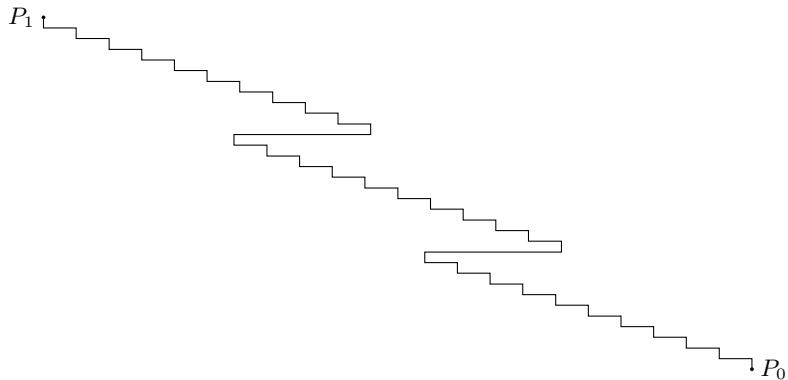


Figura 91: Lanços de escada criados por múltiplas invocações da função Auto Lisp `lancos-escada`.

```
(defun c:lancos-escada (/ p0 p1 h l lancos lpatamar n ntotal)
  (setq p0 (getpoint "\nPonto inicial da escada?"))
  (p1 (getpoint "\nPonto final da escada?"))
  (h (- (cy p1) (cy p0)))
  (l (- (cx p1) (cx p0)))
  (lancos (getint "\nNúmero de lanços?"))
  (lpatamar (getreal "\nComprimento do patamar?"))
  (n (getint "\nNúmero de cobertores por lanço?"))
  (ntotal (* lancos n))
  (lancos-escada
    p0
    (/ (- l (* (- lancos 1) lpatamar)) ntotal)
    (/ h (* lancos (+ n 1))))
    n
    lancos
    lpatamar))
```

Exercicio 9.8 A função `c:lancos-escada` apresenta um comportamento bizarro quando o ponto inicial P_0 se encontra à direita do ponto final P_1 . Neste caso, tal como se pode ver na figura abaixo, a escada desenhada fica com um patamar que se desloca na direcção contrária.



Modifique a função `c:lancos-escada` de modo a que o patamar fique orientado na mesma direcção da escada, a menos que o utilizador forneça um comprimento de patamar negativo, caso em que o patamar deverá ficar com a direcção oposta.

9.19.1 Dimensionamento de Escadas

Como é fácil de observar nas Figuras 89 e 91, nem todas as escadas são suficientemente ergonómicas para poderem ser usadas por seres humanos pois é necessário, por exemplo, que o espelho de um degrau não seja tão grande que impeça a pessoa de o conseguir subir ou tão pequeno que se torne desconfortável. Tendo em conta as dimensões humanas, é razoável convencionar que o espelho e tem de estar compreendido entre um valor mínimo e_{\min} e um valor máximo e_{\max} (que, para um adulto, se convicia que $e_{\min} = 0.14m$ e $e_{\max} = 0.18m$). Isto implica que o conjunto de equações

que regulam as dimensões de uma escada passa a ser

$$\begin{cases} c = \frac{l}{n} \\ e = \frac{h}{n+1} \\ e_{\min} \leq e \leq e_{\max} \end{cases}$$

Para além disto é ainda necessário ter em conta que o número de cobertores n tem de ser um número inteiro.

Estas restrições tornam mais difícil o dimensionamento de uma escada pois impedem-nos de usar um número arbitrário de cobertores. Felizmente, podemos usar o Auto Lisp e desenvolver um algoritmo que calcula quais são os números aceitáveis apenas a partir do comprimento l e altura h da escada. Para isso, substituímos a segunda equação na terceira, ficando

$$e_{\min} \leq \frac{h}{n+1} \leq e_{\max}$$

ou seja

$$\frac{h}{e_{\max}} - 1 \leq n \leq \frac{h}{e_{\min}} - 1$$

Para incluirmos a restrição de que n tem de ser um número inteiro, basta-nos acrescentar os arredondamentos adequados:

$$\left\lceil \frac{h}{e_{\max}} - 1 \right\rceil \leq n \leq \left\lfloor \frac{h}{e_{\min}} - 1 \right\rfloor$$

Assim, dada uma altura h e as alturas mínimas e_{\min} e máximas e_{\max} do espelho, podemos obter o número mínimo e máximo de cobertores. Por exemplo, consideremos uma escada que necessita de vencer uma altura $h = 3.55\text{m}$ usando degraus com um espelho e entre $e_{\min} = 0.14\text{m}$ e $e_{\max} = 0.18\text{m}$. Substituindo os valores na inequação anterior, temos $\lceil 18.7 \rceil \leq n \leq \lfloor 24.4 \rfloor$, i.e., poderemos usar 19, 20, 21, 22, 23 ou 24 cobertores. Estamos então em condições de implementar uma função que nos devolve um par com este número mínimo e máximo de cobertores:

```
(defun cobertores (h emin emax / nmin nmax)
  (setq nmin (ceiling (- (/ h emax) 1)))
    nmax (fix (- (/ h emin) 1)))
  (cons nmin nmax))
```

Podemos agora confirmar o número mínimo e máximo de cobertores:

```
_\$ (cobertores 3.55 0.14 0.18)
(19 . 24)
```

Infelizmente, a função `cobertores` não tem em conta todas as restrições ergonómicas: para além das limitações na altura do espelho e de um

degrau é ainda necessário termos em conta que a dimensão do cobertor c não pode ser tão pequena que não permita apoiar um pé, i.e., temos de ter $c \geq c_{\min}$ (em termos humanos, é usual considerar $c_{\min} = 0.23m$). Como $c = \frac{l}{n}$, temos que $\frac{l}{n} \geq c_{\min}$, ou seja, $n \leq \frac{l}{c_{\min}}$. Mais uma vez, tendo em conta que n tem de ser um número inteiro, ficamos com $n \leq \lfloor \frac{l}{c_{\min}} \rfloor$.

Para exemplificar, voltemos a considerar a escada de altura $h = 3.55m$, para a qual tínhamos encontrado um número de cobertores n entre 19 e 24 e acrescentemos agora que a escada tem também de cobrir um comprimento $l = 5.10m$. Substituindo os valores, temos $n \leq \lfloor \frac{5.10}{0.23} \rfloor$ ou seja, $n \leq \lfloor 22.1 \rfloor$. Isto implica que, do conjunto de possibilidades anterior, apenas podemos escolher entre usar 19, 20, 21 ou 22 cobertores. No entanto, se o comprimento fosse um metro mais pequeno, i.e., $l = 4.10m$, teríamos $n \leq \lfloor 17.8 \rfloor$ que, infelizmente, não seria possível compatibilizar com o intervalo de variação anterior. Nesse caso, deveríamos alertar o utilizador para o problema e, para sermos simpáticos, poderíamos também dizer-lhe qual é o comprimento mínimo da escada que satisfaz a restrição. A seguinte redefinição da função `cobertores` tem estes aspectos em conta.

```
(defun cobertores (h emin emax l cmin / nmin nmax)
  (setq nmin (ceiling (- (/ h emax) 1)))
    nmax (fix (- (/ h emin) 1))
    ncmax (fix (/ l cmin)))
  (if (< ncmax nmin)
    (alerta
      (strcat "Não é possível dimensionar o cobertor."
        "\nAumente o comprimento para "
        (rtos (* nmin cmin)))))

  (progn
    (setq nmax (min nmax ncmax))
    (cons nmin nmax))))
```

Para alertar o utilizador vamos simplesmente escrever a *string* argumento.

```
(defun alerta (str)
  (princ str)
  (princ "\n")
  nil)
```

Podemos agora confirmar os resultados que obtivemos para os exemplos anteriores, o primeiro para um comprimento de 5.1m, o segundo para um comprimento de 4.1m:

```
_ $ (cobertores 3.55 0.14 0.18 5.1 0.23)
(19 . 22)
_ $ (cobertores 3.55 0.14 0.18 4.1 0.23)
Não é possível dimensionar o cobertor.
Aumente o comprimento para 4.3700
nil
```

Note-se que, para $l = 4.1m$, a função `cobertores` não consegue cal-

cular o número de cobertores mas sugere uma medida para a qual já o conseguiria fazer. Podemos então experimentar essa medida:

```
_\$ (cobertores 3.55 0.14 0.18 4.37 0.23)
(19 . 19)
```

A última expressão permite-nos dimensionar uma escada de 20 degraus (19 cobertores), cada um com um cobertor de $c = \frac{4.37}{19} = 0.23\text{m}$ e espelho de $e = \frac{3.55}{19+1} = 0.1775\text{m}$.

Apesar de bastante mais ergonómicas, as nossas escadas não são ainda totalmente ergonómicas pois é sabido que o uso confortável de uma escada depende de uma relação de proporção entre o cobertor e o espelho de cada degrau. Esta proporção é conhecida desde o século dezassete e resultou das investigações conduzidas pelo arquitecto François Blondel⁶⁸

... à chaque fois qu'on s'élève d'un pouce, la valeur de la partie horizontale se trouve réduite de deux pouces et que la somme de la hauteur doublée de la marche et de son giron doit demeurer constante et être de deux pieds...

Os “deux pieds” a que Blondel faz referência designam o *módulo b* do degrau. Uma vez que o módulo *b* depende do tamanho da pessoa que usa a escada, não existe uma medida exacta para o seu valor *e*, por isso, considera-se apenas que ele deve estar contido entre dois extremos b_{\min} e b_{\max} . Por exemplo, para uma pessoa adulta, é usual convencionar que $b_{\min} = 0.62\text{m}$ e $b_{\max} = 0.64\text{m}$. Assim sendo, as observações de Blondel escrever-se-iam modernamente na forma:

$$b_{\min} \leq c + 2e \leq b_{\max}$$

Se aplicarmos a regra de Blondel ao degrau que calculámos atrás (que tinha $c = 0.23\text{m}$ e $e = 0.1775\text{m}$), obtemos $c + 2e = 0.585$ que, infelizmente, está longe do intervalo ideal $[0.62, 0.64]$.

Para complementarmos o nosso algoritmo com a regra de Blondel vamos começar por traduzi-la em termos do número de cobertores, usando as relações $c = \frac{l}{n}$ e $e = \frac{h}{n+1}$:

$$b_{\min} \leq \frac{l}{n} + 2 \frac{h}{n+1} \leq b_{\max}$$

Multiplicando ambos os lados por $n(n+1)$, obtemos

$$b_{\min}(n^2 + n) \leq l(n+1) + 2hn \leq b_{\max}(n^2 + n)$$

⁶⁸Nicolas-François Blondel foi um famoso arquitecto, engenheiro e matemático francês do século dezassete. Para além de ter projectado obras arquitecturais importantes, publicou o “*Cours d’architecture enseigné à l’Académie royale d’architecture*” onde descreveu as suas descobertas sobre o dimensionamento de escadas.

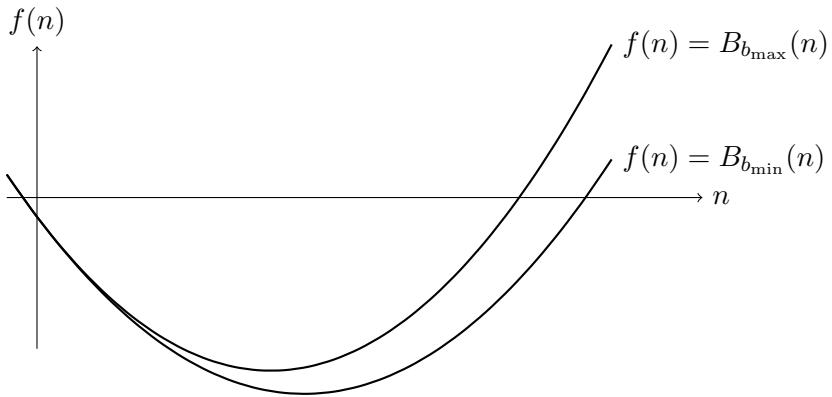


Figura 92: Curvas da “função de Blondel” para $b = b_{\min}$ e $b = b_{\max}$.

ou seja

$$b_{\min}n^2 - (l + 2h - b_{\min})n - l \leq 0 \leq b_{\max}n^2 - (l + 2h - b_{\max})n - l$$

Esta desigualdade caracteriza o número de cobertores n em termos dos limites b_{\min} e b_{\max} do módulo de Blondel. Uma vez que as expressões apenas diferem no módulo b que usam (que será b_{\min} para uma e b_{\max} para a outra), podemos definir a “função de Blondel”

$$B_b(n) = bn^2 - (l + 2h - b)n - l$$

e reescrever a desigualdade como

$$B_{b_{\min}}(n) \leq 0 \leq B_{b_{\max}}(n)$$

Sendo a “função de Blondel” quadrática em n e sendo que o coeficiente de segundo grau da equação (b) é sempre positivo, a curva da função é uma parábola com a concavidade para cima, tal como se apresenta na Figura 92.

O número de cobertores que satisfaz a equação de Blondel ficará então contido no intervalo em que a curva $B_{b_{\min}}(n)$ assume valores negativos e a curva $B_{b_{\max}}(n)$ assume valores positivos. Para determinarmos os limites desse intervalo temos de identificar as raízes das funções, i.e., os valores de n para os quais as duas funções se tornam zero e as respectivas curvas cruzam o eixo das abscissas. Assim, temos de resolver a equação

$$B_b(n) = bn^2 - (l + 2h - b)n - l = 0$$

Uma vez que se trata de uma equação do segundo grau, podemos aplicar a fórmula resolvente para a resolver em ordem a n . Assim, temos

$$n = \frac{l + 2h - b \pm \sqrt{(l + 2h - b)^2 + 4bl}}{2b}$$

Das duas raízes, há uma que é obviamente impossível pois, sendo $b > 0$ e $l > 0$, $l + 2h - b$ é estritamente menor que $\sqrt{(l + 2h - b)^2 + 4bl}$ e, consequentemente, a sua subtração corresponderia a um número de cobertores negativo, tal como é visível no gráfico da função. Assim, temos que

$$n = \frac{l + 2h - b + \sqrt{(l + 2h - b)^2 + 4bl}}{2b}$$

Dado que b é o coeficiente de segundo grau da equação e b é sempre positivo, a desigualdade $B_{b_{\min}}(n) \leq 0 \leq B_{b_{\max}}(n)$ é equivalente a

$$\begin{cases} n \geq \frac{l+2h-b_{\max}+\sqrt{(l+2h-b_{\max})^2+4b_{\max}l}}{2b_{\max}} \\ n \leq \frac{l+2h-b_{\min}+\sqrt{(l+2h-b_{\min})^2+4b_{\min}l}}{2b_{\min}} \end{cases}$$

Para calcular os extremos do intervalo anterior, vamos começar por definir a função que calcula o número de cobertores n a partir da fórmula de Blondel para o módulo b :

```
(defun cobertores-blondel (h l b / l+2h-b)
  (setq l+2h-b (+ l (* 2 h) (- b)))
  (/ (+ l+2h-b (sqrt (+ (quadrado l+2h-b) (* 4 b 1))))))
  (* 2 b)))
```

Para exemplificar, voltemos a considerar a escada de altura $h = 3.55\text{m}$ e comprimento $l = 5.10\text{m}$ para a qual tínhamos encontrado um número de cobertores n entre 19 e 22. Relativamente ao módulo máximo e mínimo, vamos usar a convenção usual $b_{\min} = 0.62\text{m}$ e $b_{\max} = 0.64\text{m}$. Usando a função anterior, temos:

```
_\$ (cobertores-blondel 3.55 5.10 0.64)
18.4934
_\$ (cobertores-blondel 3.55 5.10 0.62)
19.1079
```

Substituindo, temos $18.5 \leq n \leq 19.1$ ou seja, dos valores de n que tínhamos, apenas o primeiro satisfaz todos os requisitos. Assim, para este exemplo, temos $n = 19$ e, consequentemente, ficamos com degraus cujo cobertor c e espelho e serão

$$\begin{cases} c = \frac{5.10}{19} = 0.2684 \\ e = \frac{3.55}{19+1} = 0.1775 \end{cases}$$

Infelizmente, na prática, a partir de um comprimento l e altura h arbitrários, dificilmente conseguiremos encontrar valores de n que satisfaçam todas as restrições. Por exemplo, se retirarmos 5cm à altura anterior, i.e., se

a nossa escada passar a ter de vencer $h = 3.50m$, os intervalos de variação do número de cobertores n passam a ser

$$\begin{cases} 18.4 \leq n \leq 23.5 \\ n \leq 22.2 \\ 18.37 \leq n \leq 18.95 \end{cases}$$

Uma vez que n tem de ser um número inteiro, a terceira desigualdade não é satisfazível. Isto implica que, tal como anteriormente, o nosso algoritmo poderá não conseguir dimensionar a escada e, nesse caso, deverá avisar o utilizador de qual deverá ser o comprimento adequado. Para isso, recuperamos a fórmula de Blondel

$$b_{\min} \leq \frac{l}{n} + 2 \frac{h}{n+1} \leq b_{\max}$$

e resolvêmo-la em relação a l :

$$n(b_{\min} - 2 \frac{h}{n+1}) \leq l \leq n(b_{\max} - 2 \frac{h}{n+1})$$

Assim, podemos definir:

```
(defun comprimento-blondel (h b n)
  (* n (- b (/ (* 2 h) (+ n 1)))))
```

Podemos agora redefinir a função `cobertores` para ter em conta a regra de Blondel, não só garantindo que a regra permite encontrar um número inteiro de cobertores mas ainda que esse número está contido no intervalo de variação determinado pelas restrições nas dimensões do espelho e do cobertor. Para isso, temos de acrescentar as variáveis locais `nbmin` e `nbmax`, que representam os limites do número de cobertores que satisfazem a regra de Blondel, bem como as variáveis `nfinalmin` e `lfinalmax` que irão conter os limites finais após satisfação de todas as restrições.

```

(defun cobertores (h emin emax l cmin bmin bmax /
                     nmin nmax ncmax nbmin nbmax
                     nfinalmax nfinalmin)
  (setq nmin (ceiling (- (/ h emax) 1)))
    nmax (fix (- (/ h emin) 1))
    ncmax (fix (/ l cmin)))
  (if (< ncmax nmin)
    (alerta
      (strcat "Não é possível dimensionar o cobertor."
              "\nAumente o comprimento para "
              (rtos (* nmin cmin))))
    (progn
      (setq nmax (min nmax ncmax)
            nbmin (cobertores-blondel h l bmax)
            nbmax (cobertores-blondel h l bmin))
      (if (> (ceiling nbmin) (fix nbmax))
        (alerta
          (strcat "Não é possível respeitar a regra de Blondel."
                  "\nMude o comprimento para ser maior que "
                  (rtos (comprimento-blondel h bmin (ceiling nbmax)))
                  " ou menor que "
                  (rtos (comprimento-blondel h bmax (fix nbmin)))))

        (progn
          (setq nfinalmin (max nmin (ceiling nbmin))
                nfinalmax (min nmax (fix nbmax)))
          (if (> nfinalmin nfinalmax)
            (alerta
              (strcat "Não é possível respeitar a regra de Blondel."
                      "\nMude o comprimento para ser maior que "
                      (rtos (comprimento-blondel h bmin nmin))
                      " e menor que "
                      (rtos (comprimento-blondel h bmax nmax))))
            (cons nfinalmin nfinalmax)))))))

```

A seguinte interacção mostra o processo iterativo de dimensionamento de uma escada com altura $h = 2.8\text{m}$ e comprimento inicial $l = 3.0\text{m}$ (sujeita às restrições habituais $e_{\min} = 0.14\text{m}$, $e_{\max} = 0.18\text{m}$, $c_{\min} = 0.23\text{m}$, $b_{\min} = 0.62\text{m}$ e $b_{\max} = 0.64$):

```

_§ (cobertores 2.8 0.14 0.18 3.0 0.23 0.62 0.64)
Não é possível dimensionar o cobertor.
Aumente o comprimento para 3.4500
nil
_§ (cobertores 2.8 0.14 0.18 3.45 0.23 0.62 0.64)
Não é possível respeitar a regra de Blondel.
Mude o comprimento para ser maior que 3.4533 ou menor que 3.1200
nil
_§ (cobertores 2.8 0.14 0.18 3.46 0.23 0.62 0.64)
Não é possível respeitar a regra de Blondel.
Mude o comprimento para ser maior que 4.0500 e menor que 4.3500
nil
_§ (cobertores 2.8 0.14 0.18 4.06 0.23 0.62 0.64)
(15 . 15)

```

Reparemos que foi necessário passar sucessivamente o comprimento da escada do valor inicial de 3.00m para 3.45m, 3.46m e, finalmente, 4.06m.

Naturalmente, é um pouco irritante ter de fazer estas iterações manualmente. Para o evitarmos, podemos convencionar que, no caso de se violar alguma restrição ergonómica, o algoritmo invoca-se recursivamente com o comprimento maior da escada que evita essa violação. No final, o algoritmo devolverá o comprimento final da escada e o número médio de cobertores para esse comprimento. Há apenas mais um detalhe: temos de ter cuidado com os potenciais erros de arredondamento que podem fazer com que o comprimento que o algoritmo sugere acabe por ser insuficiente e, por isso, obrigar o algoritmo a sugerir-lo de novo, entrando em recursão infinita. Para contornar este problema, podemos simplesmente adicionar um milímetro (ou, se quisermos mais precisão, uma décima ou centésima de milímetro) ao comprimento sugerido.

A seguinte função `comprimento-cobertores` implementa este algoritmo:

```

(defun comprimento-cobertores (h emin emax l cmin bmin bmax /
                                nmin nmax ncmax nbmin nbmax
                                nfinalmax nfinalmin)
  (setq nmin (ceiling (- (/ h emax) 1)))
    nmax (fix (- (/ h emin) 1))
    ncmax (fix (/ l cmin)))
  (if (< ncmax nmin)
    (comprimento-cobertores
      h emin emax
      (+ 0.001 (* nmin cmin))
      cmin bmin bmax)
    (progn
      (setq nmax (min nmax ncmax)
            nbmin (cobertores-blondel h l bmax)
            nbmax (cobertores-blondel h l bmin))
      (if (> (ceiling nbmin) (fix nbmax))
        (comprimento-cobertores
          h emin emax
          (+ 0.001 (comprimento-blondel h bmin (ceiling nbmax)))
          cmin bmin bmax)
        (progn
          (setq nfinalmin (max nmin (ceiling nbmin))
                nfinalmax (min nmax (fix nbmax)))
          (if (> nfinalmin nfinalmax)
            (comprimento-cobertores
              h emin emax
              (+ 0.001 (comprimento-blondel h bmin nmin))
              cmin bmin bmax)
            (cons l (round (/ (+ nfinalmin nfinalmax) 2))))))))))

```

Usando esta função é agora possível calcular automaticamente a “melhor” escada (em termos dos critérios ergonómicos que adoptámos). Por exemplo, a escada anterior pode ser dimensionada partindo de um comprimento 0.0:

```
_\$ (comprimento-cobertores 2.8 0.14 0.18 0.0 0.23 0.62 0.64)
(4.051 . 15)
```

Exercicio 9.9 A função `cobertores` começa por calcular o número mínimo e máximo de cobertores a partir da altura h a vencer e das dimensões máxima e mínima do espelho (e_{\max} e e_{\min}):

$$\left\lceil \frac{h}{e_{\max}} - 1 \right\rceil \leq n \leq \left\lfloor \frac{h}{e_{\min}} - 1 \right\rfloor$$

As dimensões máxima e mínima do espelho dependem, naturalmente, da legislação aplicável. No caso português, por exemplo, o Regulamento Geral de Edificações Urbanas refere:

Os degraus da escada não poderão ter largura inferior a 23 centímetros, não contando o focinho; a sua altura deve ficar compreendida entre 14 e 18 centímetros;

Infelizmente, ao tentarmos dimensionar uma escada com uma altura de 55 centímetros de altura, encontramos a seguinte impossibilidade:

$$\left\lceil \frac{0.55}{0.18} - 1 \right\rceil \leq n \leq \left\lfloor \frac{0.55}{0.14} - 1 \right\rfloor$$

ou seja

$$3 \leq n \leq 2$$

Redefina a função `cobortores` de modo a detectar estas impossibilidades e a sugerir alturas que as evitem.

9.19.2 Dimensionamento de Lanços

Embora o nosso algoritmo para determinação do número de cobortores de uma escada seja razoavelmente sofisticado, convém recordar que ele apenas implementa algumas das regras arquitectónicas em uso e que há ainda mais regras que deveríamos ter em conta. Por exemplo, em geral, considera-se que não é aceitável ter um lanço de escadas a vencer uma altura excessivamente grande⁶⁹, sendo obrigatório “partir” a escada em vários lanços separados por patamares com uma dimensão adequada (normalmente, correspondente a um passo, i.e., à dimensão de Blondel, adicionado de um cobertor).

Para incorporarmos mais esta regra, temos de considerar a altura da escada e vermos quantos lances ela pode conter. Para isso, basta-nos dividir a altura da escada pela altura máxima dos lances e arredondar o resultado para cima. A altura actual de cada lance será então determinada pela divisão da altura total da escada pelo número de lances. Dispondo da altura de um lance, podemos dimensioná-lo usando a função `comprimento-cobortores` e, com base no comprimento e número de cobortores calculado para cada lance (bem como na altura do lance), podemos calcular a dimensão do cobertor e do espelho. Finalmente, podemos usar a função `lancos-escada` para desenhar a escada pretendida. Todos estes cálculos podem ser realizados pela seguinte função:

```
(defun dimensiona-escada (p0 p1 hmax emin emax cmin bmin bmax lpatamar /
                           h nlancos hlenco dims llanco clanco c e)
  (setq h (- (cy p1) (cy p0)))
  nlancos (ceiling (/ h hmax))
  hlenco (/ h nlancos)
  dims (comprimento-cobortores hlenco emin emax 0.0 cmin bmin bmax)
  llanco (car dims)
  clanco (cdr dims)
  c (/ llanco clanco)
  e (/ hlenco (+ clanco 1)))
(lancos-escada
 (+xy p1 (- 0.0 (* nlancos llanco) (* (- nlancos 1) lpatamar)) (- h))
   c e clanco nlancos lpatamar))
```

Para facilitar a utilização da função, vamos definir o seguinte comando que pede toda a informação necessária:

⁶⁹Equivalentemente, alguns regulamentos limitam o tamanho dos lanços de escadas em termos do seu número de degraus.

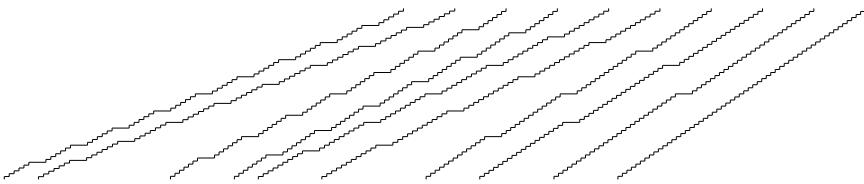


Figura 93: Lanços de escada criados por múltiplas invocações da função Auto Lisp dimensiona-escada. As escadas vencem todas a mesma altura mas o número de lanços diminui da esquerda para a direita.

```
(defun c:dimensiona-escada (/ p0 p1 hmax emin emax cmin bmin bmax lpatamar)
  (setq p0 (getpoint "\nPonto inicial da escada? (móvel na horizontal)")
        p1 (getpoint "\nPonto final da escada? (fixo)")
        hmax (getreal "\nAltura máxima por lance?")
        emin (getreal "\nAltura mínima do degrau?")
        emax (getreal "\nAltura máxima do degrau?")
        cmin (getreal "\nComprimento mínimo do degrau?")
        bmin (getreal "\nDimensão mínima de Blondel?")
        bmax (getreal "\nDimensão máxima de Blondel?")
        lpatamar (getreal "\nComprimento do patamar"))
  (dimensiona-escada p0 p1 hmax emin emax cmin bmin bmax lpatamar))
```

A Figura 93 mostra uma sucessão de escadas que vencem a mesma altura mas com alturas máximas de lance sucessivamente crescentes.

Exercicio 9.10 A função `c:dimensiona-escada` tem um inconveniente grave: sempre que é invocada, pede todos os parâmetros necessários ao utilizador, embora muitos desses parâmetros (como, por exemplo, e_{\min} , e_{\max} , c_{\min} , b_{\min} e b_{\max}) sejam tendencialmente constantes.

Redefina a função `c:dimensiona-escada` de modo a que os parâmetros possam ter valores de omissão que o utilizador só especifica se assim o pretender.

9.19.3 “El Castillo”

Embora actualmente as escadas sejam objecto de um conjunto relativamente grande de regras, nem sempre isso aconteceu e existem inúmeros exemplos no passado de escadas desenhadas para fins muito mais importantes do que meramente o de permitir deslocar pessoas entre diferentes níveis.

Um bom exemplo ocorre no templo de Cuculcán, em Chichén Itzá no México, apresentado na Figura 94. Cuculcán é uma divindade da mitologia Maia, representada por uma serpente emplumada. O templo a ela dedicado terá sido construído há mais de mil e cem anos e foi denominado “El Castillo” pelos conquistadores espanhóis quando encontraram as ruínas de Chichén Itzá. As escadas que ladeiam cada um dos quatro lados do templo têm, cada uma, 91 degraus e vários historiadores atribuem a este número um significado cronológico: as quatro escadas totalizam 364 degraus ao



Figura 94: “El Castillo,” o templo de Cuculcán localizado em Chichén Itzá, no México. Fotografia de Filippo Manaresi.

qual se soma um degrau final no topo do templo, correspondendo aos 365 dias do ano. Outros especialistas argumentam que as escadas teriam sido desenhadas para provocarem interessantes efeitos sonoros através dos diferentes ecos que os seus degraus provocariam. Finalmente, atendendo a que estas escadas vencem um comprimento e altura de 24 metros, há quem argumente que esta elevadíssima inclinação se destinava a produzir um efeito óptico de elevação aos céus.

Em qualquer caso, é interessante constatar que as escadas do templo de Cuculcán não seriam aprovadas pelas modernas regras de dimensionamento. Podemos facilmente confirmar isto através da seguinte interacção:

```
_\$ (cobertores 24.0 0.14 0.18 24.0 0.23 0.62 0.64)
Não é possível dimensionar o cobertor.
Aumente o comprimento para 30.5900
nil
_\$ (cobertores 24.0 0.14 0.18 31.0 0.23 0.62 0.64)
Não é possível respeitar a regra de Blondel.
Mude o comprimento para ser maior que 34.8182 e menor que 38.1156
nil
_\$ (cobertores 24.0 0.14 0.18 35.0 0.23 0.62 0.64)
(133 . 133)
```

Como podemos ver, para que as escadas do “Castillo” obedecam aos modernos critérios ergonómicos, será necessário passar o número de degraus de 91 para 134 e passar o comprimento de 24m para 35m. Ainda assim, as escadas continuariam a violar a regra que proíbe lanços com uma altura excessiva, pelo que seria necessário introduzir patamares intermédios. Se admitirmos que cada lanço não pode exceder os 3.5m de altura, então serão necessários 6 patamares, fazendo o comprimento final da escada atingir cerca de 43 metros.

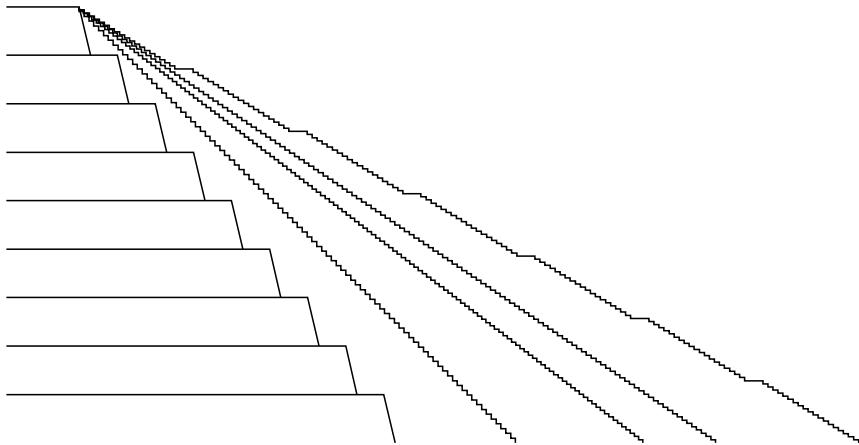


Figura 95: Esquema de “El Castillo.” Da esquerda para a direita, encontrase (1) a escada original do templo, (2) a escada que possui um cobertor mínimo de 0.23m e espelho entre 0.14m e 0.18m, (3) a escada que, para além disso, obedece à regra de Blondel (com limites entre 0.62m e 0.64m) e, finalmente, (4) a escada que, para além das restrições anteriores, possui lanços que não excedem os 3.5m de altura.

Para compararmos a evolução destas escadas, a Figura 95 mostra um esquema do templo de Cuculcán onde, da esquerda para a direita, mostramos as escadas originais do templo, as escadas que respeitam o comprimento mínimo do cobertor, as escadas que respeitam também a regra de Blondel e, finalmente, as escadas finais que respeitam ainda a altura máxima dos lanços.

9.20 Caixas de Escada

Como vimos nas secções anteriores, a função `c : dimensiona-escada` aumenta o comprimento da escada tanto quanto for necessário para respeitar as restrições ergonómicas. Isto implica que se a altura a vencer for muito elevada, como acontece com “El Castillo,” a escada resultante pode ficar com um comprimento enorme.

Infelizmente, nem sempre é possível aumentar o comprimento de uma escada. Na maioria das situações, na realidade, quer o espaço horizontal quer o espaço vertical disponíveis para o desenvolvimento de uma escada estão limitados (ou é desejável que estejam) e a solução para vencer alturas elevadas assenta no uso de patamares intermédios e lanços de escadas alternados, tal como se apresenta na Figura 96.

Para implementar este tipo de escadas, vamos começar por modelá-las de acordo com a Figura 97. Nessa figura podemos ver vários lanços de escadas, interligados entre si por patamares representados por rectângulos



Figura 96: Escada com patamares intermédios. Fotografia de Stijn Nieuwendijk.

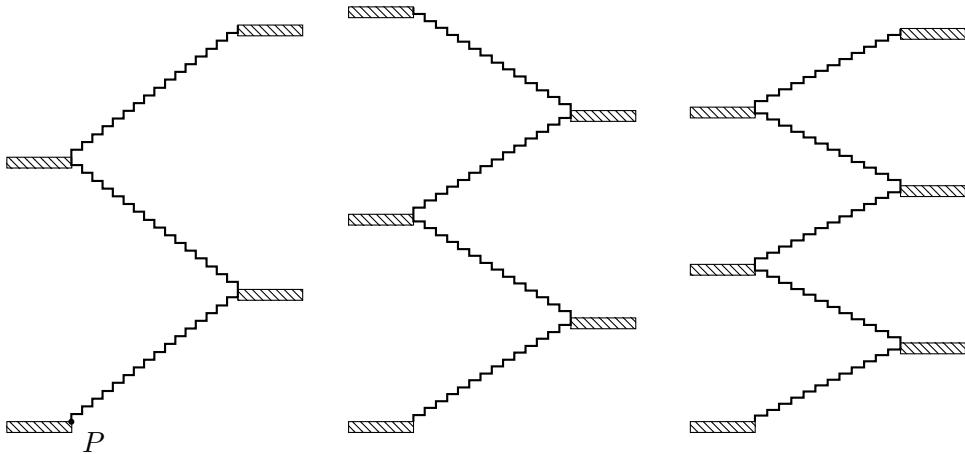


Figura 97: Um modelo de escadas com lanços alternados. Da esquerda para a direita, as escadas possuem três, quatro e cinco lanços. O ponto P representa o ínicio da primeira escada.

a tracejado.

Do ponto de vista da modelação destas escadas, os patamares são totalmente desnecessários pois, admitindo que os patamares têm todos a mesma dimensão, para uma dada dimensão horizontal de escada que inclua patamares é trivial derivar a dimensão que não inclui os patamares. Por este motivo, vamos considerar que o ponto de partida de cada escada não inclui o patamar e coincide com o arranque da escada, tal como se vê representado pelo ponto P na escada mais à esquerda.

Para desenarmos este tipo de escadas não temos mais do que definir uma função idêntica à função `lancos-escada` mas que, de cada vez que inicia um novo lance de escada, usa o simétrico do cobertor para efectuar a “mudança de direcção.”

```
(defun lancos-alternados-escada (p c e n lancos)
  (if (= lancos 1)
      (lanco-escada p c e n)
      (lancos-alternados-escada
       (lanco-escada p c e n)
       (- c) e n (- lancos 1))))
```

Para dimensionarmos estas escadas temos, antes de mais, de saber quantos lanços serão necessários para vencer uma dada altura. Para isso, é importante percebermos que o número de lanços depende (entre outros factores) de o início da escada ficar, ou não, do mesmo lado que o seu fim. Como é fácil percebermos pela análise da Figura 97, se o início da escada está do mesmo lado que o fim, a escada terá necessariamente de ter um número par de lanços. Se o início da escada está no lado oposto ao do fim, então

a escada terá necessariamente um número ímpar de lanços. Este facto implica que, para um dado início e fim de escada, à medida que a altura da escada aumenta, o número de lanços apropriado para a vencer terá sempre de aumentar dois lanços de cada vez.

Deste modo, podemos idealizar um algoritmo para o dimensionamento destas escadas que, para além das medidas do espaço onde temos de inserir a escada, irá também receber um número inicial de lanços pretendidos. Se esse número for par, o início e o fim da escada estão no mesmo lado, caso contrário, estão em lados opostos. Este algoritmo irá determinar as dimensões do cobertor e espelho de cada lanço individual e, caso eles violem as restrições ergonómicas que já discutimos anteriormente, aumenta o número de lanços em duas unidades e tenta de novo. Quando o número de lanços se tornar tão grande que o tamanho do espelho e correspondente fica inferior à altura mínima admissível e_{\min} , o algoritmo desiste, devolvendo `nil` para assinalar que não é possível satisfazer os constrangimentos. Nessa altura, o utilizador deverá relaxar algumas das restrições como, por exemplo, os limites da regra de Blondel.

Atendendo a este algoritmo, a seguinte função é uma adaptação da função `comprimento-cobortores` mas que, no caso de violação de restrições, se limita a aumentar o número de lanços:

```
(defun altura-cobortores-lancos
  (lancos ht emin emax l cmin bmin bmax /
    h nmin nmax ncmax nbmin nbmax nfinalmax nfinalmin)
  (setq h (/ ht lancos)
        nmin (ceiling (- (/ h emax) 1))
        nmax (fix (- (/ h emin) 1))
        ncmax (fix (/ l cmin)))
  (if (< h emin)
      nil
      (if (< ncmax nmin)
          (altura-cobortores (+ lancos 2)
                             ht emin emax l cmin bmin bmax)
          (progn
            (setq nmax (min nmax ncmax)
                  nbmin (cobortores-blondel h l bmax)
                  nbmax (cobortores-blondel h l bmin))
            (if (> (ceiling nbmin) (fix nbmax))
                (altura-cobortores (+ lancos 2)
                                   ht emin emax l cmin bmin bmax)
                (progn
                  (setq nfinalmin (max nmin (ceiling nbmin))
                        nfinalmax (min nmax (fix nbmax)))
                  (if (> nfinalmin nfinalmax)
                      (altura-cobortores (+ lancos 2)
                                         ht emin emax l cmin bmin bmax)
                      (list h (round (/ (+ nfinalmin nfinalmax) 2)) lancos)))))))
      ))))
```

Para facilitar a utilização desta função podemos definir outra que, dados dois pontos que delimitam um rectângulo, dado o número inicial de

lanços e dados os limites ergonómicos dos degraus, dimensiona a escada que cabe naquele rectângulo e desenha-a a começar no primeiro ponto.

```
(defun dimensiona-escada-limitada
    (p0 p1 nlancos emin emax cmin bmin bmax /
     l h hlanco dims llanco clanco c e)
  (setq l (- (cx p1) (cx p0))
        h (- (cy p1) (cy p0))
        dims (altura-cobertores-lancos
               nlancos h emin emax l cmin bmin bmax))
  (if (null dims)
      (alerta "Não é possível dimensionar a escada")
      (progn
        (setq hlanco (car dims)
              clanco (cadr dims)
              nlancos (caddr dims)
              c (/ l clanco)
              e (/ hlanco (+ clanco 1)))
        (lancos-alternados-escada
         p0 c e clanco nlancos))))
```

Finalmente, para podermos invocar esta função a partir do AutoCad vamos definir um comando que, entre outros dados, pede ao utilizador que delimita um rectângulo onde deverá ser inserida a escada e que invoca a função anterior.

```
(defun c:dimensiona-escada-limitada (/ p0 p1 nlancos emin emax cmin bmin bmax)
  (setq p0 (getpoint "\nPonto inicial da escada?")
        p1 (getcorner p0 "\nPonto final da escada?")
        nlancos (getint "Número mínimo de lanços?")
        emin (getreal "\nAltura mínima do degrau?")
        emax (getreal "\nAltura máxima do degrau?")
        cmin (getreal "\nComprimento mínimo do degrau?")
        bmin (getreal "\nDimensão mínima de Blondel?")
        bmax (getreal "\nDimensão máxima de Blondel?"))
  (dimensiona-escada-limitada p0 p1 nlancos emin emax cmin bmin bmax))
```

Para ilustrar a utilização deste comando, apresentamos na Figura 98 uma imagem onde dimensionámos várias escadas de modo a ficarem contidas dentro dos rectângulos apresentados e onde, após a execução do comando, acrescentámos os patamares.

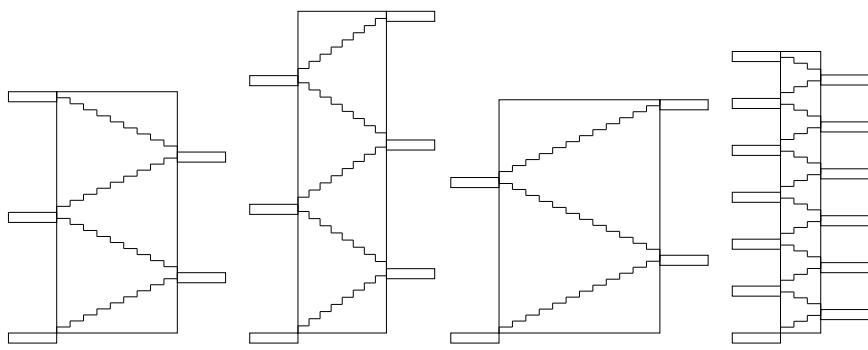


Figura 98: Lanços de escada criados de modo a ocuparem as áreas delimitadas pelos rectângulos. Da esquerda para a direita, as escadas são dimensionadas para um número mínimo de dois, três, três e dois lanços, respectivamente, mas as suas dimensões finais implicam quatro, cinco, três e doze lanços, respectivamente. As escadas possuem um cobertor mínimo de 0.23m, espelho entre 0.14m e 0.18m e limites de Blondel entre 0.62m e 0.64m.