

Conceitos de Programação em *Kotlin*

Pedro Pereira

28 de Fevereiro de 2023

Conteúdo

1	Generalidades	5
1.1	Hardware	5
1.2	Software	6
1.3	Sistema de ficheiros	7
1.4	Linha de comandos	7
1.4.1	Variável de ambiente <i>Path</i>	9
1.5	Execução de Programas	9
1.6	Ambiente de desenvolvimento	10
1.6.1	Criação da pasta de trabalho	11
1.6.2	Editor de texto	11
1.6.3	<i>JDK</i>	11
1.6.4	Ferramentas de <i>Kotlin</i> para a <i>JVM</i>	12
1.6.5	Primeiro programa em <i>Kotlin</i>	12
1.6.6	Utilização do <i>REPL</i>	13
1.7	Diagramas sintáticos	13
1.8	Diagramas de atividade	15
2	Tipos, valores e expressões	17
2.1	Números inteiros	17
2.1.1	Tipos <i>Int</i> , <i>Long</i> , <i>Byte</i> e <i>Short</i>	19
2.2	Declaração de valores	21
2.2.1	Inteiros positivos	22
2.3	Números reais	23
2.3.1	Tipos <i>Float</i> e <i>Double</i>	24
2.4	Texto	26
2.4.1	Tipo <i>Char</i>	26
2.4.2	Tipo <i>String</i>	28
2.5	Lógica	31
2.5.1	Tipo <i>Boolean</i>	31
2.6	Intervalos	33
2.7	Sintaxe das expressões	34
3	Programas, leituras e escritas	37
3.1	A função <i>main</i>	37
3.2	Erros de compilação e erros de execução	38
3.3	Escrita na consola	39
3.4	Utilização de variáveis	40
3.4.1	Afetação de variáveis	40
3.5	Leitura da consola	42
3.5.1	Leitura de números	42
3.5.2	Leitura de símbolos	43
3.6	Leitura e escrita standard	43
3.6.1	Redirecionamento do input	44
3.6.2	Redirecionamento do output	44
3.7	Instruções e expressões	45

4	Decisões	47
4.1	Decisão binária - <i>if</i>	47
4.1.1	Parte <i>else</i> da decisão binária	48
4.1.2	Expressão <i>if</i>	50
4.2	Decisão múltipla - <i>when</i>	51
4.3	Ambiente integrado <i>IntelliJ</i>	54
4.3.1	Utilização do <i>debugger</i>	56
5	Ciclos	59
5.1	Repetição com condição final - <i>do-while</i>	59
5.2	Repetição com condição inicial - <i>while</i>	60
5.3	Incrementar e decrementar	62
5.4	Repetição por iteração de sequência - <i>for</i>	63
5.5	Quebra da repetição - <i>break</i>	64
5.6	Valores e variáveis locais	65
6	Funções	67
6.1	Funções sem parâmetros	67
6.1.1	Valores e variáveis globais	68
6.2	Funções com parâmetros	68
6.2.1	Argumentos posicionais ou nomeados	69
6.2.2	Parâmetros não são variáveis	70
6.3	Assinatura e sobrecarga	70
6.3.1	Argumentos por omissão	71
6.4	Retorno das funções	71
6.4.1	Funções puras	71
6.4.2	Expressão como corpo da função	72
6.4.3	Retorno do tipo <i>Unit</i>	73
6.5	Funções como valores	74
6.5.1	Expressões <i>lambda</i>	75
6.5.2	Função <i>repeat</i>	76
6.6	Múltiplos ficheiros fonte	76
7	Definição de tipos	79
7.1	Tipos agregados	79
7.1.1	Função <i>toString()</i>	80
7.1.2	Comparação de objetos	80
7.2	Funções extensão	82
7.2.1	Função extensão <i>copy</i>	82
7.2.2	Sobrecarga de operadores	82
7.3	Enumerados	83
7.3.1	Enumerados com propriedades	85
7.4	Tipos anuláveis	86
7.4.1	Verificar condições com <i>null</i>	87
8	Listas	89
8.1	Operadores e propriedades das listas	89
8.2	Operações sobre listas	91
8.3	Filtragens	92
8.4	Transformações	93
8.5	Iterações	94
8.5.1	Iteração com índice	95
8.6	Listas mutáveis	95
8.7	Criação de listas iniciadas	96
8.8	Arrays	97
8.9	Reconhecimento e conversão de numeração romana	99

Capítulo 1

Generalidades

Este capítulo apresenta a organização básica do computador na perspectiva de quem o utiliza para programar usando a linguagem *Kotlin*.

Para programar não é necessário conhecer em pormenor o que é o computador, mas o programador deve saber quais são os constituintes principais do computador no nível de abstração oferecido pela linguagem e quais as ferramentas utilizadas para programar.

1.1 Hardware

O computador é fisicamente constituído por dispositivos e componentes eletrónicos que no seu conjunto se designa hardware.

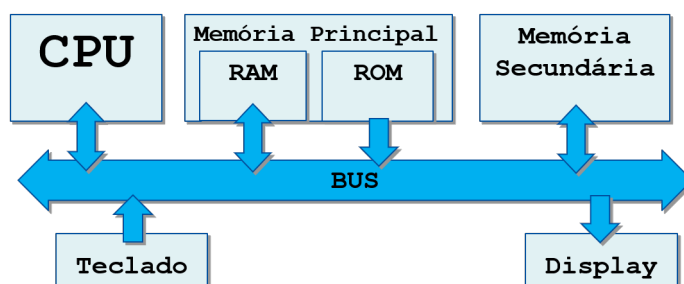


Figura 1.1: Arquitetura computacional

Os dispositivos externos são normalmente utilizados para entrada e saída de dados (*IO - Input Output*) fazendo a comunicação com o utilizador, podendo alguns serem conectados e desconectados do computador. Por exemplo:

- Teclado para introdução de dados;
- Display para apresentação de dados;
- Rato e *Touchpad* para auxiliar a introdução de dados;
- Impressora para apresentação de dados;

Os componentes eletrónicos internos fazem parte da placa principal (*mother board*) do computador ou de outras placas adicionais. Os principais componentes internos são:

- Processador ou Unidade Central de Processamento (*CPU - Central Processing Unit*);
- Memória principal só para leituras (*ROM - Read Only Memory*);
- Memória principal para escrita e leitura (*RAM - Random Access Memory*);
- Memória secundária, normalmente disponível em discos rígidos (*HDD - Hard Disk Drive*) ou em unidades de memória de estado sólido (*SSD - Solid-State Drive*);

Tal como apresenta a figura 1.1, estes dispositivos e componentes estão ligados através de barramentos (*BUS*) que permitem a transferência de informação. Estes barramentos têm linhas ou ligações onde passam os dados a transferir, os endereços dos componentes e os sinais de controlo.

O *CPU* executa sequencialmente as instruções que vai lendo da memória. A execução destas instruções realizam leituras de dados dos dispositivos de entrada ou das memórias e realizam escritas nos dispositivos de saída ou nas memórias.

São características importantes do hardware:

- A dimensão (em polegadas) e a resolução (em pixels na vertical e na horizontal) do display;
- A velocidade de execução (em GHz - Giga Hertz) e a dimensão dos dados a operar (em número de bits) do *CPU*;
- A dimensão (em GB - Giga Bytes) da *RAM*;
- A dimensão (em GB ou TB - Giga ou Tera Bytes) do *HDD* e/ou *SSD*;

Por exemplo, um computador poderá ter um display de 17 polegadas com resolução de 1920 x 1080 pixels, um *CPU* de 64Bits que funciona a 3.7GHz, 16GB de *RAM* e 512GB de *SSD* + 1TB de *HDD*.

1.2 Software

É designado por software o conjunto de dados e instruções que dão suporte lógico ao funcionamento do computador. É o software que indica ao hardware o que deve fazer.

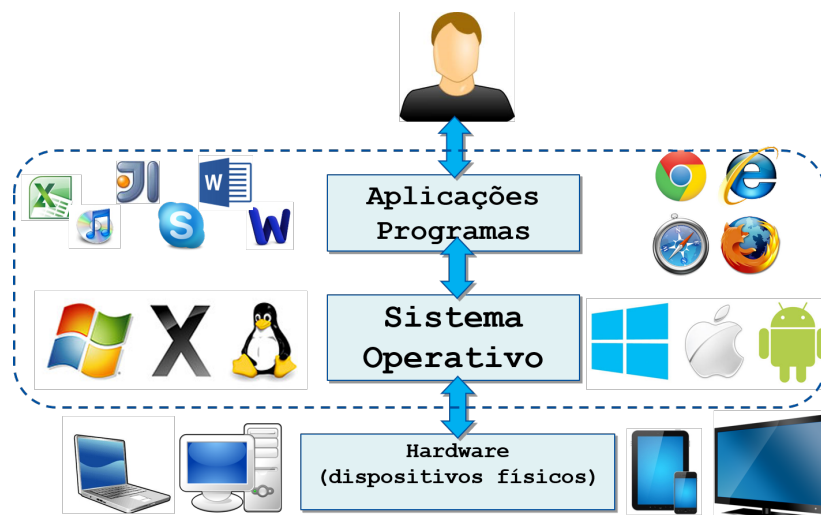


Figura 1.2: Interação com o computador

O **sistema operativo** (*SO*) é a parte do software que interage diretamente com o hardware e fornece uma interface de utilização aos programas, ou aplicações. O *Windows*, o *Linux* e o *macOS*, são dos sistemas operativos mais usados em computadores fixos e portáteis. O *Android* e o *iOS* são dos mais usados em dispositivos móveis.

Os **programas** e as **aplicações** são a parte do software visível para o utilizador que usam a interface disponibilizada pelo *SO*. Os programas podem ser instalados e desinstalados, mas alguns são instalados simultaneamente com o *SO*. O navegador na Internet (*browser*), o editor de documentos e o gestor de ficheiros e de pastas, são exemplos de programas que são instalados nos computadores.

A figura 1.2 mostra a interação do utilizador com o computador identificando as duas camadas de software. Usando esta divisão por camadas, os programas ficam apenas comprometidos com determinado *SO* e são independentes do hardware. Podem existir programas que usam diretamente o hardware, mas isso impede que funcionem noutros computadores com hardware diferente.

No texto deste livro, a grande maioria da informação é independente do sistema operativo, mas quando for necessário dar exemplos concretos da utilização de um deles, será usado o *Windows* e o leitor que use outro terá que consultar a documentação do seu sistema operativo para fazer o equivalente.

1.3 Sistema de ficheiros

Os sistemas operativos organizam a informação na memória secundária (em *HDD*, *SSD*, etc.) usando um sistema de ficheiros. O utilizador usa a aplicação “explorador de ficheiros” ou “gestor de arquivos” para usar este sistema.

Os sistemas de ficheiros estruturam a informação usando os seguintes elementos:

- **Ficheiro**, ou arquivo, para armazenar um bloco de informação;
- **Pasta**, ou diretório, para conter ficheiros e outras pastas;
- **Unidade**, ou drive, para representar cada dispositivo ou partição de memória secundária;

No *Windows*, cada unidade é identificada por uma letra (por exemplo: *C* ou *D*) que normalmente é apresentada seguida do símbolo *:*, por exemplo *C:*, e cada unidade tem a sua pasta raiz. No *Linux* e no *macOS* as unidades ficam “montadas” em pastas de um único sistema de ficheiros.

Cada ficheiro e cada pasta tem um nome que deve ter algum significado para o utilizador. Normalmente, o nome do ficheiro termina com uma sequência de letras depois de um ponto final, designada por extensão, que indica o tipo de informação armazenada. Por exemplo, no nome *res.doc*, a extensão *.doc* significa que o conteúdo do ficheiro é um documento.

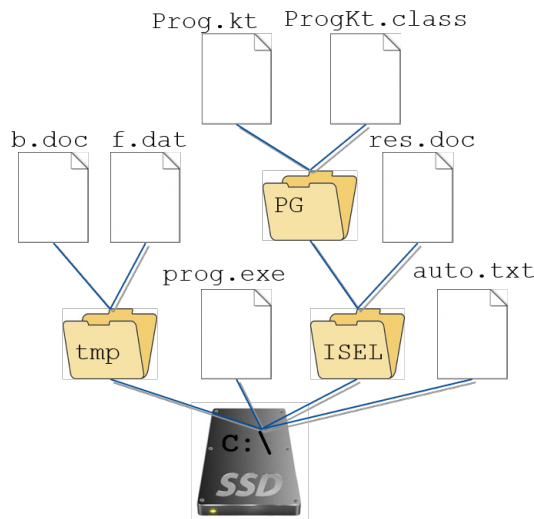


Figura 1.3: Unidade com pastas e ficheiros

A figura 1.3 apresenta os ficheiros da unidade *C* organizados em pastas. O ficheiro *Prog.kt* está na pasta *PG*, que por sua vez está na pasta *ISEL* da unidade *C*. O caminho completo que identifica este ficheiro será: *C:\ISEL\PG\Prog.kt*¹.

Dependendo do *SO*, as extensões dos nomes dos ficheiros podem ter significados diferentes. Por exemplo, a extensão *.exe* é usada no *Windows* para ficheiros com código que é executável. A extensão *.txt* é usada em ficheiros que podem ser editados com qualquer editor de texto. A extensão *.kt* também é usada para ficheiros de texto, mas neste caso, o texto descreve código escrito em linguagem *Kotlin*.

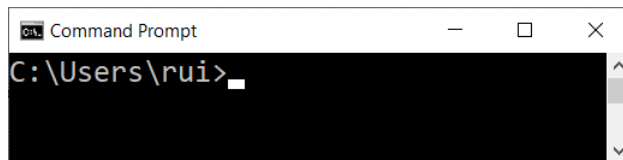
1.4 Linha de comandos

Qualquer sistema operativo tem uma forma, normalmente designada por linha de comandos, para executar comandos introduzidos textualmente pelo utilizador. Para tal, deve ser usada a aplicação *Console*, *Terminal* ou *Command Prompt* do sistema.

A linha de comandos permite executar comandos do *SO*, mas também permite colocar em execução programas, como se fossem comandos.

Por exemplo, no *Windows*, quando é utilizado o programa *Command Prompt* é aberta uma janela onde aparece o *prompt*, que indica a pasta corrente, ficando o cursor logo a seguir, tal como mostra a figura 1.4. Tipicamente, a pasta inicial é a pasta do utilizador, que neste exemplo é *C:\Users\rui*.

¹Dependendo do sistema operativo, o símbolo separador dos nomes poderá ser ** ou */*

Figura 1.4: Janela da linha de comandos no *Windows*

Se o utilizador escrever a palavra **dir** seguida da tecla `[↵]`, é executado o comando **dir** (*directory*) do sistema, que apresenta as entradas (ficheiros e pastas) contidos na pasta corrente.

Alguns dos comandos de sistema mais usados são:

- **dir** (**ls** no *Linux* ou *macOS*) - Apresenta o conteúdo de uma pasta, a corrente ou a que for indicada como argumento;
- **cd** (*change directory*) - Muda a pasta corrente para a que for indicada como argumento;
- **md** (*make directory*) (**mkdir** no *Linux* ou *macOS*) - Cria uma nova pasta com o nome indicado como argumento;
- **rd** (*remove directory*) (**rmdir** no *Linux* ou *macOS*) - Remove a pasta indicada como argumento, caso esteja vazia;
- **del** (*delete*) (**rm** no *Linux* ou *macOS*) - Remove o ficheiro indicado como argumento;

Todos os comandos indicados recebem como argumento o nome de uma entrada (pasta ou ficheiro) que poderá ser o caminho relativo à pasta corrente ou completo (se começar por \), podendo usar `..` como pasta mãe. Por exemplo, o comando **md Nova** cria a pasta com o nome *Nova* na pasta corrente, mas o comando **md \ISEL\Nova** cria a pasta com o nome *Nova* dentro da pasta *ISEL* que está na raiz da unidade. O comando **cd ISEL\PG** muda para a pasta *PG* contida na pasta *ISEL* contida na pasta corrente, mas o comando **cd ..\PG** muda para a pasta *PG* contida na pasta mãe da pasta corrente.

Se o utilizador escrever a palavra *prog*² seguida da tecla `[↵]`, como **prog** não é um comando, o sistema procura nas pastas indicadas na variável de ambiente **Path** se existe um ficheiro com o nome **prog.exe** e, caso encontre, coloca-o em execução, caso contrário dá uma mensagem de erro semelhante à apresentada neste exemplo.

```
C:\Users\rui>prog
'prog' is not recognized as an internal or external command, operable
program or batch file.
```

Admitindo que o sistema começa a procura na pasta corrente, para executar o programa do ficheiro **prog.exe** apresentado no sistema de ficheiros da figura 1.3, na linha de comandos poderiam ser executados os comandos:

```
C:\Users\rui>cd \
C:\>prog
... resultado da execução de prog ...
```

que primeiro muda a pasta corrente para a raiz da unidade e coloca em execução **prog.exe**, que desta forma se encontra na pasta corrente. Sem mudar a pasta corrente pode-se indicar o caminho completo do ficheiro executável:

```
C:\Users\rui>C:\prog
... resultado da execução de prog ...
```

Para executar o programa a partir de qualquer pasta corrente sem ter que indicar o caminho completo do executável, é necessário adicionar a pasta **C:**, onde está o ficheiro **prog.exe**, à variável **Path** do sistema. Depois disso, qualquer que seja a pasta corrente pode executar:

```
C:\Users\rui>prog
... resultado da execução de prog ...
```

²Nos exemplos de utilização o texto introduzido pelo utilizador é sempre apresentado em itálico.

1.4.1 Variável de ambiente *Path*

Os sistemas operativos têm um conjunto de variáveis de ambiente com vários objetivos, uma delas é a variável *Path* que tem a lista das pastas onde o sistema procura por ficheiros executáveis para poderem ser usados como se fossem comandos do sistema. No *Windows*, para consultar esta variável execute o comando:

```
C:\Users\rui>echo %Path%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0
```

O comando *echo* apresenta o texto que for indicado como argumento, que poderá conter nomes de variáveis entre os símbolos %. O conteúdo da variável *Path* é uma lista de nomes completos das pastas, separados pelo símbolo ;, onde o sistema procura por ficheiros executáveis. No *Linux* e *macOS*, o comando seria *echo \$PATH* e os nomes estão separados pelo símbolo :.

O comando *set*, do *Windows*, permite alterar variáveis de ambiente. Assim, para acrescentar mais uma pasta à lista de pastas da variável *Path*, por exemplo a pasta *C:*, e depois executar *prog*, é necessário executar os comandos:

```
C:\Users\rui>set Path=%Path%;C:\
C:\Users\rui>prog
... resultado da execução de prog ...
```

A partir deste momento, enquanto usar esta janela de comandos, o sistema também procurará por executáveis na pasta *C:*. Mas esta alteração da variável *Path* só é válida para a janela de comandos atual, ou seja, quando for aberta uma nova janela de comandos, a variável *Path* não tem esta alteração.

Para alterar definitivamente uma variável de ambiente no *Windows*, deve utilizar um programa fornecido pelo próprio sistema operativo que permite a edição das variáveis. Para abrir este programa, use a pesquisa do *Windows*, procure por “var”, ou seja, clique no botão *Start* do *Windows* (que tem o símbolo de uma janela) e escreva “var”, depois escolha a opção que permite editar as variáveis de ambiente (*environment variables*), escolha a edição da variável *Path* e acrescente mais uma pasta no final.

Depois desta alteração, as novas janelas de comandos usarão estas variáveis de ambiente.

1.5 Execução de Programas

Cada programa tem um conjunto de instruções que são executadas para realizar operações. As operações processam dados e produzem resultados. Normalmente, os dados são lidos de uma ou mais entradas (*input*) que pode ser o teclado, o conteúdo de ficheiros, etc. Os resultados são escritos numa ou mais saídas (*output*) que pode ser o *display*, o conteúdo de ficheiros, etc. A figura 1.5 é uma representação do conceito de programa.

Existem muitos ambientes de execução de programas. Quando desenvolvemos um programa é necessário decidir em que ambiente pretendemos que ele se execute. Podemos executar programas em:

- **Sistemas dedicados** - Execução direta no hardware de sistemas de dimensão reduzida (*Raspberry pi*, *Arduino*, etc.);
- **Sistemas operativos** - Usando a interface fornecida pelo *SO* previamente instalado num computador (*Windows*, *Linux*, *macOS*, *Android*, *iOS*, etc.);
- **Máquinas virtuais** (*VM* - *Virtual Machine*) - Ambientes suportados no *SO* ou em programas previamente instalados. A Máquina virtual Java (*JVM* - *Java Virtual Machine*) e o *.Net*, são as mais utilizadas;

Um programa desenvolvido para funcionar numa *VM* é mais portátil do que para funcionar num *SO*, porque desta forma ele pode ser executado em qualquer computador com qualquer *SO* que suporte a *VM* escolhida.

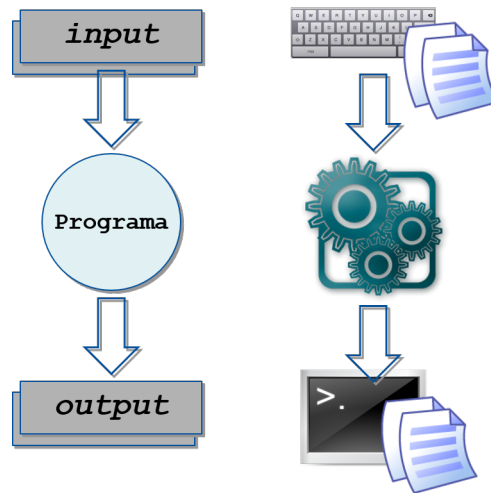


Figura 1.5: Conceito de Programa

Atualmente, a *JVM* é suportada nos sistemas operativos mais utilizados. Por este motivo, vamos fazer programas para este ambiente de execução, permitindo que seja utilizado qualquer computador com *Windows*, *Linux* ou *macOS*, que tenha a *JVM*.

Para colocar em execução um programa desenvolvido para a *JVM*, em que o código fica em ficheiros com extensão `.class`, é necessário executar a máquina virtual (disponível no ficheiro executável `java.exe` numa pasta que conste na variável `Path` do sistema) seguido do nome do programa a executar. Por exemplo, no *Windows*, para executar o programa do ficheiro `ProgKt.class` apresentado no sistema de ficheiros da figura 1.3, na janela de comandos com o diretório corrente em `C:\ISEL\PG`, basta executar o comando:

```
C:\ISEL\PG>java ProgKt
... resultado da execução de ProgKt ...
```

Desta forma, estamos a executar diretamente no sistema operativo a *JVM* disponível no programa contido no ficheiro `java.exe`, ao qual estamos a dar o argumento `ProgKt`. Por sua vez, a *JVM* procura um ficheiro como o nome indicado no argumento, mas com a extensão `.class` e põe em execução esse programa, que foi desenvolvido para a *JVM*.

1.6 Ambiente de desenvolvimento

Existem diversas ferramentas para desenvolvimento de programas em *Kotlin*. Poderíamos usar já um ambiente integrado em que todos os passos do desenvolvimento acontecem de forma automática, mas tal não é aconselhável no percurso pedagógico pretendido, em que queremos controlar todos os passos do desenvolvimento. Assim, vamos primeiro usar um ambiente em que utilizamos um editor de texto qualquer, executando tudo na janela de comandos do sistema operativo, que poderá ser *Windows*, *Linux* ou o *macOS*. Mais adiante, no desenvolvimento de programas de maior dimensão iremos usar um ambiente integrado.

Outra escolha importante é sobre o ambiente de execução onde os nossos programas se executam. Atualmente, existem compiladores de *Kotlin* que geram código para diversos ambientes de execução: *JavaScript*, *Windows* (*x64* e *X86*), *Linux* (*x64*, *arm32*, *arm64*, etc.), *macOs*, *Android*, *WebAssembly* e *JVM*. Pelas razões já explicadas anteriormente, vamos executar os nossos programas na *JVM*.

Para instalar no computador as ferramentas básicas para desenvolvimento de programas em *Kotlin* para a *JVM*, são necessários três componentes:

- **Editor de texto**, especializado para código;
- **Ferramentas da máquina virtual Java** (*JDK* - *Java Development Kit*);
- **Ferramentas da linguagem *Kotlin*** para a *JVM*;

1.6.1 Criação da pasta de trabalho

Antes de instalar estes três componentes é aconselhável criar uma pasta específica, que muito provavelmente terá mais sub-pastas, que irão servir para armazenar os programas a desenvolver. Por exemplo, crie uma pasta ISEL na raiz da unidade C e depois uma pasta PG dentro da pasta ISEL. Para tal, abrindo a janela de comandos do *Windows*, posicionada numa pasta qualquer, executamos os seguintes comandos:

```
C:\>>>>cd \
C:\>>>>md ISEL\PG
C:\>>>>cd ISEL\PG
C:\ISEL\PG>dir
Directory of C:\ISEL\PG
    07:58 PM <DIR> .
    07:58 PM <DIR> ..
    0 File(s) 0 bytes
    2 Dir(s) ??? bytes free
```

Listagem 1.1: Criar a pasta de trabalho.

As pastas `.` e `..` são pastas virtuais que representam a pasta corrente e a pasta mãe, respetivamente. Assim, o comando `cd ..` muda para a pasta mãe, independentemente do seu nome.

1.6.2 Editor de texto

Qualquer sistema operativo integra um editor de texto, mas não é especializado para a escrita de código de programas. Um editor especializado para código identifica as palavras reservadas da linguagem, facilita a indentação, etc.

Dependendo do sistema operativo instalado recomenda-se os seguintes editores, mas podem ser usados outros:

- **Notepad++** para o *Windows* disponível em www.notepad-plus-plus.org/ com a extensão para *Kotlin* em www.kotlinresources.com/library/nppextension
- **Coda** para *macOs* disponível em www.panic.com/coda/
- **Sublime** para *Linux*, *macOs* ou *Windows* disponível em www.sublimetext.com/ instalando o *package* *Kotlin*.

Caso tenha instalado o *Notepad++* ou o *Sublime* no *Windows*, deve ter sido adicionada uma pasta com o nome *Notepad++* ou *Sublime Text* dentro da pasta *C:\Program Files* ou dentro da pasta *C:\Programas*, caso o *Windows* esteja configurado para português. O ficheiro com extensão *.exe*, com nome sugestivo, é o programa desse editor, por exemplo *notepad++.exe*.

Caso a pasta corrente seja *C:\Program Files\Notepad++*, para executar o editor criando o ficheiro *abc.txt*, basta executar:

```
C:\Program Files\Notepad++>Notepad++ abc.txt
```

mas o ficheiro *abc.txt* fica na pasta corrente, ou seja, o caminho completo do ficheiro de texto será *C:\Program Files\Notepad++\abc.txt*.

Para poder executar o editor, na linha de comando, a partir de qualquer pasta corrente, é necessário acrescentar ao *Path* do sistema a pasta onde ficou o ficheiro executável do editor, que neste caso seria *C:\Program Files\Notepad++*, tal como foi descrito em 1.4.1.

Assim, para criar um ficheiro usando o editor na pasta de trabalho, será:

```
C:\ISEL\PG>notepad++ Prog.kt
C:\ISEL\PG>_
```

Com este comando será aberta uma nova janela para o editor e a janela de comandos permanece. Se já existir um ficheiro com esse nome, o editor aproveita o conteúdo atual. Caso não exista, o editor cria um ficheiro novo com esse nome.

1.6.3 JDK

Para executar programas na *JVM* seria apenas necessário instalar o conjunto de utilitários que dão suporte à máquina virtual Java (*JRE - Java Runtime Environment*). Mas, como o ambiente integrado que será usado mais adiante precisa também de alguns utilitários para o desenvolvimento em

Java, é preferível instalar já o (*JDK - Java Development Kit*) que também inclui o *JRE*.

O *JDK*, para qualquer ambiente, está disponível em: www.oracle.com/technetwork/java/javase/downloads/index.html.

Siga as instruções, e instale a versão mais recente para o seu sistema operativo.

Para poder executar os comandos do *JDK* na linha de comando a partir de qualquer pasta corrente, é necessário acrescentar ao *Path* do sistema a pasta onde ficaram os ficheiros executáveis, que neste caso seria `C:\Program Files\Java\jdk-XX\bin3`, tal como foi descrito em 1.4.1.

1.6.4 Ferramentas de *Kotlin* para a *JVM*

Para instalar as ferramentas para programar em *Kotlin* para a *JVM* faça download do ficheiro `kotlin-compiler-1.8.10.zip` disponível no final da página www.github.com/JetBrains/kotlin/releases/tag/v1.8.10 que terá o compilador, ou consulte em www.kotlinlang.org/docs/tutorials/command-line.html uma versão mais atual.

Abra o ficheiro `.zip`, utilizando um programa qualquer que permita abrir ficheiros deste tipo, que no *Windows* é o próprio “Explorador de ficheiros”. Copie a pasta contida, que deverá chamar-se `kotlinc`, para a pasta `C:\ISEL`. Ficando assim com a pasta `C:\ISEL\kotlinc`.

Finalmente, acrescente ao *Path* do sistema a pasta onde ficaram os ficheiros executáveis, que deve ser `C:\ISEL\kotlinc\bin`, tal como foi descrito em 1.4.1.

Para verificar que ficou tudo correto, vamos fazer o primeiro programa em *Kotlin*.

1.6.5 Primeiro programa em *Kotlin*

Abra uma nova janela de comandos e coloque em execução o editor, para editar o ficheiro `Hello.kt`:

```
C:\ISEL\PG>notepad++ Hello.kt
C:\ISEL\PG>_
```

Usando o editor escreva o seguinte programa e grave o ficheiro.

```
fun main() {
    println("Hello World!")
}
```

Listagem 1.2: Hello World! em Kotlin

Depois de gravar o ficheiro no editor, continue a utilizar a linha de comandos para verificar que o ficheiro ficou gravado.

```
C:\ISEL\PG>dir
Directory of C:\ISEL\PG
07:58 PM <DIR> .
07:58 PM <DIR> ..
01:40 AM      42 Hello.kt
1 File(s) 42 bytes
2 Dir(s) ??? bytes free
```

Como se constata, o editor gravou o ficheiro `Hello.kt` com 42 bytes.

Agora, compile o programa, executando o compilador de *Kotlin* e verifique qual foi o ficheiro gerado.

```
C:\ISEL\PG>kotlinc Hello.kt
C:\ISEL\PG>dir
Directory of C:\ISEL\PG
07:58 PM <DIR> .
07:58 PM <DIR> ..
01:40 AM      42 Hello.kt
01:42 AM     682 HelloKt.class
1 File(s) 724 bytes
2 Dir(s) ??? bytes free
```

³XX é o número da versão, por exemplo 17.0.2

O compilador criou o ficheiro `HelloKt.class` com 682 bytes.

Para executar o programa, pode-se usar a *JVM* diretamente com o comando `java -cp . HelloKt`, que pode necessitar da opção `-cp .` para indicar à *JVM* que deve procurar na pasta corrente o ficheiro `.class`, ou indiretamente com o comando `kotlin HelloKt`, que se encarrega de passar à *JVM* todas as opções necessárias.


```
C:\ISEL\PG>java -cp . HelloKt
Hello World!
C:\ISEL\PG>kotlin HelloKt
Hello World!
```

1.6.6 Utilização do *REPL*

Se o compilador `kotlinc` for executado sem passar qualquer argumento (ficheiro a compilar) é executado um programa interativo que permite “experimental” instruções em *Kotlin*. Este programa realiza um ciclo de leituras, avaliações e escritas do resultado de expressões (*REPL* - *Read Eval Print Loop*).

O *REPL* pode ser usado da seguinte forma:

```
C:\ISEL\PG>kotlinc
Welcome to Kotlin version ...
Type :help for help, :quit for quit
>>> println("Hello World!") /*Chamada a função */
Hello World!
>>> 2 + 5 /*Expressão*/
res1: Int = 7
>>> 2 < 5 // Comparação de valores
res2: Boolean = true
>>> :quit
C:\ISEL\PG>_
```

O utilizador escreve uma instrução ou uma expressão em *Kotlin* ou um comando do *REPL* imediatamente a seguir a `>>>` terminando com  e na linha seguinte é apresentado o resultado.

Qualquer introdução de texto iniciado com o símbolo `:` é interpretado como um comando. Texto não iniciado por `:` é uma instrução ou uma expressão em *Kotlin*.

O comando `:quit` termina a execução do *REPL* e volta à linha de comandos do sistema operativo.

Nesta utilização do *REPL* foi introduzida a instrução `println("Hello World!")` que quando executada escreve o texto entre aspas, foi introduzida a expressão `2+5` que quando avaliada produz o valor 7 do tipo `Int` e foi introduzida a expressão `2<5` que quando avaliada verifica se 2 é menor que 5, produzindo o valor `true` do tipo `Boolean`.

Em *Kotlin*, o texto escrito entre `/* e */` e o texto depois de `//` até ao final da linha são considerados comentários, sendo ignorados pelo compilador.

Nas utilizações do *REPL*, nos restantes capítulos, não será apresentado em itálico o texto após `>>>`, apesar de ser texto introduzido pelo utilizador.

No capítulo seguinte "Tipos, Valores e Expressões" ainda não serão produzidos programas em *Kotlin* e será usado sempre o *REPL* para avaliar expressões.

1.7 Diagramas sintáticos

Nos capítulos seguintes são usados diagramas sintáticos para descrever as regras sintáticas da linguagem *Kotlin*.

Cada diagrama descreve uma regra, com um determinado nome, tendo um ponto de entrada e um ponto de saída. Cada diagrama tem ligações direcionais entre símbolos terminais (figuras sem arestas) e outras regras (retângulos com o nome da regra). O texto válido segundo uma regra percorre um caminho possível desde a entrada até à saída do diagrama.

Por exemplo, os diagramas da figura 1.6 descrevem a regra *frase*. O primeiro diagrama indica que *frase* começa com uma letra maiúscula (de A a Z) seguida de uma *palavra* (outra regra) ou nenhuma *palavra*, termina com um ponto final, mas antes do ponto pode ter várias vezes *palavra* com uma

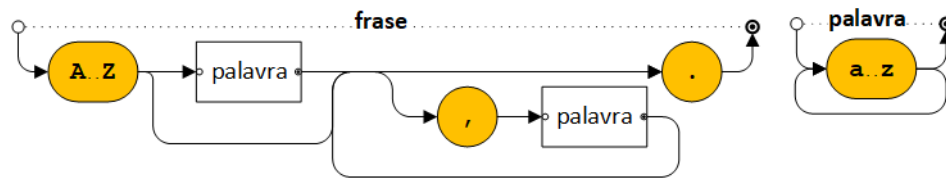


Figura 1.6: Exemplo com Diagramas sintáticos

vírgula antes de cada uma. O diagrama da regra *palavra* indica que é constituída por uma ou várias letras minúsculas (de a a z).

Segundo estas regras os textos:

Kotlin,abc,xpto.

X.

Abc,x.

cumprem a regra *frase*.

Mas os textos:

abc,xpto.

X,abc

A,,.

não cumprem a regra *frase*. O primeiro porque não começa por maiúscula, o segundo porque não termina com ponto e o terceiro porque depois de uma vírgula não tem pelo menos uma letra minúscula.

A árvore de expansão do texto Kotlin,abc,xpto confirma a aceitação desse texto para as regras sintáticas.

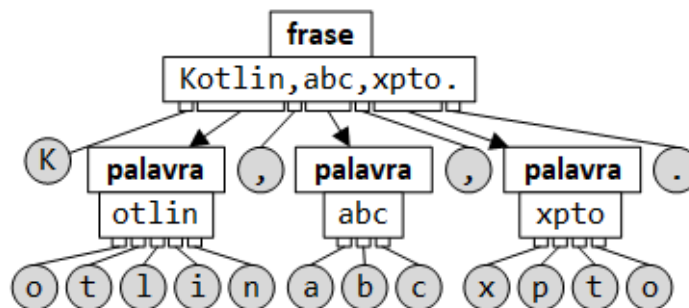


Figura 1.7: Árvore de expansão de um texto

Na linguagem *Kotlin*, os espaços em branco (SPACE), as mudanças de linha (ENTER) e as tabulações (TAB), são considerados símbolos separadores. Na maioria das regras sintáticas, podem ser usados separadores entre os elementos.

Por este motivo, nos diagramas sintáticos apresentados nos restantes capítulos, as ligações direcionais "normais" entre elementos admitem que se possa usar ou não separadores. Quando tal não acontece, são usadas ligações direcionais a tracejado para indicar que os elementos têm que estar seguidos (sem separadores), ou são usadas ligações direcionais a cor azul para indicar que os elementos têm que estar obrigatoriamente separados (com separadores).

Por exemplo, podemos reformular o exemplo da figura 1.6 para o apresentado na figura 1.8.

Assim, passamos a indicar que as letras de cada palavra têm que estar seguidas, mas antes e depois de cada palavra, vírgula ou ponto final, queremos admitir separadores. Além disto, podemos ter várias palavras que em vez de estarem separadas por vírgula, terão obrigatoriamente um separador entre elas.

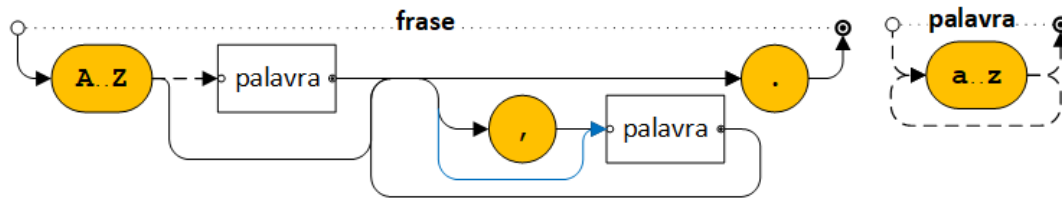


Figura 1.8: Diagramas sintáticos com ligações especiais

1.8 Diagramas de atividade

Nos capítulos seguintes serão usados diagramas de atividade para descrever o fluxo de execução das instruções de um algoritmo ou uma parte dele.

Um diagrama de atividade tem um início, representado por o círculo a cheio, e um fim, representado por um círculo com duplo rebordo. Entre o início e o fim pode haver decisões, representadas por losangos, e ações ou atividades, representadas por retângulos com cantos arredondados.

Por exemplo, o diagrama da figura 1.9 descreve a atividade para processar um email.

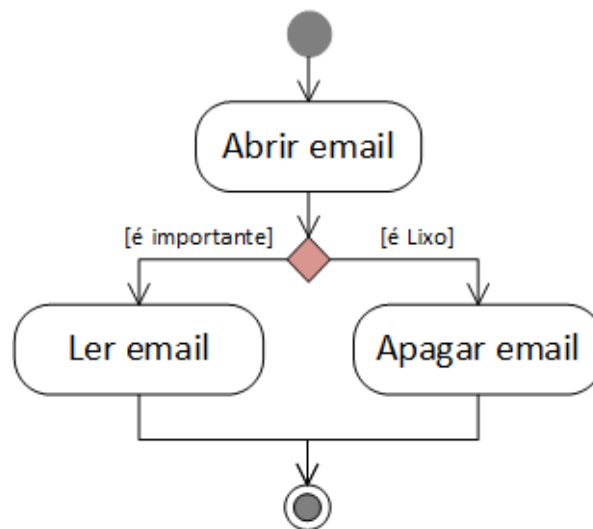


Figura 1.9: Exemplo de diagrama de atividade

Esta atividade tem três ações. A ação "Abrir email" é realizada após o início e antes da decisão. A decisão verifica se o email é lixo ou é importante. Caso seja lixo é realizada a ação "Apagar email" e o processo termina. Caso seja importante é realizada a ação "Ler email" e depois o processo também termina.

Capítulo 2

Tipos, valores e expressões

O armazenamento e a transmissão de informação nos componentes usados pelos computadores digitais só usam informação binária.

Em binário só existem zeros (valor 0) e uns (valor 1). A existência de tensão elétrica num componente, representa o valor 1 e a ausência de tensão elétrica representa o valor 0.

Os computadores usam apenas números para representar todo o tipo de informação e esses números são representados em binário. As memórias só armazenam sequências de números. O processador faz operações entre números.

Os números representam diretamente quantidades, mas as informações que não são quantidades também são representadas por números.

Cada símbolo usado nos textos é representado por um número, por exemplo a letra **A** maiúscula é representada pelo número 65. As imagens são representadas por sequências de números que indicam a cor de cada pixel. Não existe nada que não seja possível representar por números.

Em programação cada tipo de informação tem uma representação interna adequada. Existem tipos para representar valores numéricos inteiros (ex: 27), valores numéricos reais (ex: 9,65), valores lógicos (**true** e **false**) e texto (ex: 'Z' ou "ABC").

Em *Kotlin* cada valor tem um tipo associado. Por exemplo, 27 é um valor do tipo **Int** e 'Z' é um valor do tipo **Char**.

As operações que se podem realizar entre valores dependem dos seus tipos e o resultado é outro valor. Por exemplo: 3+2 é uma soma de dois valores do tipo **Int** e o resultado é também do tipo **Int**, com o valor 5. A expressão 'A'+'B', seria a soma de dois valores do tipo **Char**, mas esta soma não é permitida em *Kotlin*.

Uma expressão é um conjunto de zero ou muitas operações entre valores e o seu resultado é um valor de um determinado tipo. Por exemplo, 3, 3+2 e (27-3)/2 são expressões.

2.1 Números inteiros

Tipicamente, lidamos com números usando a base 10 para os representar.

Por exemplo, o número inteiro 122 tem 3 dígitos na sua representação na base 10: o dígito das centenas (1), o dígito das dezenas (2) e o dígito das unidades (2), em que a posição de cada dígito define o seu peso. $122 = 1 \times 100 + 2 \times 10 + 2 \times 1$, ou usando potências de 10:

$$122_{(10)} = 1 \times 10^2 + 2 \times 10^1 + 2 \times 10^0$$

Usando 3 dígitos na base 10 podemos representar $10^3 = 1000$ valores diferentes, desde 000 a 999.

Representação na base 2

Para representar números na base 2 (binário), os dígitos só podem ser 0 ou 1 e são pesados por potências de 2. Por exemplo, o número 1101 tem 4 dígitos e representa o número 13 na base 10:

$$1101_{(2)} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$\mathbf{1101}_{(2)} = \mathbf{1} \times 8 + \mathbf{1} \times 4 + \mathbf{0} \times 2 + \mathbf{1} \times 1$$

$$\mathbf{1101}_{(2)} = 8 + 4 + 0 + 1 = 13_{(10)}$$

Cada dígito em binário é designado por *bit*. Um conjunto de 8 bits é designado por *byte*. Por exemplo, o valor 122, na base 10, é representado em binário pelo byte 0111 1010.

$$\mathbf{122}_{(10)} = \mathbf{0} \times 2^7 + \mathbf{1} \times 2^6 + \mathbf{1} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{0} \times 2^0$$

$$\mathbf{122}_{(10)} = \mathbf{0} \times 128 + \mathbf{1} \times 64 + \mathbf{1} \times 32 + \mathbf{1} \times 16 + \mathbf{1} \times 8 + \mathbf{0} \times 4 + \mathbf{1} \times 2 + \mathbf{0} \times 1$$

$$\mathbf{122}_{(10)} = 0 + 64 + 32 + 16 + 8 + 0 + 2 + 0 = \mathbf{0111\ 1010}_{(2)}$$

Usando um byte (8 bits) podemos representar $2^8 = 256$ valores diferentes desde 0 a 255.

Para representar também valores negativos em binário, são usadas representações que têm o bit de maior peso a 1. Desta forma, um byte representa valores desde -128 (-2^7) até 127 ($2^7 - 1$), segundo a tabela 2.1.

Tabela 2.1: Representação de valores positivos e negativos num byte

Valor (base 10)	Representação a 8 bits (base 2)
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
...	...
126	01111110
127	01111111
-1	11111111
-2	11111110
-3	11111101
-4	11111100
...	...
-127	10000001
-128	10000000

Os valores negativos são representados usando o código de complementos para 2. Usando este sistema de codificação um valor negativo $-v$ é representado pelos bits invertidos do valor $v - 1$. Por exemplo, as representações de $-4_{(10)}$ e de $-12_{(10)}$ usando 8 bits são:

$$-4_{(10)} = \text{invert}(\mathbf{3}_{(10)}) = \text{invert}(\mathbf{0000\ 0011}_{(2)}) = \mathbf{1111\ 1100}_{(2)}$$

$$-12_{(10)} = \text{invert}(\mathbf{11}_{(10)}) = \text{invert}(\mathbf{0000\ 1011}_{(2)}) = \mathbf{1111\ 0100}_{(2)}$$

Representação na base 16

A base 16 (hexadecimal) também é usada em programação para descrever números grandes. Cada dígito em hexadecimal tem um valor de 0 até 15. As letras A até F são usadas para dígitos com valor de 10 a 15. Por exemplo, o valor 122 na base 10 é representado por 7A na base 16.

$$\mathbf{122}_{(10)} = 7 \times 16^1 + 10 \times 16^0$$

$$\mathbf{122}_{(10)} = 7 \times 16 + 10 \times 1$$

$$\mathbf{122}_{(10)} = \mathbf{7A}_{(16)}$$

Cada dígito hexadecimal corresponde a 4 bits em binário. A tabela 2.2 apresenta a equivalência de valores de 0 a 15 em dígitos na base 16 e bits na base 2.

Tabela 2.2: Decimal, Hexadecimal e Binário

Dec	Hex	Bin	Dec	Hex	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

2.1.1 Tipos *Int*, *Long*, *Byte* e *Short*

27 15L 0b1100 0x4F

Na linguagem *Kotlin*, os números inteiros são representados por 1, 2, 4 ou 8 bytes, aos quais correspondem os tipos *Byte*, *Short*, *Int* e *Long*, respetivamente, de acordo com a tabela 2.3.

Tabela 2.3: Tipos *Kotlin* para números inteiros

Tipo	bits	bytes	valor mínimo	valor máximo
Byte	8	1	$-128 = -(2^7)$	$127 = 2^7 - 1$
Short	16	2	$-32768 = -(2^{15})$	$32767 = 2^{15} - 1$
Int	32	4	$-2147483648 = -(2^{31})$	$2147483647 = 2^{31} - 1$
Long	64	8	$-9223372036854775808 = -(2^{63})$	$9223372036854775807 = 2^{63} - 1$

Em *Kotlin*, um número inteiro é descrito literalmente com um ou mais dígitos, podendo ser sufixado com a letra L e separados por *_* (*underline*). Os dígitos podem descrever o valor na base 10, na base 2 (se começar por 0b) ou na base 16 (se começar por 0x). Os diagramas da figura 2.1 descrevem as regras sintáticas na escrita de literais de valores inteiros.

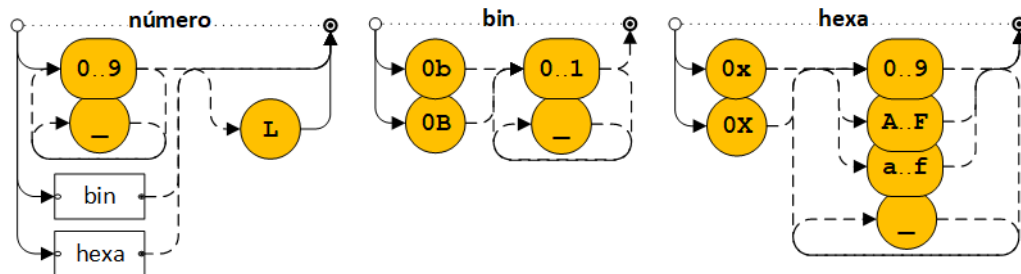


Figura 2.1: Diagramas sintáticos de literais inteiros

Qualquer número inteiro compreendido entre -2147483648 e 2147483647 é considerado do tipo *Int*. Um número inteiro maior que 2147483647 ou menor que -2147483648 ou sufixado com a letra L maiúscula é considerado do tipo *Long*.

Usando o *REPL*, tal como foi descrito na secção 1.6.6, podemos escrever algumas expressões só com literais de números inteiros para verificar estas regras.

```
>>> 27
res1: Int = 27
```

O *REPL* avalia a expressão, que neste caso só tem um valor, e apresenta o local onde armazenou (*res1*), o tipo (*Int*) e o valor (27).

```
>>> 5_000_000_000
res2: Long = 5000000000
>>> 32L
res3: Long = 32
>>> 0x7A
res4: Int = 122
>>> 0b0111_1010L
```

```
res5: Long = 122
```

Não existe forma sintática para descrever literalmente números do tipo `Byte` ou `Short`. No entanto, caso seja necessário, a um valor inteiro de qualquer tipo podem ser aplicadas as funções de conversão `toByte()`, `toShort()`, `toInt()` e `toLong()`. Para tal, basta sufixar o valor com um ponto seguido da função a aplicar.

```
>>> 32.toByte()
res1: Byte = 32
>>> 400.toShort()
res2: Short = 400
>>> 500.toByte() //500 cabe em Byte?
res3: Byte = -12
>>> 0b1111_0100.toByte()
res4: Byte = -12
```

Estas funções de conversão de tipo aproveitam apenas os bits significativos para o tipo destino. Por exemplo, usando apenas os 8 bits de menor peso do valor 500 da representação a 16 bits (do `Short`), ficamos com a representação de -12 num byte.

$$500_{(10)} = 0000\ 0001\ 1111\ 0100_{(2)} \text{ com 16 bits}$$

$$-12_{(10)} = 1111\ 0100_{(2)} \text{ com 8 bits}$$

Operações aritméticas

Os tipos inteiros suportam as quatro operações aritméticas básicas (somas, subtrações, multiplicações e divisões), usando os operadores (+, -, * e /) sendo o resultado um valor do tipo `Int` ou `Long`. Continuando a usar o *REPL*, podemos usar estas operações escrevendo apenas expressões.

```
>>> 27
res1: Int = 27
>>> 27 + 10 // Int + Int --> Int
res2: Int = 37
>>> res2-42 /* Subtração usando o valor armazenado */
res3: Int = -5
>>> 8L /*Long*/ * 5 // Long - Int --> Long
res4: Long = 40
>>> 20/ res3 //Divisão inteira. Se fosse 19/res3 ?
res5: Int = -4
```

Em *Kotlin*, são ignorados todos os espaços escritos entre o operador e os operandos. O texto escrito entre /* e */ e o texto depois de // até ao final da linha também é ignorado, servindo para comentar o código escrito.

As expressões podem usar os valores armazenados pelas avaliações das expressões anteriores.

Operações entre dois valores do tipo `Int` ou de tipos menos abrangentes (`Short` ou `Byte`) têm um resultado do tipo `Int`. Operações em que um dos argumentos seja do tipo `Long` tem um resultado do tipo `Long`.

A divisão de inteiros tem dois resultados, o quociente e o resto. O operador / calcula o quociente e o operador % calcula o resto.

```
>>> 26/3 // 26 é o dividendo e 3 é o divisor
res6: Int = 8 // Quociente
>>> 26 % 3
res7: Int = 2 // Resto
>>> 3 * res6 + res7 // divisor * quociente + resto = dividendo
res8: Int = 26
```

No caso da expressão ter mais do que um operador eles são avaliados pela sua prioridade.

Os três operadores multiplicativos (*, / e %) são mais prioritários que os aditivos (+ e -).

A utilização de parêntesis permite forçar a ordem de avaliação.

```
>>> 3 * 8 + 2
res9: Int = 26
>>> 2 + 3 * 8 //Primeiro a multiplicação
```

```
res10: Int = 26
>>> (2 + 3) * 8 //Primeiro a soma
res11: Int = 40
```

Os três operadores multiplicativos (*, / e %) têm a mesma prioridade, assim como os dois operadores aditivos (+ e -). Quando dois ou mais operadores com a mesma prioridade estão consecutivos, eles são avaliados da esquerda para a direita.

```
>>> 2 * 3 / 2 // 0 mesmo que 6/2
res12: Int = 3
>>> 3 / 2 * 2 // 0 mesmo que 1*2
res13: Int = 2
```

Os operadores - e + também podem ser usados como unários a prefixar cada valor.

```
>>> -5
res14: Int = -5
>>> -(2-3)
res15: Int = 1
>>> ++27
res16: Int = -27
```

Utilizando apenas os operadores aritméticos já descritos, podemos resumir as regras sintáticas das expressões aos diagramas apresentados na figura 2.2.

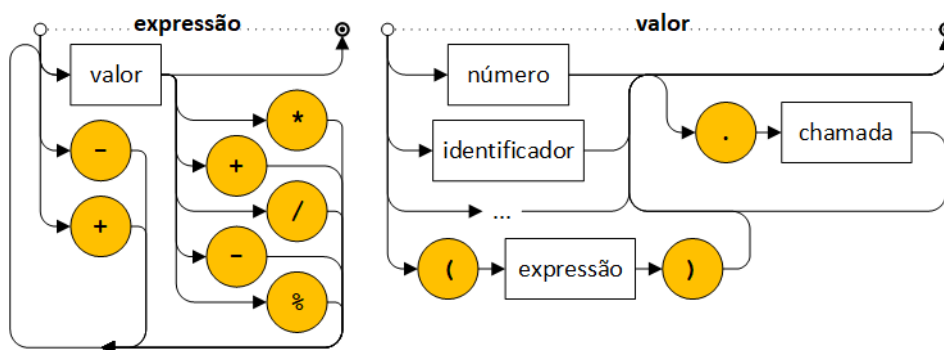


Figura 2.2: Diagramas sintáticos das expressões com inteiros

Os dois diagramas sintáticos da figura 2.2 descrevem as regras *expressão* e *valor*. A regra *número* foi apresentada na figura 2.1. A regra *chamada* especifica a chamada a uma função de conversão. A regra *identificador* especifica o nome de um valor armazenado. Estas duas últimas regras serão apresentadas mais adiante.

As reticências no diagrama *valor* indicam que neste local o diagrama será expandido para suportar outros valores.

Uma expressão pode ser apenas um valor.

Da avaliação de uma expressão resulta um valor cujo tipo depende dos operadores e dos tipos dos valores operados.

2.2 Declaração de valores

```
val ten : Int = 10
```

É possível armazenar valores indicando explicitamente o seu identificador (nome) e opcionalmente o seu tipo. A figura 2.3 apresenta o diagrama sintático desta declaração.

A declaração atribui um nome ao valor da expressão.

A declaração começa pela palavra reservada *val*. A regra *tipo* é o nome de um dos tipos em *Kotlin*. Quando o tipo é omitido é assumido o tipo do valor que resulta da expressão.

A regra *expressão* foi apresentada no diagrama da figura 2.2

A regra *identificador* é uma sequência de símbolos, que não seja uma palavra reservada, sem espaços e em que o primeiro símbolo tem que ser obrigatoriamente uma letra. O diagrama da figura 2.4 apresenta esta regra sintática, assumindo que *LETRA* é qualquer letra minúscula ou maiúscula com ou sem acentos.

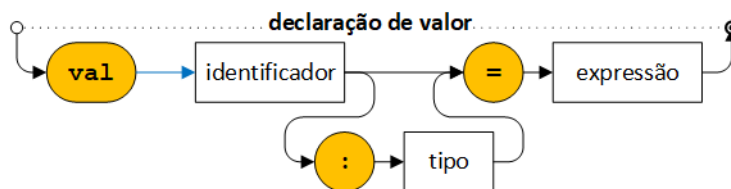
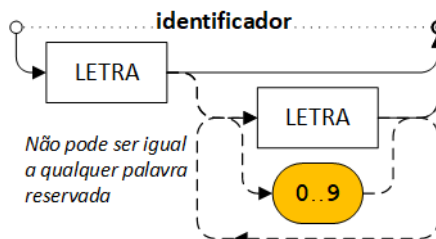
Figura 2.3: Diagrama sintático da declaração de `val`

Figura 2.4: Diagrama sintático dos identificadores

Apesar de não ser regra da linguagem, os programadores de *Kotlin* usam a convenção *lowerCamelCase* na atribuição destes nomes. Esta convenção estabelece que o nome é escrito em minúsculas e se for composto por várias palavras é usada uma maiúscula no início das palavras seguintes. Por exemplo, um valor que armazena o número de pessoas numa sala poderia ter o nome `peopleInRoom`.

A linguagem *Kotlin* é fortemente tipificada. Todos os valores têm um tipo e a compatibilidade dos tipos é verificada pelo compilador.

Quando a declaração não explicita o tipo, este é inferido do valor da expressão.

Quando a declaração indica o tipo explicitamente e a expressão só usa literais não tipificados, então o tipo da expressão é adaptado ao tipo da declaração.

As declarações não são expressões, como tal não produzem um resultado.

```
>>> val ten = 10    //Int é o tipo inferido
>>> ten
res1: Int = 10
>>> val number = 4000000000    // Long é o tipo inferido
>>> number
res2: Long = 4000000000
>>> val bigNumber :Long = 1000    // Long <- Int
>>> ten + bigNumber    // Int+Long -> Long
res3: Long = 1010
>>> val dayOfMonth :Byte = 12    // Byte <- Int
>>> dayOfMonth
res4: Byte = 12
>>> val myByte :Byte = 500
error: the integer literal does not conform to the expected type Byte
val myByte :Byte = 500
~
>>> val myByte :Byte = ten
error: type mismatch: inferred type is Int but Byte was expected
val myByte :Byte = ten
~
```

No exemplo anterior, os tipos dos valores 1000 e 12 foram adaptados, porque 1000 cabe em `Long` e 12 cabe em `Byte`. A declaração do valor `myByte` não foi permitida porque na primeira tentativa 500 não cabe em `Byte` e na segunda tentativa a expressão não é literal, por isso o tipo não é adaptado.

2.2.1 Inteiros positivos

Para representar números inteiros positivos (*unsigned*) existem também em *Kotlin* os tipos `UByte`, `UShort`, `UInt` e `ULong`, apresentados na tabela 2.4. Estes tipos permitem representar o dobro da

gama de valores positivos, relativamente ao respetivo tipo sem o prefixo U, usando o mesmo número de bits.

Tabela 2.4: Tipos *Kotlin* para números inteiros positivos

Tipo	bits	bytes	valor mínimo	valor máximo
UByte	8	1	0	$255 = 2^8 - 1$
UShort	16	2	0	$65535 = 2^{16} - 1$
UInt	32	4	0	$4294967295 = 2^{32} - 1$
ULong	64	8	0	$18446744073709551615 = 2^{64} - 1$

Um número inteiro positivo é descrito literalmente acrescentando a letra U ou u como sufixo do valor. O Diagrama da figura 2.5 é uma extensão do diagrama da figura 2.1 para suportar também números inteiros sem sinal.

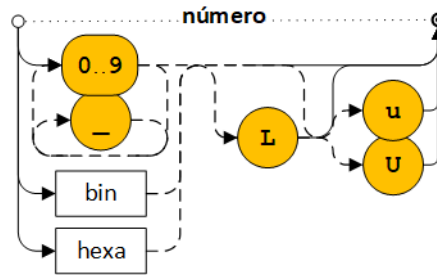


Figura 2.5: Diagrama sintático de literais inteiros com ou sem sinal

As funções de conversão `toUByte()`, `toUShort()`, `toUInt()` ou `toULong()` convertem para valores sem sinal usando a representação binária do valor original.

```
>>> val v1: Byte = 250 // Valor máximo positivo de Byte é 127
error: the integer literal does not conform to the expected type Byte
val v1 :Byte = 250
~

>>> val v1: UByte = 250u // Valor máximo de UByte é 255
>>> v1
res1: UByte = 250
>>> val x = 4000000000 // Tipo inferido é Long
>>> x
res2: Long = 4000000000
>>> x.toUInt() // Valor máximo de UInt é 4294967295
res3: UInt = 4000000000
>>> (-12).toUByte() // 0b1111_0100 = -12 (com sinal)
res4: UByte = 244 // 0b1111_0100 = 244 (sem sinal)
```

Os tipos inteiros sem sinal têm as mesmas operações aritméticas e as mesmas regras que os inteiros com sinal.

2.3 Números reais

Os números reais também são representados em binário, usando potências de 2. Mas é necessário usar potências negativas para representar os dígitos depois da vírgula. As potências negativas de 2 são o inverso das potências positivas, ou seja, $2^{-x} = 1/2^x$. Por exemplo, o número real 6,625 na base 10 seria:

$$6,625_{(10)} = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$6,625_{(10)} = 1 \times 4 + 1 \times 2 + 0 \times 1 + 1 \times 1/2^1 + 0 \times 1/2^2 + 1 \times 1/2^3$$

$$6,625_{(10)} = 1 \times 4 + 1 \times 2 + 0 \times 1 + 1 \times 1/2 + 0 \times 1/4 + 1 \times 1/8$$

$$6,625_{(10)} = 1 \times 4 + 1 \times 2 + 0 \times 1 + 1 \times 0,5 + 0 \times 0,25 + 1 \times 0,125$$

$$6,625_{(10)} = 110,101_{(2)}$$

De facto, a representação interna é um pouco mais complicada, porque é usada uma representação com vírgula flutuante, que normaliza o número para $1, \dots \times 2^n$ e depois armazena os bits da mantissa (os bits depois da vírgula) e do expoente (n) da potência de 2. Para o exemplo anterior seria:

$$110,101_{(2)} = 1,10101 \times 2^{10}_{(2)}$$

$$\text{mantissa} = 10101 \text{ e } \text{expoente} = 10$$

O *Kotlin*, assim como a grande maioria das linguagens de programação, usa a norma *IEEE754*, que estabelece esta representação e define o número de bits usados na mantissa, no expoente e no bit de sinal, dependendo da precisão necessária. Em *Kotlin* é usada a precisão simples (6 a 7 dígitos na base 10) ou precisão dupla (15 a 16 dígitos), aos quais correspondem os tipos `Float` e `Double` apresentados na tabela 2.5.

Tabela 2.5: Tipos *Kotlin* para números reais

Tipo	bytes	bits	sinal	mantissa	expoente	precisão (dígitos)
<code>Float</code>	4	32	1	23	8	6 a 7
<code>Double</code>	8	64	1	52	11	15 a 16

Caso não tenha entendido todos os detalhes sobre a representação dos números reais, não se preocupe. O importante é saber que existem dois tipos para representar números reais em *Kotlin*: `Float` para precisão simples e `Double` para precisão dupla. Em que `Float` ocupa 4 bytes (32 bits) e `Double` ocupa 8 bytes (64 bits).

2.3.1 Tipos *Float* e *Double*

32.54 6.625f 3.5e3

Na linguagem *Kotlin*, um número real é descrito literalmente com um ou mais dígitos de 0 a 9 seguido de . (ponto decimal) e um ou mais dígitos depois do ponto decimal. Pode ser indicado um expoente em notação científica e pode ser sufixado com a letra `F` ou `f`. O diagrama da figura 2.6 é uma extensão do diagrama da figura 2.5 para suportar também números reais usando a parte delimitada a tracejado.

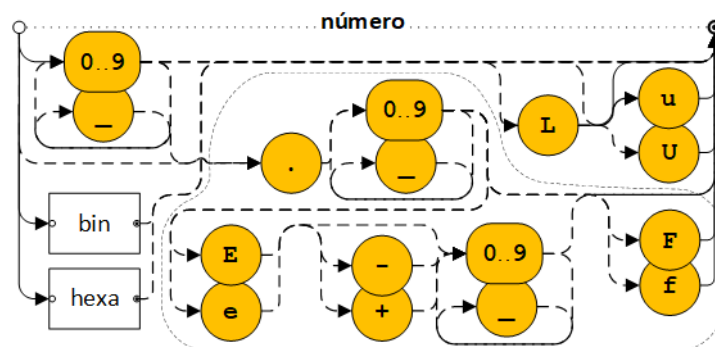


Figura 2.6: Diagrama sintático de literais inteiros ou reais

Qualquer número sem sufixo mas com ponto decimal ou com expoente é considerado do tipo `Double`. Um número sufixado com a letra `f` ou `F` é considerado do tipo `Float`.

Usando novamente o *REPL* podemos confirmar estas regras para os números reais.

```
>>> 32.54
res1: Double = 32.54
>>> 6.625F
res2: Float = 6.625
>>> 5.2e3
res3: Double = 5200.0
>>> 27F
res4: Float = 27.0
>>> 3e2
res5: Double = 300.0
```



```
>>> 0.0
res6: Double = 0.0
>>> .5f
res7: Float = 0.5
>>> val pi = 3.14
>>> pi
res8: Double = 3.14
```

As funções de conversão de tipo `toInt()`, `toLong()`, `toUInt()` e `toULong()` podem ser aplicadas também a valores reais, ficando apenas a parte inteira, sem arredondamentos. As funções `toUInt()` e `toULong()` retornam zero se forem aplicadas a valores reais negativos. As funções `toFloat()` e `toDouble()` podem ser aplicadas a valores inteiros e reais.

```
>>> 65.98.toInt() // sem arredondamento
res1: Int = 65
>>> 3.5321e2.toLong()
res2: Long = 353
>>> 45.toFloat()
res3: Float = 45.0
>>> -23.67.toInt()
res4: Int = -23
>>> -23.67.toUInt()
res5: UInt = 0
```

Os tipos reais também suportam as quatro operações aritméticas básicas sendo o resultado um valor do tipo `Float` ou `Double`. Enquanto as divisões inteiras têm dois resultados (quociente e resto) obtidos com os operadores `/` e `%`, as divisões reais têm um só resultado com o operador `/`. Apesar de não ser comum, o operador `%` também pode ser usado entre reais para obter o resto, simulando a divisão inteira.

As operações podem envolver valores reais e inteiros, mas estas serão realizadas no tipo mais abrangente. Pela ordem: $Double \Rightarrow Float \Rightarrow Long \Rightarrow Int$. Ou seja, operações em que um dos argumentos é do tipo `Double` tem um resultado do tipo `Double`. Caso contrário, se um dos argumentos for um `Float` o resultado será um `Float`. Caso contrário, se um dos argumentos for `Long` o resultado é `Long`. Caso contrário, o resultado será do tipo `Int`.

```
>>> 4 * 5.3F // Int * Float -> Float
res1: Float = 21.2
>>> 4.0 * 5 // Double * Int -> Double
res2: Double = 20.0
>>> 43.5 / 4
res3: Double = 10.875
>>> 43.5 % 4
res4: Double = 3.5
```

Qualquer valor real que não seja possível decompor em potências de 2 terá uma representação do seu valor aproximado. Por exemplo, o valor 0,75 tem uma representação exata $0,75_{(10)} = 2^{-1} + 2^{-2} = 0,5 + 0,25$, mas 0,1 tem uma representação aproximada $0,1_{(10)} = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-8} \dots = 0.100000001490$, usando os 23 bits possíveis da mantissa do `Float`.

Usando valores do tipo `Double` o erro será menor, mas haverá sempre erro nas representações não exatas.

```
>>> val price = 5.30f //Cinco euros e 30 cêntimos
>>> price - 0.10f //10 cêntimos de desconto
res1: Float = 5.2000003
>>> 0.10f.toDouble()
res2: Double = 0.10000000149011612
```

Devido aos valores aproximados na utilização de `Float` e `Double`, é comum resolver este tipo de problemas com valores inteiros. Neste exemplo, basta lidar com os preços em cêntimos, em vez de euros.

```
>>> val price = 530 //530 cêntimos
>>> price - 10 //10 cêntimos de desconto
res1: Int = 520
>>> (price-10)/100F //Conversão para euros
res2: Float = 5.2
```

2.4 Texto

Cada símbolo utilizado em texto é codificado por um número inteiro segundo a tabela de codificação *Unicode*. O texto é uma série ordenada de símbolos e é armazenado internamente numa sequência de códigos em *Unicode*.

A tabela *Unicode* codifica todos os símbolos internacionalmente conhecidos usando inteiros positivos a 16 bits, podendo codificar até 65536 símbolos. A tabela 2.7 apresenta alguns desses códigos na base 10 (decimal) e na base 16 (hexadecimal).

Convenientemente, os códigos das letras maiúsculas de 'A' a 'Z' foram atribuídos sequencialmente, assim como as letras minúsculas de 'a' até 'z' e os dígitos de '0' a '9'.

Tal como mostra a tabela 2.6, na linguagem *Kotlin* existem dois tipos para texto. O tipo `Char` para um símbolo e o tipo `String` para uma sequência de símbolos.

Tabela 2.6: Tipos *Kotlin* para texto

Tipo	bytes	descrição
<code>Char</code>	2	Um símbolo/caráter em <i>Unicode</i>
<code>String</code>	variável	Uma sequência de zero ou mais símbolos

Tabela 2.7: Alguns símbolos da tabela *Unicode*

Maiúsculas			Minúsculas			Dígitos			Outros		
sim	dec	hex	sim	dec	hex	sim	dec	hex	sim	dec	hex
A	65	41	a	97	61	0	48	30	!	33	21
B	66	42	b	98	62	1	49	31	"	34	22
C	67	43	c	99	63	2	50	32	#	35	23
D	68	44	d	100	64	3	51	33	(40	28
E	69	45	e	101	65	4	52	34)	41	29
F	70	46	f	102	66	5	53	35			
G	71	47	g	103	67	6	54	36			
H	72	48	h	104	68	7	55	37	ç	231	E7
I	73	49	i	105	69	8	56	38	ã	227	E3
J	74	4A	j	106	6A	9	57	39			
K	75	4B	k	107	6B				€	8364	20AC
...				☎	9742	260E
Z	89	59	z	122	7A				©	9786	263A

2.4.1 Tipo *Char*

```
'A' '3' '\n'
```

Em *Kotlin* um literal do tipo `Char` é um símbolo entre plicas. Por exemplo, 'A' é um `Char`.

Por vezes é necessário identificar símbolos especiais com uma sequência de escape entre plicas. Esta sequência começa com o símbolo \ (barra invertida ou *backslash*) seguida de outro símbolo que identifica o símbolo especial. Por exemplo, '\t' é o símbolo `TAB` ou →| usado para fazer tabulação. Também é possível indicar o código do símbolo com 4 dígitos em hexadecimal depois de \u.

Os diagramas da figura 2.7 apresentam a regra sintática dos literais do tipo `Char` e a tabela 2.8 identifica as sequências de escape.

A regra *símbolo* deverá ser acrescentada ao ponto de expansão da regra *valor* apresentada na figura 2.2.

A regra SÍMBOLO do diagrama sintático significa qualquer símbolo existente em *Unicode* que seja possível escrever exceto os símbolos ' (plica) e \ (barra invertida).

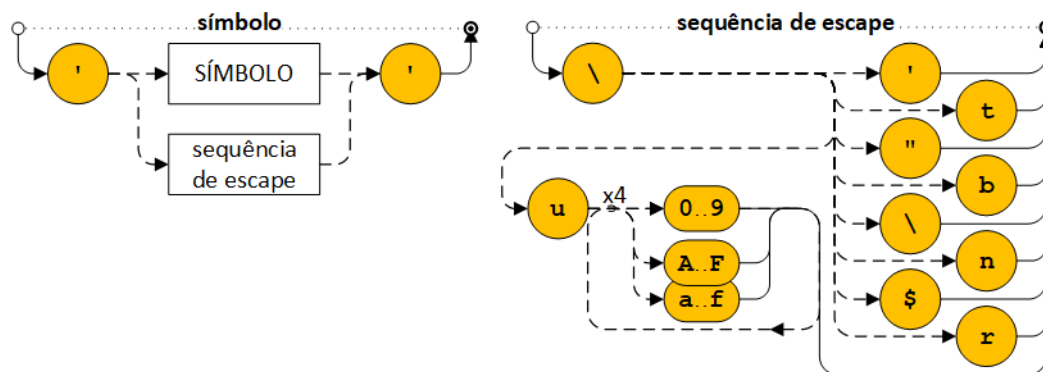
Figura 2.7: Diagramas sintáticos de literais *Char*

Tabela 2.8: Sequências de escape

Sequência	Símbolo
<code>\'</code>	<code>'</code>
<code>\"</code>	<code>"</code>
<code>\\</code>	<code>\</code>
<code>\\$</code>	<code>\$</code>
<code>\t</code>	Tabulação (TAB)
<code>\b</code>	<i>backspace</i>
<code>\n</code>	Mudança de linha (<i>Line feed</i>)
<code>\r</code>	Início de linha (<i>Carriage return</i>)
<code>\uHHHH</code>	Com o código <i>Unicode</i> HHHH em hexadecimal

Usando o REPL podemos constatar algumas propriedades do tipo `Char`.

```
>>> 'K'
res1: Char = K
>>> 'K'.code    //Obtém o código do Char
res2: Int = 75
>>> '\''
res3: Char = '
>>> val x = '\u0041'
>>> x
res4: Char = A
>>> x.code
res5: Int = 65
```

Cada valor do tipo `Char` tem a propriedade `code` do tipo `Int` com o código do símbolo.

Valores do tipo `Char` ou do tipo `Short` ocupam 16 bits (2 bytes), mas os valores do tipo `Short` apresentam-se como inteiros e têm aritmética de inteiros e os valores do tipo `Char` apresentam-se como símbolos e têm aritmética própria.

Os valores do tipo `Char` podem ser usados como números apenas nas seguintes operações: Somar ou subtrair um valor do tipo `Int` a um `Char` resulta noutro `Char` cujo código é igual ao resultado da operação; Subtrair dois valores do tipo `Char` resulta num `Int` que corresponde à diferença entre os códigos dos valores. Na soma ou na subtração entre `Char` e `Int`, o `Char` tem que ser o argumento esquerdo.

```
>>> x+4
res6: Char = E
>>> 'E'-'A'
res7: Int = 4
>>> 'E'-2
res8: Char = C
>>> 2+'C'
error: operation not allowed
2+'C'
^
```

A diferença entre os códigos de uma letra minúscula e a maiúscula correspondente é de ('a'-'A'). Para transformar uma maiúscula numa minúscula basta somar essa diferença.

```
>>> val letter = 'F'
>>> letter.code
res1: Int = 70
>>> val lower = letter + ('a'-'A')
>>> lower
res2: Char = f
>>> lower.code
res1: Int = 102
```

Neste exemplo de utilização, quando é avaliada a expressão `letter + ('a'-'A')`, são realizadas as seguintes avaliações parciais:

`letter + ('a'-'A') ⇒ 'F' + ('a'-'A') ⇒ 'F' + 32 ⇒ 'f'`

em que é avaliado em primeiro lugar o valor de `letter`, resultando 'F'. É realizada a subtração 'a'-'A', resultando 32, que é a distância entre os códigos das maiúsculas e das minúsculas na tabela *Unicode*. Por fim, é realizada a soma 'F' + 32, resultando o valor 'f' do tipo `Char`.

Aos valores inteiros pode ser aplicada a função de conversão `toChar()` para obter o símbolo respetivo.

```
>>> (65+4).toChar()
res1: Char = E
>>> 'ã'.code
res2: Int = 227
```

O código em *Unicode* do dígito não é o valor do dígito. Para obter o valor do dígito é necessário subtrair o código do dígito '0' (zero).

```
>>> val dig :Char = '5'
>>> dig.code
res1: Int = 53
>>> dig-'0'
res2: Int = 5
```

Para obter o valor do dígito podemos usar a função de conversão `digitToInt()` aplicada a `Char`, assumindo que o símbolo é um dígito.

```
>>> dig.digitToInt()
res3: Int = 5
```

2.4.2 Tipo *String*

"Hello"

Em *Kotlin* um literal do tipo `String` é uma sequência de símbolos entre aspas. Por exemplo, "Hello World!" é uma `String`.

```
>>> "Hello World!"
res1: String = Hello World!
>>> "123"
res2: String = 123
>>> val str = "abc+123"
>>> str
res3: String = abc+123
```

Um valor de qualquer tipo pode ser convertido em texto usando a função de conversão `toString()`.

```
>>> val v = 28.7
>>> v.toString()
res4: String = 28.7
```

Expressões embutidas

É possível definir *strings* com o valor embutido de expressões (*string template*). Para embutir um valor a partir do seu nome, basta colocar o símbolo \$ antes do nome, desde que o símbolo depois do nome sirva de separador.

```
>>> val city = "Lisboa"
>>> "Cidade = $city"
res1: String = Cidade = Lisboa
```

Quando o valor embutido não é do tipo `String`, o valor é implicitamente convertido usando a função de conversão `toString()`.

```
>>> val price = 2048
>>> val textPrice = price.toString()
>>> "Preço é $textPrice"
res2: String = Preço é 2048
>>> "Preço é $price"
res3: String = Preço é 2048
```

Para embutir valores de expressões, ou quando o símbolo a seguir ao nome não é separador, é necessário limitar a expressão entre chavetas.

```
>>> val start = "Multipilca"
>>> "$start: $ax$b = $a*b"
error: unresolved reference: ax
>>> "$start: ${a}*b = $a*b"
res4: String = Multiplica: 2x15 = 2*b
>>> "$start: ${a}*b = ${a*b}"
res5: String = Multiplica: 2x15 = 30
```

Sequências de escape


Para incluir símbolos especiais em literais do tipo `String` são usadas sequências de escape, tal como são usadas em literais do tipo `Char`. A tabela 2.8 apresenta as sequências possíveis. Por exemplo, para incluir aspas no conteúdo é necessário usar a sequência de escape `\"`.

```
>>> "Cidade \"Lisboa\" \$\\" // "\" = '$'    "\\\" = '\ '
res6: String = Cidade "Lisboa" $\
>>> "Linha1\nLinha2\nLinhaX\b3" // "\n" = line feed  "\b" = backspace
res7: String = Linha1
Linha2
Linha3
```

Texto em bruto

Existe uma forma de indicar *raw strings* que podem conter qualquer símbolo tal como são escritos (texto em bruto), ignorando as sequências de escape. Estas *strings* são delimitadas por 3 aspas `"""` e podem incluir mudanças de linha.

```
>>> """Cidade \"Lisboa\" \$\
...Segunda Linha"""
res8: String = Cidade \"Lisboa\" \$\
Segunda Linha
```

O *REPL* apresenta ... depois de introduzido  quando a expressão ou a instrução ainda não terminou, continuando ler o texto introduzido.

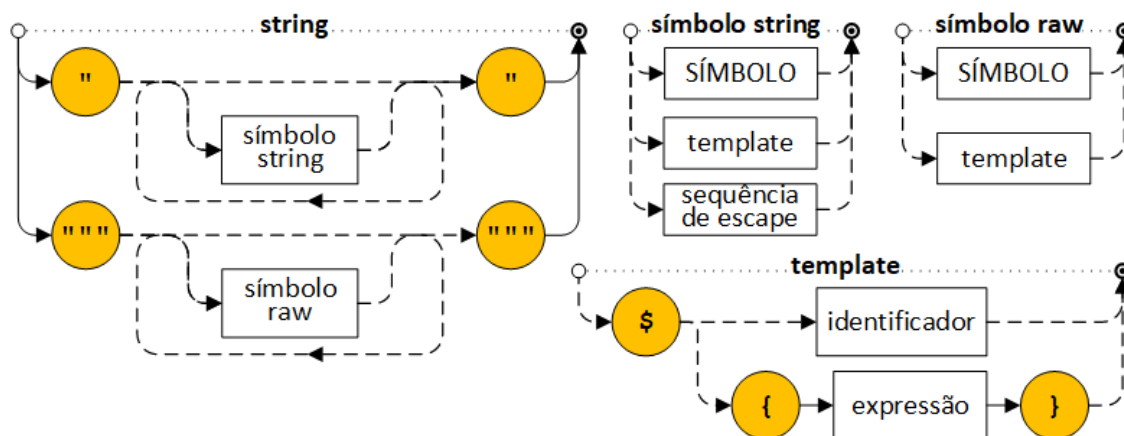
Sintaxe

Os diagramas sintáticos da figura 2.8 descrevem as diferentes possibilidades de definir um literal do tipo `String`, sendo a regra *string* acrescentada ao ponto de expansão da regra *valor* apresentada na figura 2.2.

A regra SÍMBOLO do diagrama sintático *símbolo string* significa qualquer símbolo existente em *Unicode* que seja possível escrever exceto os símbolos `"` (aspas) e `\` (barra invertida). Mas a regra SÍMBOLO do diagrama *símbolo raw* pode ser qualquer símbolo.

Conversões

Para reconhecer como números inteiros, *strings* com sequências de dígitos, podem ser aplicadas as funções de conversão para inteiros (`toInt()`, `toByte()`, `toShort()` e `toLong()`), podendo ser indicada a base de numeração (de 2 a 36), sendo por omissão a base 10.

Figura 2.8: Diagramas sintáticos de literais *String*

```
>>> val num = "1101"
>>> num.toInt() //Assume dígitos na base 10
res1: Int = 1101
>>> num.toByte(2) //Assume dígitos na base 2
res2: Byte = 13
>>> num.toInt(16) //Assume dígitos na base 16
res3: Int = 4353
```

No entanto, se a *string* tiver símbolos que não são válidos na base de numeração usada, em vez de resultar um valor na avaliação da expressão é lançada uma exceção.

```
>>> val num = "12A"
>>> num.toInt(16) //Na base 16 são também usados dígitos de A..F
res1: Int = 298 //12A na base 16 --> 298 na base 10
>>> num.toInt(10) //Lança exceção na avaliação da expressão
NumberFormatException: For input string: "12A"
```

Para obter textualmente os dígitos dos valores inteiros pode ser aplicada a função de conversão `toString()`, podendo também ser indicada a base de numeração.

```
>>> (65+4).toString()
res1: String = 69
>>> 13.toString(2)
res2: String = 1101
```

Para reconhecer *strings* com os dígitos de valores reais, podem ser aplicadas a funções de conversão para reais (`toFloat()` e `toDouble()`). Aos valores reais pode ser aplicada a função de conversão `toString()` para obter a representação textual.

```
>>> val num = "3.5e2"
>>> num.toFloat()
res1: Float = 350.0
>>> val x = 23e4f
>>> x.toString()
res2: String = 230000.0
```

A valores do tipo `Char` também pode ser aplicada a função de conversão `toString()`, mas o inverso não, ou seja, não existe a função de conversão `toChar()` aplicada a *string*, nem faria sentido existir.

```
>>> val symb = 'X'
>>> symb.toString()
res1: String = X
>>> "abc".toChar()
error: unresolved reference: toChar
```

Operações com strings

O operador `+` é usado para concatenar o texto de uma *string* com o texto que resulte de qualquer outro valor, mas o argumento esquerdo tem que ser a *String*. Caso o argumento direito não seja

do tipo `String`, esse será convertido automaticamente usando a função de conversão `toString()`.

```
>>> "abc"+"xpto"
res1: String = abcxpto
>>> val num = 27
>>> "abc"+num
res2: String = abc27
>>> "bin="+num.toString(2)
res3: String = bin=11011
```

O operador `[]` permite obter o símbolo (do tipo `Char`) numa determinada posição da string. A propriedade `length` é do tipo `Int` e tem o comprimento total de símbolos. O primeiro símbolo está na posição zero e o último símbolo está na posição `length-1`.

```
>>> val text = "abcd"
>>> text.length
res1: Int = 4
>>> text[0]
res2: Char = a
>>> text[3]
res3: Char = d
```

A figura 2.9 mostra a representação da `String` para melhor compreensão do operador `[]` e da propriedade `length`.

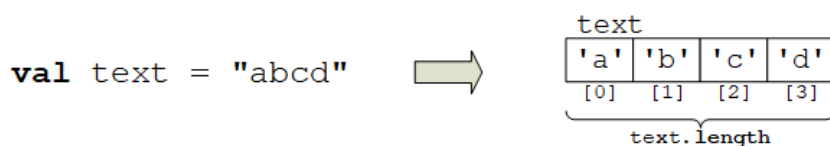


Figura 2.9: Representação de *String*

2.5 Lógica

George Boole foi um matemático britânico que criou a álgebra booleana que permite lidar com relações lógicas.

Nesta álgebra só existem os valores lógicos verdadeiro e falso. Os operadores lógicos principais realizam as operações *OR* (Ou lógico), *AND* (E lógico) e *NOT* (Negação lógica).

2.5.1 Tipo *Boolean*

true false

O tipo `Boolean` representa apenas os valores lógicos: `true` (verdadeiro) e `false` (falso).

Sintaticamente, um valor literal do tipo `Boolean` é uma das palavras reservadas da linguagem: `true` ou `false`, tal como apresenta o diagrama sintático da figura 2.10. A regra *lógico* terá que ser acrescentada ao ponto de expansão da regra *valor* apresentada na figura 2.2.

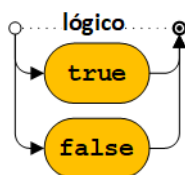


Figura 2.10: Diagrama sintático de literais *Boolean*

```
>>> val yes : Boolean = true      // Constante do tipo Boolean
>>> yes
res1: Boolean = true
>>> val logic = yes
```

```
>>> logic
res2: Boolean = true
```

O resultado das operações de comparação são do tipo `Boolean`. Por exemplo, comparar dois valores inteiros ou reais com o operador `>` resulta num valor do tipo `Boolean`.

```
>>> val x = 27
>>> x > 10
res1: Boolean = true
>>> val greater = 32.5 > 200.7
>>> greater
res2: Boolean = false
```

Operações de comparação

Para verificar se dois valores são iguais é usado o operador `==` e para verificar se são diferentes é usado o operador `!=`, dos quais também resulta um valor do tipo `Boolean`.

```
>>> val y = 32
>>> val equals = (y == 32)
>>> equals
res1: Boolean = true
>>> y != 32
res2: Boolean = false
```

As operações de comparação do *Kotlin* são as apresentadas na tabela 2.9.

Tabela 2.9: Operadores de comparação

Operador	Descrição	Operador	Descrição
<code>></code>	maior que	<code>>=</code>	maior ou igual a
<code><</code>	menor que	<code><=</code>	menor ou igual a
<code>==</code>	igual a	<code>!=</code>	diferente de

Estas operações só são permitidas entre valores do mesmo tipo. Em alguns casos faz sentido a comparação de tipos diferentes, como é o caso da comparação entre inteiros e reais.

```
>>> 'C' > 'A'
res1: Boolean = true
>>> 56.3f < 10
res2: Boolean = false
>>> false == true
res3: Boolean = false
>>> "abc" != "ABC"
res4: Boolean = true
>>> 3 >= 10
res5: Boolean = false
>>> 3 <= 3.0
res6: Boolean = true
```

Operações lógicas

Para realizar operações entre valores lógicos existem os operadores: `&&` para realizar o (*AND* - *E lógico*), `||` para realizar o (*OR* - *OU lógico*) e o operador `!` para realizar o (*NOT* - *Negação lógica*). O resultado da avaliação destes operadores é o indicado na tabela 2.10.

Tabela 2.10: Operadores lógicos

<i>AND</i>	Res.	<i>OR</i>	Res.	<i>NOT</i>	Res.
false && false	false	false false	false	! false	true
false && true	false	false true	true	! true	false
true && false	false	true false	true		
true && true	true	true true	true		

Usando os operadores de comparação e os operadores lógicos, escrevemos expressões que verificam condições.


```

>>> !(1 == 2)
res1: Boolean = true
>>> val symb = 'S'
>>> symb>='A' && symb<='Z'
res2: Boolean = true
>>> val isDigit = symb>='0' && symb<='9'
>>> isDigit
res3: Boolean = false
>>> val num = 27
>>> val isOdd = num % 2 != 0
>>> val isEven = !isOdd
>>> isEven
res4: Boolean = false

```

2.6 Intervalos

1..10

O operador `..` permite a definição de intervalos de valores dos tipos enumeráveis. Por exemplo, `1..10` representa o intervalo de 1 até 10, incluindo os extremos. Como o tipo `Char` também é enumerável, é possível definir intervalos de símbolos, por exemplo, o intervalo `'A'..'F'`.

O operador `in` verifica se um valor pertence a um intervalo e o resultado é do tipo `Boolean`.

```

>>> 15 in 0..20           // Equivalente a: 15>=0 && 15<=20
res1: Boolean = true
>>> val upperCase = 'A'..'Z'
>>> 'r' in upperCase      // Equivalente a: 'r'>='A' && 'r'<='Z'
res2: Boolean = false
>>> 'R' in upperCase      // Equivalente a: 'R'>='A' && 'R'<='Z'
res3: Boolean = true

```

Aos intervalos pode ser aplicada a função `count()` para obter o número de elementos. As propriedades `first` e `last` permitem consultar o primeiro e o último elemento do intervalo.

```

>>> val range = 3..15
>>> range.first
res1: Int = 3
>>> range.last
res2: Int = 15
>>> range.count()
res3: Int = 13

```

Os intervalos também podem ser definidos excluindo o último elemento, usando a palavra `until` em vez de `..` (ponto ponto). Por exemplo, `1 until 8` é equivalente a `1..7`.

```

>>> val x = 1 until 8
>>> 8 in x
res1: Boolean = false
>>> 1..7 == x
res2: Boolean = true
>>> x.last
res3: Int = 7

```

Opcionalmente, pode ser indicado o passo do intervalo, usando a palavra `step`. Por exemplo, `1..9 step 2` indica que os elementos do intervalo progridem de 2 em 2.

Também é possível definir intervalos por ordem decrescente, usando a palavra `downTo` em vez de `..` (ponto ponto).

```

>>> val d = 12 downTo 1 step 2
>>> d.last
res1: Int = 2
>>> d.count()
res2: Int = 6
>>> 9 in d
res3: Boolean = false

```

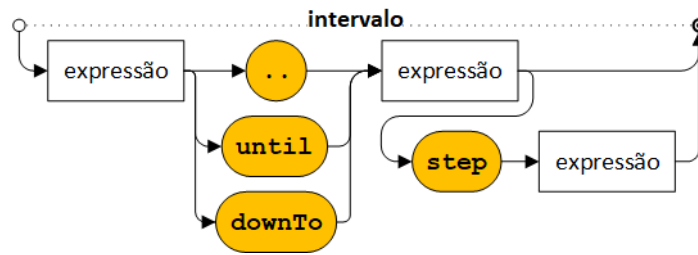


Figura 2.11: Diagrama sintático de literais de intervalos

O diagrama da figura 2.11 apresenta a regra sintática para definir um intervalo de valores enumeráveis, sendo a regra *intervalo* acrescentada ao ponto de expansão da regra *valor*.

O *Kotlin* tem tipos para representar estes intervalos. Por exemplo, um intervalo de valores inteiros é do tipo `IntRange` ou `LongRange` e um intervalo de símbolos é do tipo `CharRange`.

A função `toString()` é aplicável a um intervalo para obter a sua descrição textual.

O operador `!in` avalia se um elemento não pertence ao intervalo.

```

>>> val grade = (3-2)..5
>>> grade
res1: IntRange = 1..5
>>> 12 !in grade
res2: Boolean = true
>>> grade.toString()
res3: String = 1..5
  
```

Também é possível definir intervalos de valores comparáveis não enumeráveis, mas não têm as mesmas características. Por exemplo, valores dos tipos `Float`, `Double` e `String`, são comparáveis mas não enumeráveis, porque não é possível enumerar todos os valores entre um determinado valor mínimo e um valor máximo.

Estes intervalos não podem ser definidos com `until`, `downTo` ou `step`, não é possível aplicar a função `count()` e em vez das propriedades `first` e `last` têm as propriedades `start` e `endInclusive`.

```

>>> val values = 0.0 ..(20.0)
>>> 15.5 in values
res1: Boolean = true
>>> values.start
res2: Double = 0.0
>>> val words = "abc".. "abx"
>>> "abdaa" in words // "abdaa" > "abc" && "abdaa" < "abx"
res3: Boolean = true
>>> "abz" in words // "abz" > "abx"
res4: Boolean = false
>>> words.endInclusive
res5: String = abx
>>> words.count()
error: unresolved reference: count()
  
```

Os operadores `in` e `!in` também podem ser utilizados para verificar se um símbolo ou uma *string* está contido numa *string*.

```

>>> 'o' in "Kotlin"
res1: Boolean = true
>>> 'K' !in "Kotlin"
res2: Boolean = false
>>> "otl" in "Kotlin"
res3: Boolean = true
>>> "ota" in "Kotlin"
res4: Boolean = false
  
```

2.7 Sintaxe das expressões

Usando os tipos e os operadores apresentados neste capítulo, os diagramas sintáticos *expressão* e *valor* apresentados anteriormente na figura 2.2 são atualizados com os apresentados na figura 2.12.

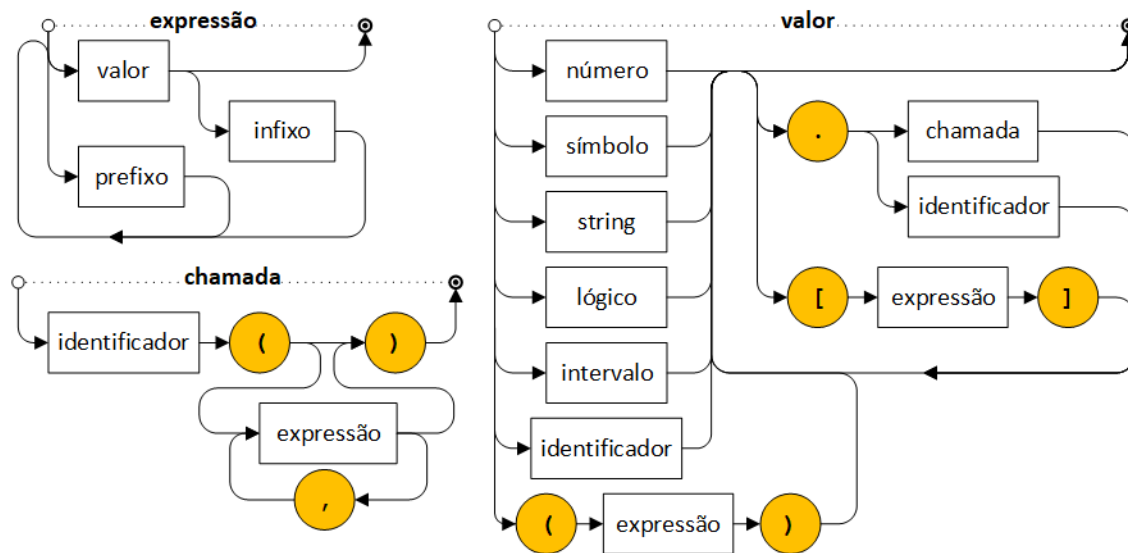


Figura 2.12: Diagramas sintáticos atualizados

A regra *prefixo* refere-se aos operadores unários que são utilizados a prefixar o valor, como são o caso dos operadores `!` (negação lógica), `-` (valor negativo) e `+` (valor positivo).

A regra *infixo* refere-se aos operadores com argumento esquerdo e direito, como é o caso dos operadores aritméticos (`+`, `-`, `*`, `/` e `%`), dos operadores de comparação (`==`, `!=`, `<`, `>`, `<=` e `>=`), dos operadores lógicos (`&&` e `||`) e os operadores de pertença (`in` e `!in`).

```
>>> val str = "abc"
>>> ("xy" + (-3 * 2.toFloat()) + str[1]).length > 5 && c in 'a'..'z'
res1: Boolean = true
```

A expressão apresentada neste exemplo é reconhecida pelas regras sintáticas já apresentadas, como demonstra a árvore de expansão da figura 2.13.

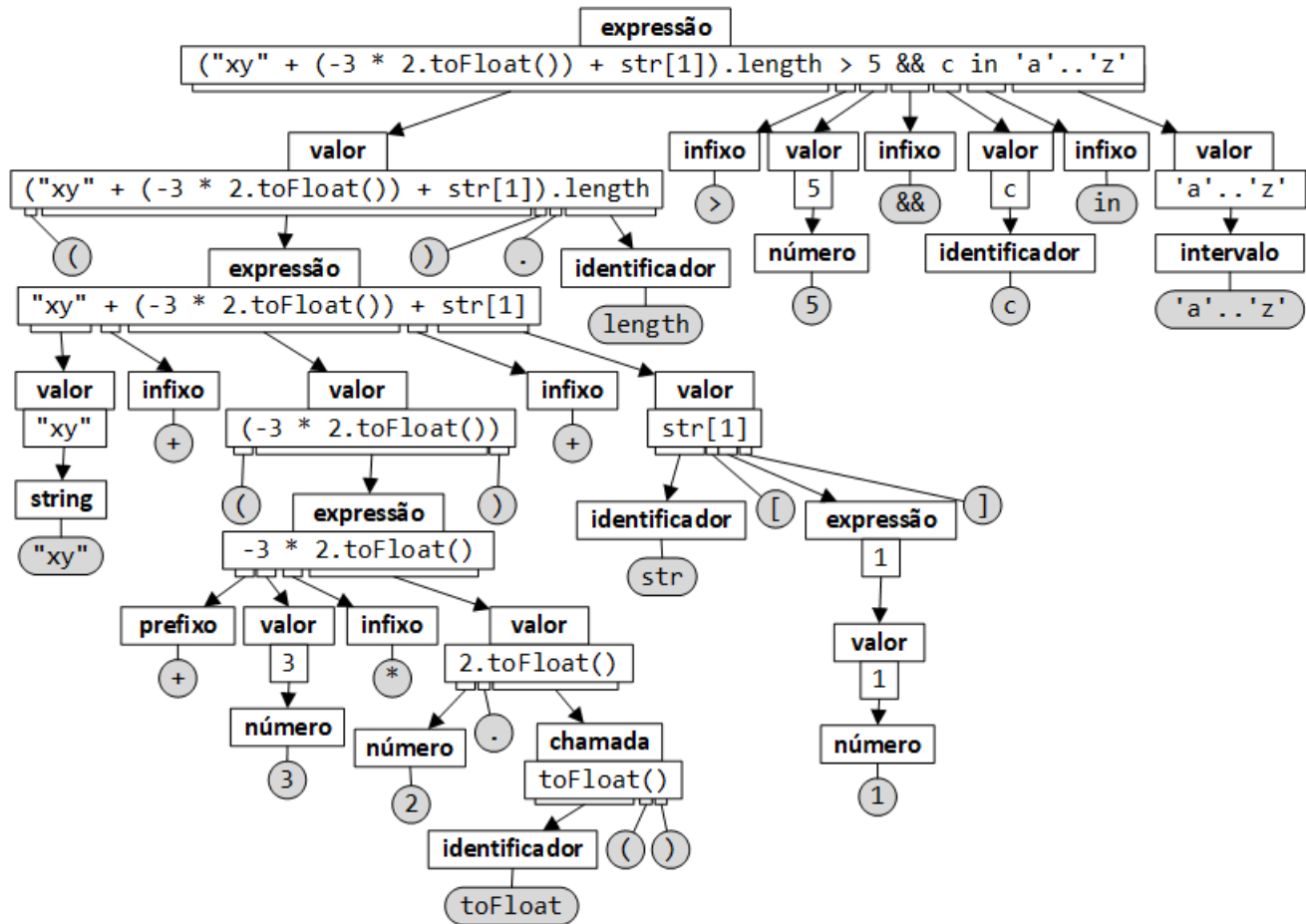


Figura 2.13: Árvore de expansão das regras para uma expressão

Capítulo 3

Programas, leituras e escritas

Tipicamente, um programa obtém informação a partir de leituras ou de estímulos externos, processa essa informação e apresenta resultados.

Os primeiros programas que iremos construir, fazem leituras de dados de ficheiros ou introduzidos pelo utilizador através do teclado, realizam operações com esses dados e escrevem os resultados no ecrã ou em ficheiros.

Um programa em *Kotlin* é descrito em ficheiros de texto com extensão `.kt`, designados por ficheiros fonte, que são editados com um editor de texto.

Os ficheiros fonte declaram os elementos constituintes do programa que são, entre outros tipos de elementos, funções que produzem resultados.

Tipicamente, as declarações de funções são prefixadas com a palavra reservada `fun`, os parâmetros são descritos entre parêntesis curvos (...) e o bloco das instruções que a função executa, designado como corpo da função, é descrito entre chavetas {...}.

3.1 A função *main*

A função com o nome `main` é o ponto de entrada de um programa, ou seja, é automaticamente chamada quando se executa um programa. Ao contrário das palavras `val` e `fun`, `main` não é uma palavra reservada da linguagem, apenas é o nome da função que se convencionou como ponto de entrada do programa, entre muitas outras funções que o programa poderá ter.

O programa da listagem 3.1 declara apenas a função identificada por `main`.

Neste caso, a função `main()` não tem parâmetros e tem duas instruções no seu corpo: Uma declaração do valor identificado por `name`, do tipo `String`, e uma chamada à função `println()` que passa como parâmetro o texto a escrever. O texto passado como parâmetro é um valor do tipo `String` com uma expressão embutida que usa o valor identificado por `name`.

O texto desde `//` até ao fim da linha é um comentário que tem significado para quem lê o programa, mas é ignorado pelo compilador.

Ao contrário de várias linguagens de programação, não é obrigatório colocar um ponto e vírgula (;) no final de cada instrução em *Kotlin*. Isso é apenas necessário para separar várias instruções na mesma linha, o que é invulgar em *Kotlin*.

```
fun main() {  
    val name :String = "Kotlin"  
    println("Hello $name!") //String com expressão embutida  
}
```

Listagem 3.1: Greeting.kt

Usando o editor de texto, conforme foi descrito na secção 1.6.5, para editar e gravar o ficheiro `Greeting.kt`, e usando depois o compilador de *Kotlin*, indicando como argumento o nome do ficheiro fonte, é gerado o ficheiro `GreetingKt.class` que pode ser executado.

Para executar o programa é utilizado o comando `kotlin` indicando como argumento o nome do ficheiro gerado, sem extensão, que tem o programa já compilado com a função `main()`.

Os três comandos seguintes são os necessários para editar, compilar e executar o programa.

```
C:\ISEL\PG>notepad++ Greeting.kt ↵
C:\ISEL\PG>kotlinc Greeting.kt ↵
C:\ISEL\PG>kotlin GreetingKt ↵
Hello Kotlin!
```

3.2 Erros de compilação e erros de execução

É normal cometer erros quando se programa. Existem basicamente dois tipos de erros: Os que são detetados e reportados pelo compilador, porque não foi cumprida alguma regra sintática ou semântica da linguagem; Os que são detetados quando se executa o programa, porque o programa não tem o resultado esperado ou porque termina inesperadamente com uma situação excecional.

Por exemplo, vamos supor que o programa da listagem 3.1 tinha sido escrito da seguinte forma:

```
fun main() {
    val name :string = "Kotlin"
    println("Helo $name!")
}
```

Ao compilar este programa, o compilador reporta dois erros.

```
C:\ISEL\PG>kotlinc Greeting.kt ↵
Greeting.kt:2:30: error: expecting '''
    val name :string = "Kotlin"
                        ^
Greeting.kt:2:13: error: unresolved reference: string
    val name :string = "Kotlin"
                ^
```

O primeiro erro foi na coluna 30 da linha 2, porque foi usada uma plica, em vez de aspas, para terminar o texto da string.

O segundo erro foi na coluna 13 da linha 2, porque o nome do tipo deveria ser **String** com um S maiúsculo.

O primeiro erro de compilação é sintático. Não foi cumprida a regra sintática da escrita de literais do tipo **String**, que foi descrita no diagrama da figura 2.8.

O segundo erro de compilação é semântico. Não existe o tipo com o nome **string** com um s minúsculo. Mas até poderia existir.

Agora, seria necessário utilizar novamente o editor de texto e corrigir os dois erros reportados, ficando assim:

```
fun main() {
    val name :String = "Kotlin"
    println("Helo $name!")
}
```

Depois de gravar novamente o ficheiro e compilar novamente o programa, já não daria erros de compilação.

```
C:\ISEL\PG>kotlinc Greeting.kt ↵
C:\ISEL\PG>kotlin GreetingKt ↵
Hello Kotlin!
```

Mas ao executar o programa constata-se que o texto escrito não era o esperado, porque a palavra **Helo** deveria ter dois l's.

Este erro é de execução, porque o resultado produzido não foi o esperado. Portanto, é necessário usar novamente o editor, corrigir o erro, gravar novamente o ficheiro, compilar o programa e, depois de verificar que não havia erros de compilação, executar novamente o programa.

Finalmente, como não havia erros de compilação nem erros de execução, poderíamos considerar que o desenvolvimento do programa tinha terminado.

No entanto, é muito natural, provocar novos erros de compilação quando se corrige erros de execução, sendo necessário repetir todos os passos anteriores.

O desenvolvimento de programas é um processo trabalhoso. Mesmo os programadores experientes, provocam muitos erros inadvertidamente, mas é recompensador quando finalmente se chega ao resultado pretendido.

3.3 Escrita na consola

```
println("Hello")
```

Designa-se por consola ao conjunto formado pelo teclado e o ecrã, onde os programas fazem normalmente as leituras e as escritas de informação, para interagir com o utilizador.

Para escrever uma linha de texto na consola é chamada a função `println()`. Tal como `main()`, `println()` é também uma função, mas esta, já está implementada e faz parte da biblioteca de *Kotlin*.

A instrução `println("Hello")` chama a função `println()`, que executará as instruções do seu corpo. Para chamar uma função indica-se o seu nome seguido dos argumentos entre parêntesis.

A função `println()` escreve textualmente o valor do argumento a partir da posição corrente do cursor. Depois, coloca o cursor no início da linha seguinte, para que qualquer texto escrito posteriormente seja apresentado na próxima linha.

A função `print()` também escreve textualmente o argumento que for passado como parâmetro, mas deixa o cursor no final do que foi escrito, sem mudar de linha.

A função `main()` do programa da listagem 3.2 executa três instruções que chamam a função `print()` e `println()`, com o objetivo de escrever `Hello World!` na primeira linha e `Bye.` na segunda linha.

```
fun main() {
    print("Hello")           //Cursor fica no final do texto escrito.
    println(" World!")      //Cursor muda para a linha seguinte.
    println("Bye.")         //Escrita na segunda linha.
}
/* Escreve:
Hello World!
Bye.
*/
```

Listagem 3.2: Hello.kt

O texto entre `/*` e `*/` é ignorado pelo compilador. Neste caso, é um comentário para mostrar qual é o resultado pretendido pelo programa.

É óbvio, que as duas primeiras chamadas poderiam ser substituídas por uma só:

```
println("Hello World!")
```

Apesar de não ficar mais legível e de não ser habitual escrever desta forma, é possível fazer as duas chamadas na mesma linha, terminando a primeira com ponto e vírgula.

```
println("Hello World!"); println("Bye.")
```

E também é possível substituir as três chamadas por uma só, explicitando o símbolo `\n`, que faz a mudança de linha:

```
println("Hello World!\nBye.")
```

As funções `print()` e `println()` escrevem textualmente qualquer tipo de valor, por isso o parâmetro recebido pode ser de um tipo qualquer.

Chamar `println()` sem parâmetros, apenas faz deslocar o cursor para o início da linha seguinte.

```
fun main() {
    val fractional = 1.3
    val lastLetter = 'Z'
    println(27)           // -> 27
    println()             //Escreve linha vazia
    println(2 * fractional) // -> 2.6
}
```

```

    print(lastLetter)          // -> Z (sem mudar de linha)
    println(lastLetter=='z')  // -> false
}
/* Escreve:
27

2.6
Zfalse
*/

```

Listagem 3.3: PrintVals.kt

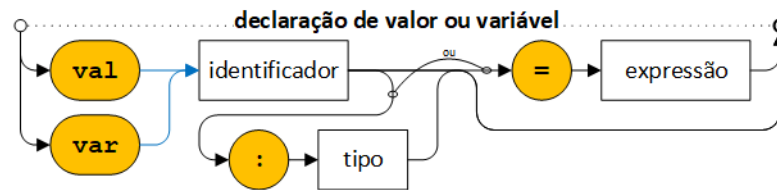
3.4 Utilização de variáveis

```
var number :Int = 0
```

Tal como foi descrito na secção 2.2 podemos atribuir um nome a um valor, para nos referirmos a esse valor noutros locais do programa. Mas o valor não pode ser alterado.

É possível declarar variáveis, em vez de valores, prefixando a declaração com a palavra reservada **var**, em vez de **val**. Neste caso, o valor pode ser alterado, ou seja, o conteúdo da variável pode ser modificado.

A figura 3.1 apresenta o diagrama sintático da declaração de valores ou variáveis.

Figura 3.1: Diagrama sintático da declaração de **val** ou **var**

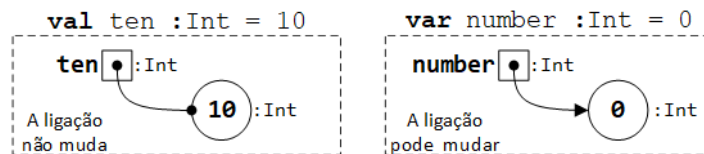
As setas azuis do diagrama indicam que para separar a palavra reservada **val** ou **var** do identificador é necessário colocar pelo menos um símbolo separador (por exemplo um espaço).

A anotação com *ou*, significa que tem que ser indicado pelo menos o tipo *ou* o valor inicial.

Quando o tipo é omitido é assumido o tipo do valor que resulta da expressão.

Se o valor inicial é omitido, ele terá que indicado mais adiante, antes da primeira utilização, sendo indicado apenas uma vez se for a declaração de um valor.

A figura 3.2 é uma possível representação que mostra a diferença entre um valor, com nome atribuído, e uma variável que refere um valor.

Figura 3.2: Representação de **val** e **var**

As variáveis (**var**) são necessárias quando a informação muda ao longo de um programa. Quando a informação não é alterada é sempre preferível usar valores (**val**).

3.4.1 Afetação de variáveis

```
number = 27
```

Para alterar o valor referido pelas variáveis é usada a instrução de afetação (*assign*).

À variável (**var**) com o identificador indicado à esquerda do operador **=** é atribuído o valor da expressão indicada à direita. Não é possível alterar o valor armazenado declarado com **val**.

O tipo do valor da expressão tem que ser igual ao tipo da variável. Esta compatibilidade dos tipos é verificada pelo compilador, no entanto, tal como nas declarações, se a expressão só envolve literais o tipo do valor pode ser adaptado ao tipo da variável.

```
fun main() {
    var number = 0    // Inferido o tipo Int
    println(number)  // -> 0
    number = 27      // Afetação
    println(number)  // -> 27
}
```

Listagem 3.4: Vars.kt

O programa da listagem 3.4 declara uma variável iniciada com o valor 0 e depois afeta-a com o valor 27.

Tal como é mostrado na figura 3.3, a afetação altera a variável para que fique a referir outro valor.

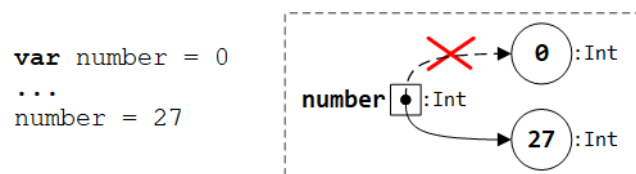


Figura 3.3: Representação de uma afetação

O programa da listagem 3.5 faz uma afetação para somar o valor 5 à variável **symb**. Por este motivo, a expressão à direita do operador **=** usa o identificador da própria variável afetada como argumento esquerdo do operador **+**.

```
fun main() {
    var symb = 'A'    // O tipo inferido é Char
    print("$symb ")
    symb = symb + 5    // Afeta com o valor que tinha mais 5
    println("+ 5 = $symb")
}
/* Escreve:
A + 5 = F
*/
```

Listagem 3.5: AddChar.kt

Como este tipo de afetações é muito frequente, existe também o operador de afetação composta com a soma **+=**. Usando este operador, a instrução **symb = symb + 5** pode ser substituída por:

```
symb += 5 //Equivalente a: symb = symb + 5
```

Assim como existe o operador **+=** existem os restantes operadores de afetação composta com as operações aritméticas. Usando todos os operadores referidos, a instrução de afetação tem a sintaxe apresentada na figura 3.4.

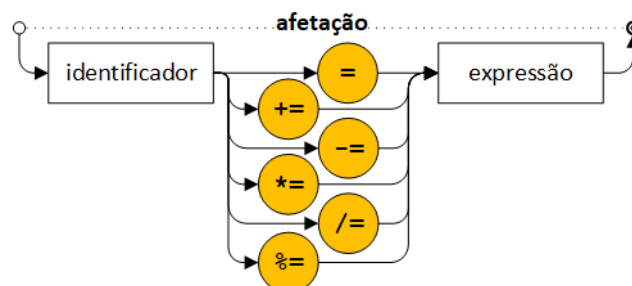


Figura 3.4: Diagrama sintático da afetação

3.5 Leitura da consola

```
line = readln()
```

Para ler uma linha de texto da consola é chamada a função `readln()`. Esta função também faz parte da biblioteca de *Kotlin*.

A função `readln()` não tem parâmetros e retorna um valor do tipo `String`. Quando esta função é chamada são lidos todos símbolos introduzidos através do teclado até ser premida a tecla `↵`.

O programa da listagem 3.6 declara uma variável que refere o nome introduzido pelo utilizador; Escreve a pergunta na consola deixando o cursor um espaço depois do ponto de interrogação; Lê a linha de texto introduzida pelo utilizador; Escreve a mensagem "Olá" seguida pelo texto introduzido.

```
fun main() {
    var name :String           // Para armazenar o nome
    print("Como se chama? ")   // Faz a pergunta
    name = readln()            // Lê o texto com o nome
    println("Olá $name")
}
```

Listagem 3.6: YourName.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin YourNameKt ↵
Como se chama? Rui Santos ↵
Olá Rui Santos
```

De facto, o programa `YourName.kt` por ser reformulado para usar um valor em vez da variável, dado que esta só é alterada uma vez. Além disso, a declaração do valor pode ser realizada em simultâneo com a afetação, ficando uma declaração iniciada. Assim, as instruções da função `main()` seriam:

```
print("Como se chama? ")   // Faz a pergunta
val name = readln()        // Lê o texto com o nome
println("Olá $name")
```

3.5.1 Leitura de números

```
x = readln().toInt()
```

Para ler valores inteiros ou outro tipo de valores que não sejam do tipo `String` é necessário ler uma linha e converter a `String` para o tipo pretendido, usando uma das funções de conversão apresentadas na secção 2.4.2

O programa da listagem 3.7 lê o valor do raio de um círculo, assumindo que é introduzido um valor inteiro, e apresenta a área do círculo num valor real.

```
fun main() {
    print("Raio do círculo? ")
    val line = readln()
    val radius = line.toInt()           // Conversão String->Int
    val area = radius * radius * 3.14159 // A = R^2 * PI
    println("Área = $area")
}
```


Listagem 3.7: CircleArea.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin CircleAreaKt ↵
Raio do círculo? 2.5 ↵
Área = 19.6349375
```

O mesmo programa poderia ser escrito com 3 instruções apenas, mantendo a mesma legibilidade, fazendo a leitura e a conversão na mesma expressão e usando uma expressão embutida no texto a escrever que faz o cálculo da área:

```
print("Raio do círculo? ")
val radius = readln().toInt() // Lê um valor inteiro do tipo Int
println("Área = ${radius * radius * 3.14159}") // A = R^2 * PI
```

Se o utilizador escrever espaços à esquerda ou à direita do valor introduzido e depois premir , o programa dará um erro. Este comportamento não é importante, se assumirmos que o utilizador escreve sem espaços.

Se for importante garantir que o programa lê corretamente o valor quando o utilizador escreve espaços, basta aplicar a função `trim()` à linha lida antes de converter para o valor pretendido. A função `trim()` aplicada a uma `String` retorna outra `String` sem os espaços do início e do fim.

```
val line = readln().trim() // Lê linha e retira espaços à volta
val radius = line.toInt() // Converte para valor inteiro do tipo Int
```

3.5.2 Leitura de símbolos

```
c = readln()[0]
```

Para ler um símbolo, recolhe-se o primeiro `Char` da `String` lida, usando o operador `[]`, tal como foi apresentado na secção 2.4.2, fazendo:



```
val line = readln() // Lê uma linha
val symb : Char = line[0] // Obter o primeiro símbolo da linha
```

O programa da listagem 3.8 lê o primeiro símbolo da linha introduzida pelo utilizador, ignorando os espaços, e apresenta o símbolo e o respetivo código.

```
fun main() {
    print("Símbolo? ")
    val char = readln().trim()[0] // O tipo Char é inferido
    val code = char.code
    println("O símbolo $char tem o código $code.")
}
```

Listagem 3.8: CharCode.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin CharCodeKt 
Símbolo? Fax 
O símbolo F tem o código 70.
```

Neste exemplo de utilização, quando é avaliada a expressão `readln().trim()[0]`, são realizadas as seguintes avaliações parciais:

`readln().trim()[0] ⇒ " Fax ".trim()[0] ⇒ "Fax"[0] ⇒ 'F'`

em que é chamada em primeiro lugar a função `readln()`, que retorna a *string* introduzida pelo utilizador " Fax ", com os espaços introduzidos. Depois é chamada a função `trim()` aplicada a essa *string*, que retorna a *string* "Fax", já sem espaços. Finalmente, é aplicado o operador `[]` com o índice zero à *string* "Fax", resultando o valor 'F' do tipo `Char`, que é o primeiro símbolo da *string*.

3.6 Leitura e escrita standard

De facto, a função `readln()` lê linhas de texto do *standard input* e as funções `print()` e `println()` escrevem texto no *standard output*.

Por omissão, quando é executado um programa, o sistema operativo assume que o *standard input* é o teclado e o *standard output* é o ecrã. Mas não é sempre assim.

Tomemos como exemplo o programa da listagem 3.9, que escreve no *standard output* a soma dos dois valores inteiros lidos do *standard input*, um em cada linha.

Propositadamente, este programa não escreve nada em forma de pergunta, porque admite que não haverá interação com o utilizador e que deve apenas escrever o resultado produzido.

```
fun main() {
    val a = readln().toInt()
    val b = readln().toInt()
    println("$a + $b = ${a+b}")
}
```

}

Listagem 3.9: SumValues.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin SumValuesKt ↵
127 ↵
95 ↵
127 + 95 = 222
```

3.6.1 Redirecionamento do input

command < file

Podemos executar este programa indicando ao sistema operativo que as leituras são realizadas de um ficheiro de texto, em vez do teclado.

Para tal, é necessário acrescentar no final do comando o símbolo < seguido do nome do ficheiro que será lido.

Assumindo que existe um ficheiro com o nome `values.txt` no diretório corrente, que foi editado com o editor de texto, para ter como conteúdo o texto:

```
128
512
```

Podemos executar este programa com o seguinte comando no sistema operativo, que indica que o input é lido do ficheiro `values.txt`:

```
C:\ISEL\PG>kotlin SumValuesKt < values.txt ↵
128 + 512 = 640
```

Desta forma, o programa escreveu o no ecrã o resultado, mas leu os valores do ficheiro `values.txt`.

Caso o ficheiro indicado não exista, é reportado um erro pelo sistema operativo e o programa não é executado.

Caso o ficheiro não tenha pelo menos duas linhas de texto com os valores inteiros esperados, o programa termina com uma exceção.

3.6.2 Redirecionamento do output

command > file

Também é possível executar o programa indicando ao sistema operativo que as escritas são realizadas num ficheiro, em vez de serem feitas no ecrã.

Para tal, acrescenta-se no final do comando o símbolo > seguido do nome do ficheiro que será escrito.

Caso o ficheiro não exista, será criado pelo sistema. Caso já exista será reescrito.

Assim, podemos executar este programa com o seguinte comando no sistema operativo, que indica que o output é escrito no ficheiro `result.txt`:

```
C:\ISEL\PG>kotlin SumValuesKt > result.txt ↵
1024 ↵
512 ↵
```

O conteúdo do ficheiro escrito pode ser observado usando o editor de texto, ou usando o comando `type` do *Windows* ou `cat` do *Linux*:

```
C:\ISEL\PG>type result.txt ↵
1024 + 512 = 1536
```

É possível redirecionar o output para um ficheiro acrescentando ao conteúdo já existente, usando >> seguido do nome do ficheiro onde será acrescentado o texto.

O seguinte comando executa o programa redirecionando o input para `values.txt` e o output para `result.txt`, acrescentando no que já existia:

```
C:\ISEL\PG>kotlin SumValuesKt < values.txt >> result.txt ↵
```

Usando o comando para mostrar o conteúdo do ficheiro `result.txt`, dará:

```
C:\ISEL\PG>type result.txt
1024 + 512 = 1536
128 + 512 = 640
```

3.7 Instruções e expressões

Expressões e instruções são elementos básicos de qualquer linguagem de programação. As instruções executam-se e as expressões avaliam-se.

Expressões produzem resultados quando são avaliadas. O resultado será um valor de um tipo bem determinado.

Por exemplo, a expressão `2.5f+2` quando avaliada produz o valor `4.5` do tipo `Float`.

Instruções produzem efeitos quando são executadas, mas não resultados. Para ser útil, o efeito deve alterar o estado de alguma coisa.

Por exemplo, a afetação `x = 2.5f+2` quando executada altera o valor da variável `x` para o valor `4.5`, mas a afetação não produz um valor.

A afetação é uma instrução e não uma expressão. Por exemplo, não é possível em *Kotlin* tentar fazer afetações simultâneas como: `y = (x = 4.5f)` porque `(x=4.5f)` não é uma expressão e não produz um valor.

As chamadas a funções também são expressões que produzem o valor retornado pela função chamada.

Por exemplo, `readln()` é uma expressão que produz um valor do tipo `String` que é o texto introduzido pelo utilizador. A expressão `"abc"+readln()` produz um valor do tipo `String` que resulta da concatenação de `"abc"` com o texto introduzido pelo utilizador.

Em *Kotlin* qualquer expressão também é uma instrução, mas o contrário não é verdade.

Uma expressão pode ser usada como instrução, mas nesse caso a execução dessa instrução avalia a expressão sem aproveitar o resultado final da expressão. Portanto, para ser útil, a avaliação dessa expressão deve produzir algum efeito.

Por exemplo, a instrução `x+5` quando executada realiza a soma entre o valor armazenado em `x` com o valor `5`, mas o resultado não é aproveitado. Este é um exemplo de uma instrução, que apesar de ser válida em *Kotlin*, não produz um efeito útil.

Sintaticamente, as declarações de valores e de variáveis são instruções.

Na maioria das instruções, que serão apresentadas nos próximos capítulos, onde é permitido colocar um bloco de instruções entre chavetas, também se pode colocar apenas uma instrução.

Nos restantes capítulos apresentam-se mais instruções, mas com o que foi descrito até ao final deste capítulo, os diagramas sintáticos da figura 3.5 descrevem o que é considerado instrução.

As regras *instruções* e *bloco* serão usadas nos próximos capítulos.

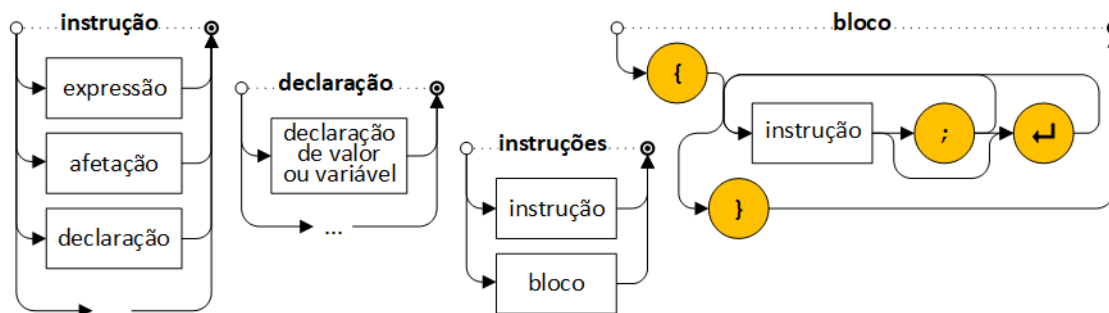


Figura 3.5: Diagramas sintáticos de instrução, declaração e bloco de instruções

Capítulo 4

Decisões

Para programar é necessário disciplinar o raciocínio para traduzir a resolução de um problema num algoritmo.

Um algoritmo descreve uma sequência lógica de operações cuja execução atinge um dado objetivo. Esta sequência está normalmente sujeita a critérios de decisão condicionados a valores que fazem parte do problema.

As operações de controlo de execução são fundamentais nas linguagens de programação. É através delas que podemos definir os critérios de decisão dos algoritmos.

As operações de controlo de execução podem ser divididas em três grupos:

- Decisões binárias das quais resulta a avaliação ou execução alternativa entre duas sequências de operações;
- Decisões múltiplas em que é avaliada ou executada uma das várias sequências de operações;
- Repetições condicionais das quais resulta a execução cíclica de uma sequência de operações.

Neste capítulo iremos abordar as decisões binárias e as decisões múltiplas ficando as repetições para mais tarde.

4.1 Decisão binária - *if*

```
if (value>0) println("Positivo.")
```

A versão mais simples da decisão binária verifica se uma condição é verdadeira ou falsa. Se a condição for verdadeira realiza uma ou mais operações.

Sintaticamente, é iniciada com a palavra reservada *if* seguida da expressão com a condição, entre parêntesis, e no final a instrução a executar, ou várias instruções entre chavetas (bloco de instruções).

O diagrama da figura 4.1 mostra a ordem de execução do *if*.

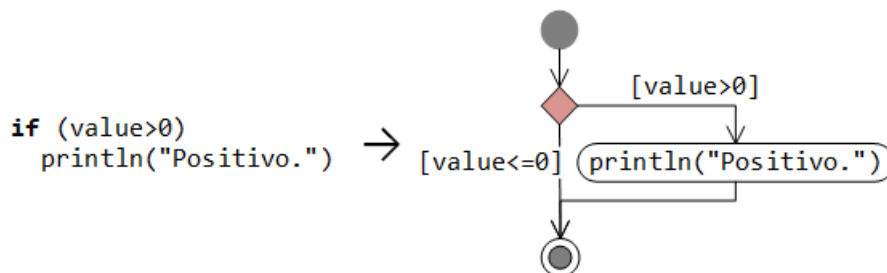


Figura 4.1: Diagrama de atividade da decisão binária

O programa apresentado na listagem 4.1 verifica, usando três decisões binárias, se o valor introduzido é positivo, é negativo e se é positivo e par.

```

fun main() {
    print("Valor? ")
    val value = readln().toInt() //Leitura do valor inteiro
    if (value > 0) println("Positivo.")
    if (value < 0)
        println("Negativo.")
    if (value > 0 && value % 2 == 0) {
        print("O valor positivo $value ")
        println("é par.")
    }
}

```

Listagem 4.1: Value1.kt

Uma possível utilização deste programa seria:

```

C:\ISEL\PG>kotlin Value1Kt ↵
Valor? 24 ↵
Positivo.
O valor positivo 24 é par.

```

O resultado da expressão entre parêntesis (a condição) tem que ser do tipo **Boolean**.

Caso seja só uma instrução a executar, esta poderá ficar na mesma linha da expressão ou na linha seguinte, mas devidamente indentada.

Caso sejam várias instruções a executar, cada instrução do bloco deve ficar em linhas distintas, devidamente indentadas. Também é possível colocar mais do que uma instrução na mesma linha desde que seja separadas por um ponto e vírgula, mas esta forma de escrever é invulgar em *Kotlin*.

As condições, ou partes da condição, podem ser avaliadas antes, armazenando o valor do tipo **Boolean**, principalmente se forem utilizadas em várias condições. Usando esta técnica, a parte do programa anterior depois da leitura do valor inteiro, poderia ser reformulado para:

```

val positive = value>0 //Avaliação prévia das duas condições
if (positive) println("Positivo.")
if (value<0)
    println("Negativo.")
if (positive && value%2 == 0)
    println("O valor positivo $value é par.")

```

As instruções a executar, se a condição do **if** for verdadeira, podem conter outro **if** (ifs aninhados). No exemplo anterior, o terceiro **if** pretende verificar se o valor é positivo e par, portanto poderia verificar apenas se é par dentro do bloco de instruções do primeiro **if** que já verificou se era positivo, ficando da seguinte forma:

```

val positive = value>0 // Esta avaliação prévia já não é necessária
if (positive) {
    println("Positivo.")
    if (value%2 == 0)
        println("O valor positivo $value é par.")
}
if (value<0)
    println("Negativo.")

```

4.1.1 Parte *else* da decisão binária

```
if (v>=0) print('+') else print('-')
```

A versão mais elaborada da decisão binária pode ter também as instruções que são executadas caso a condição seja falsa.

Sintaticamente, é adicionada a palavra reservada **else** seguida das instruções.

A sintaxe completa do **if** é a apresentada no diagrama da figura 4.2, não esquecendo que *instruções* pode ser apenas uma instrução ou um bloco de instruções entre chavetas, tal como foi descrito na figura 3.5.

O diagrama de atividade da figura 4.3 mostra o fluxo de execução das instruções do **if** com parte **else**.

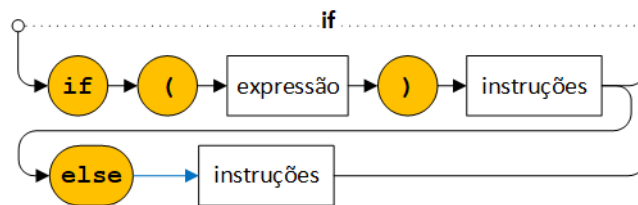
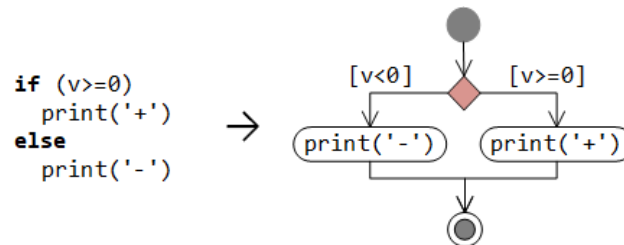


Figura 4.2: Diagrama sintático da decisão binária

Figura 4.3: Diagrama de atividade da decisão binária com *else*

O programa anterior pode ser melhorado, se a condição do segundo `if` só for verificada caso a condição do primeiro `if` seja falsa, colocando o segundo `if` na parte `else` do primeiro `if`:

```
if (value>0) {
    println("Positivo.")
    if (value%2 == 0)
        println("O valor positivo $value é par.")
}
else
    if (value<0)
        println("Negativo.")
```

Neste caso, o `if` aninhado não tem a parte `else`. O `else` pertence ao `if` exterior, sendo colocado depois do bloco de instruções que é executado quando a condição for verdadeira.

Quando se coloca um `if` na parte `else` do `if` anterior, é vulgar aparecer a palavra `else` seguida de `if` na mesma linha. Assim, as três últimas linhas podem ficar:

```
else if (value<0)
    println("Negativo.")
```

O programa da listagem 4.2 classifica o valor introduzido quanto a ser zero, positivo ou negativo.

```
fun main() {
    print("Valor? ")
    val value = readln().toInt()
    if (value==0)
        println("O valor é zero.")
    else if (value>0)
        println("$value é positivo.") //Instrução semelhante à seguinte
    else
        println("$value é negativo.") //Instrução semelhante à anterior
}
```

Listagem 4.2: Value2.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin Value2Kt ↵
Valor? 17 ↵
17 é positivo.
C:\ISEL\PG>kotlin Value2Kt ↵
Valor? 0 ↵
0 valor é zero.
```

Com o objetivo de realizar apenas uma chamada à função `println()` em vez das duas semelhantes que estão assinaladas, podemos usar uma variável do tipo `String` para ter o texto da parte que é

diferente e escrever esse texto com outra expressão embutida na string passada como parâmetro ao `println()`.

Assim, a parte `else` do primeiro `if` pode ficar da seguinte forma:

```
else {
    var signal = "positivo"           //Assume que é positivo
    if (value<0) signal = "negativo" //Muda o texto se não for
    println("$value é $signal.")
}
```

Nesta solução, `signal` é variável para poder ser alterada. É possível usar um valor (`val`), em vez de `var`, se for usado um `if` como expressão.

4.1.2 Expressão *if*

if (x>5) 10 else 0

A decisão binária do *Kotlin* pode ser usada como expressão ou como instrução, porque uma expressão também é uma instrução. Nos programas anteriores, todos os `ifs` foram usados como instruções.

A expressão `if` tem obrigatoriamente a parte `else` e ambas as partes são expressões. Quando avaliada, a expressão `if` produz um valor, se a condição for verdadeira, ou outro valor da expressão da parte `else`, se a condição for falsa. O valor produzido por ambas as expressões têm que ser do mesmo tipo.

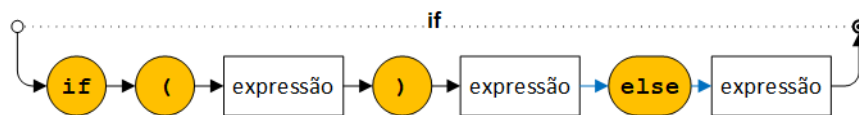


Figura 4.4: Diagrama sintático da decisão binária usada como expressão

O diagrama sintático do `if`, condicionado à utilização como expressão, fica como é descrito na figura 4.4, que é um caso particular do diagrama da figura 4.2.

Usando uma expressão `if`, a parte `else` do primeiro `if` do programa anterior pode ser reformulado para:

```
else {
    val signal = if (value>0) "positivo" else "negativo"
    println("$value é $signal.")
}
```

Ou então, usando uma expressão embutida em que a expressão é o próprio `if`:

```
else println("$value é ${if (value>0) "positivo" else "negativo"}.")
```

O programa da listagem 4.3 lê um símbolo introduzido pelo utilizador e apresenta o seu código em Unicode, indicando também se o símbolo é um dígito ou uma letra.

```
fun main() {
    print("Símbolo? ")
    val sym = readln()[0] // Lê um símbolo
    println("$sym -> Unicode=${sym.code}")
    val type =
        if (sym in '1'..'9') "dígito"
        else if (sym in 'A'..'Z' || sym in 'a'..'z') "letra" else ""
    if (type.length>0)
        println("$sym é $type")
}
```

Listagem 4.3: CharType.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin CharTypeKt ↵
Símbolo? h ↵
h -> Unicode=104
h é letra
```

Este programa usa uma expressão `if` na parte `else` de outra expressão `if`. As condições usam intervalos e o operador `in`, já apresentados na secção 2.6. O último `if` verifica se é necessário apresentar o tipo do símbolo, testando o comprimento da string `type`.

A representação dos valores envolvidos na execução do exemplo de utilização do programa `CharType` é apresentado na figura 4.5.



Figura 4.5: Representação dos valores em `CharType`

4.2 Decisão múltipla - *when*

```
when(expressão) { casos }
```

A decisão múltipla é suportada pelo `when`, que tal como o `if` também pode ser usada como instrução ou expressão.

A instrução `when` compara o valor de uma expressão com um conjunto de várias hipóteses e executa as instruções a que tem correspondência. Poderá ainda ter uma hipótese que é executada se não houver nenhuma das correspondências.

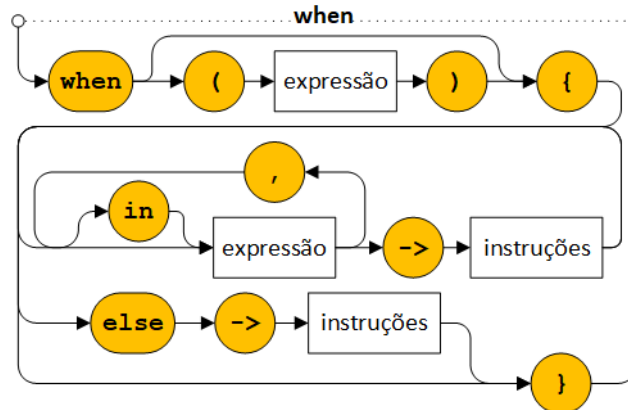


Figura 4.6: Diagrama sintático da decisão múltipla

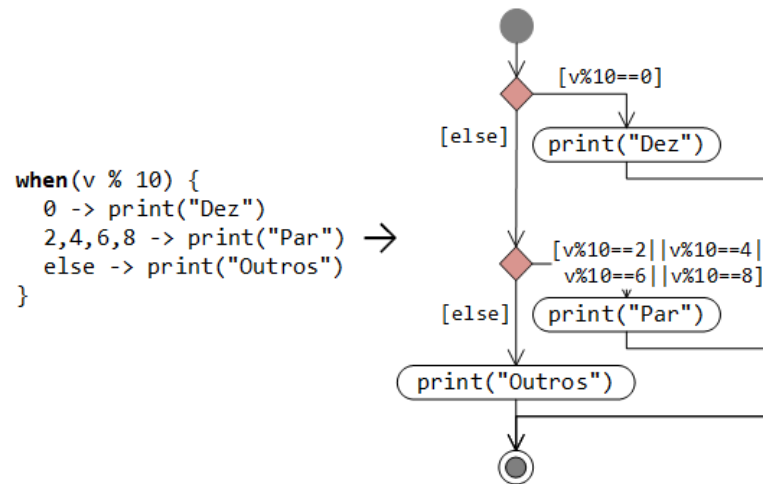
Sintaticamente, começa com a palavra reservada `when`, seguida da expressão com o valor a comparar, entre parêntesis, seguida do corpo com o conjunto de possibilidades entre chavetas.

Cada possibilidade tem uma expressão de seleção, ou várias separadas por vírgula, seguida dos símbolos menos e maior (`->`), seguidos das instruções e executar ou da expressão a avaliar.

Opcionalmente, pode ter uma última possibilidade com a palavra `else` em vez da expressão de seleção.

O diagrama da figura 4.6 tem a regra sintática do `when`, com algumas variações que serão explicadas mais adiante.

O diagrama de atividade da figura 4.7 mostra a execução de uma instrução `when`.

Figura 4.7: Diagrama de atividade do `when`

Para demonstrar a utilização do `when`, vamos fazer uma primeira versão de um programa sem usar o `when`.

O programa da listagem 4.4 lê o mês introduzido pelo utilizador e, caso seja o mês 2 (fevereiro), lê também o ano. Depois, apresenta o número de dias que tem o mês indicado. É necessário o ler o ano, no caso do mês 2, para saber se o ano é bissexto. Os anos bissextos são múltiplos de 4 mas não de 100, ou então são múltiplos de 400.

```

fun main() {
    print("Mês (1..12)? ")
    val m = readln().trim().toInt()
    val days :Int
    if (m==2) {
        print("Ano? ")
        val y = readln().trim().toInt()
        days = if (y%4==0 && y%100!=0 || y%400==0) 29 else 28
    }
    else if (m==4 || m==6 || m==9 || m==11)
        days = 30
    else
        days = 31
    println("Esse mês tem $days dias.")
}
  
```

Listagem 4.4: DayOfMonth.kt

Possíveis utilizações deste programa seriam:

```

C:\ISEL\PG>kotlin DayOfMonthKt ↵
Mês (1..12)? 4 ↵
Esse mês tem 30 dias.
C:\ISEL\PG>kotlin DayOfMonthKt ↵
Mês (1..12)? 2 ↵
Ano? 2020 ↵
Esse mês tem 29 dias.
  
```

Usando o `when` como instrução os dois ifs encadeados podem ser substituídos por um `when`.

```

val days :Int
when(m) {
    2 -> {
        print("Ano? ")
        val y = readln().trim().toInt()
        days = if (y%4==0 && y%100!=0 || y%400==0) 29 else 28
    }
    4,6,9,11 -> days = 30
    else -> days = 31
}
  
```

Usando um `when` como expressão esta poderá ser logo o valor inicial de `days`.

```
val days = when(m) {
    2 -> {
        print("Ano? ")
        val y = readln().trim().toInt()
        if (y%4==0 && y%100!=0 || y%400==0) 29 else 28
    }
    4,6,9,11 -> 30
    else -> 31
}
```

Quando o `when` é usado como expressão, tem que ter obrigatoriamente a hipótese `else` e as instruções de cada ramo são necessariamente expressões.

As expressões de seleção do `when` podem usar o operador `in` assumindo que o argumento esquerdo é o valor da expressão principal. Por exemplo, o programa seguinte faz a conversão de uma nota quantitativa para uma classificação qualitativa.

```
fun main() {
    print("Nota (0..20)? ")
    val grade = readln().trim().toInt()
    val classification = when(grade) {
        in 0..9 -> "Insuficiente"
        in 10..13 -> "Suficiente"
        in 14..16 -> "Bom"
        17,18 -> "Muito bom"
        19,20 -> "Excelente"
        else -> "ERROR"
    }
    println("$grade valores -> $classification")
}
```

Listagem 4.5: Grades.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin GradesKt ↵
Nota (0..20)? 15 ↵
15 valores -> Bom
```

A expressão principal do `when` pode ser omitida, se todas as expressões de seleção forem condições do tipo `Boolean`. O exemplo seguinte utiliza esta versão do `when`.

```
val txt :String = when {
    x > 0 -> "X positivo"
    Y == 0 -> "Y zero"
    else -> "outra condição"
}
```

4.3 Ambiente integrado *IntelliJ*

Os ambientes integrados de desenvolvimento de programas (*IDE - Integrated Development Environment*) permitem editar, compilar, executar e depurar programas, usando apenas uma interface com o utilizador.

O *IntelliJ IDEA* é um *IDE* desenvolvido pela *JetBrains*, que é a empresa que criou a linguagem *Kotlin*. O *IntelliJ* é utilizado para o desenvolvimento de projetos em várias linguagens de programação, entre as quais *Kotlin*.

Para instalar a versão livre (*Community*) mais recente do *IntelliJ IDEA* é necessário carregar o programa de instalação a partir do site www.jetbrains.com/idea/download/index.html para o sistema operativo pretendido e seguir os passos do sugeridos pelo programa.

Depois do *IntelliJ* instalado e executado deve seguir estes passos para desenvolver um programa em *Kotlin*:

Na janela de apresentação, escolha a opção "New Project" mas se estiver na janela principal, deve seleccionar na barra de menu "File" > "New" > "Project...".

Na janela "New Project" seleccionar "New Project". Em "Language:" seleccionar "Kotlin", em "Build system:" seleccionar "Gradle", em "JDK:" seleccionar a versão mais actual (por exemplo: versão 18), em "Gradle DSL:" seleccionar "Kotlin" e depois escreva o nome do projeto (neste caso **Cap4**), conforme mostra a figura 4.8, confirme que o local da pasta raiz do projeto é a pretendida (neste caso **C:\ISEL\PG**) e termine com o botão "Create".

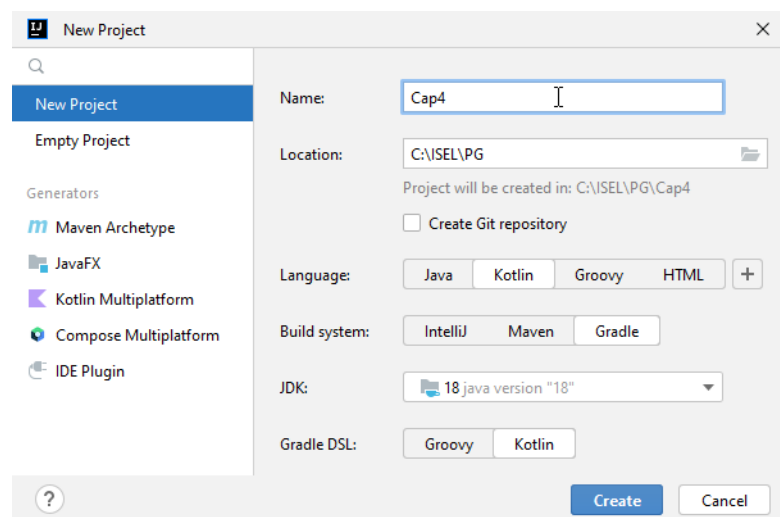


Figura 4.8: Janela para escolha do tipo e o nome do projeto

A janela principal tem três zonas de trabalho, assinaladas na figura 4.9: A zona inferior, para apresentação de resultados de execução e do terminal; A zona de edição (parte superior direita); A zona da estrutura do projeto (parte superior esquerda).

Os projetos ficam com a estrutura de pastas, apresentada na figura 4.9, em que a pasta "src" (*source*) contem os ficheiros fonte do projeto. Expanda a pasta "src", a sub-pasta "main" e sub-pasta "kotlin".

Caso já exista, o ficheiro "Main.kt", dentro da pasta "kotlin" e apresentado na zona de edição, este terá uma função `main` já implementada que não vamos utilizar. Portanto, remova esse ficheiro abrindo o menu de contexto nesse ficheiro e escolha a opção "Delete".

Para criar um ficheiro fonte, selecione a pasta "kotlin" e depois selecione na barra de menu: "File" > "New" > "Kotlin Class/File", ou em alternativa abra o menu de contexto na pasta "kotlin" e selecione: "New" > "Kotlin Class/File", como mostra a figura 4.10.

Escolha a opção "File" e insira o nome do ficheiro na janela "New Kotlin Class/File", neste caso, **DayOfMonth**, como mostra a figura 4.11.

Na zona de edição, aparece o conteúdo do ficheiro fonte **DayOfMonth.kt**, inicialmente vazio.

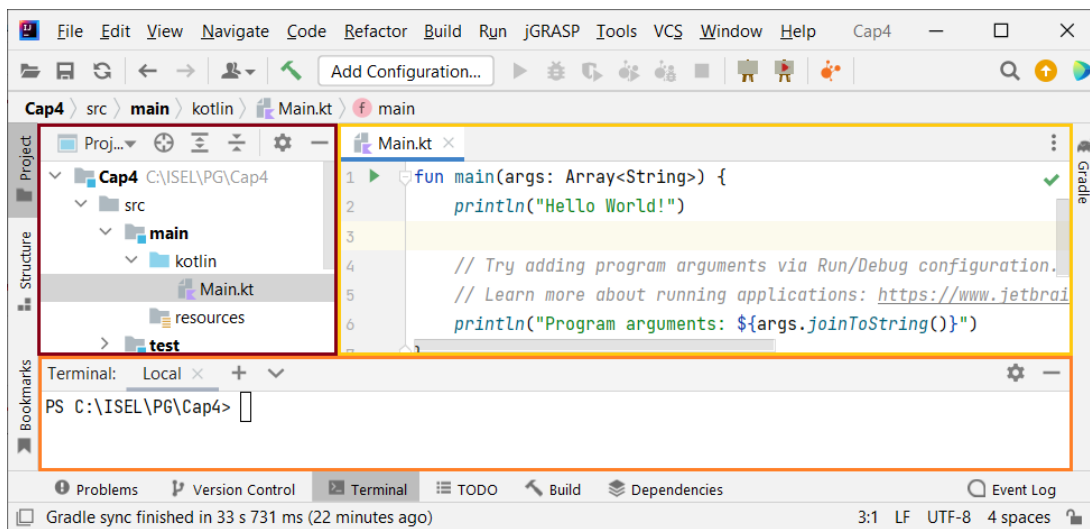


Figura 4.9: Janela principal do projeto

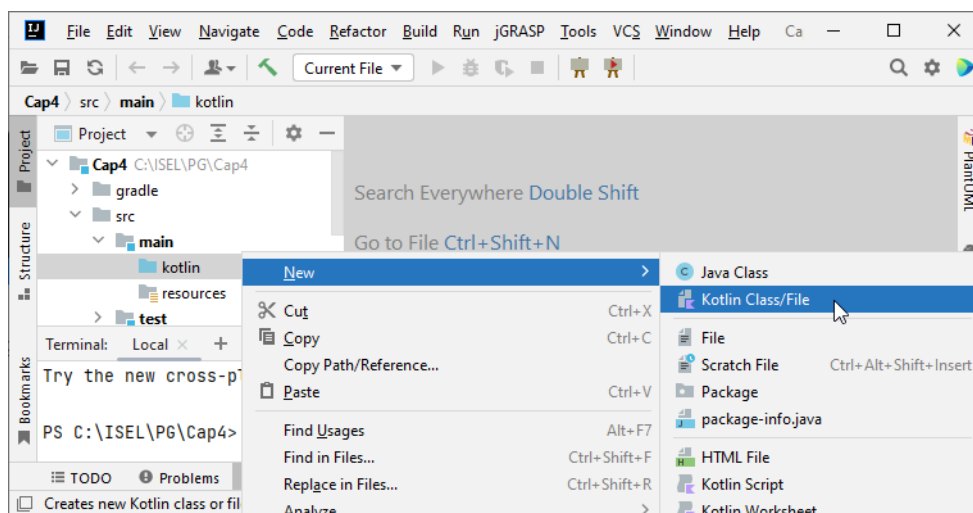
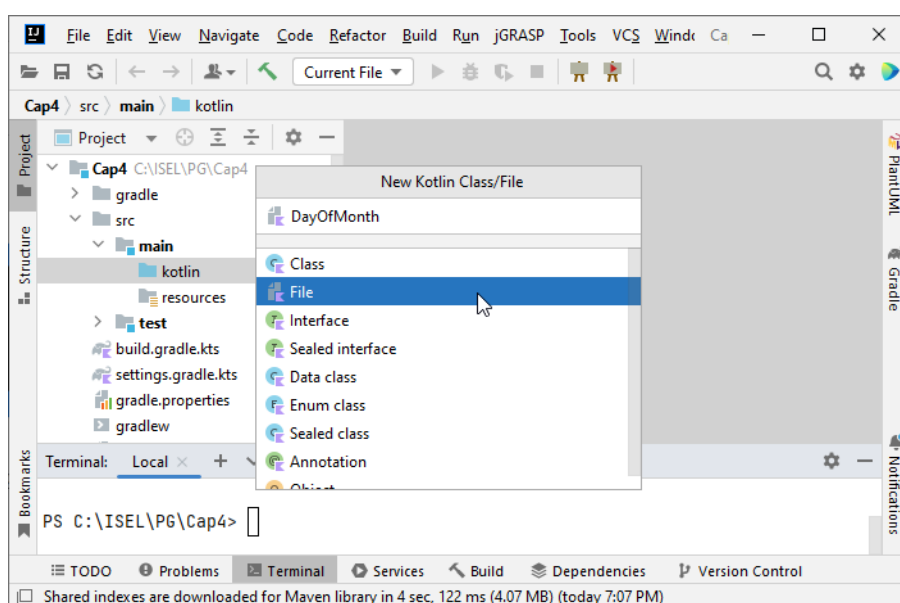


Figura 4.10: Criação do ficheiro fonte

Figura 4.11: Nome do ficheiro fonte *DayOfMonth.kt*

Escreva o código da versão final da listagem 4.4 (já com o `when` expressão) e experimente as ajudas na edição. Por exemplo, para acrescentar a função `main()` basta escrever `m` e premir `tab`.

No final, a zona de edição deve ficar como mostra a figura 4.12.

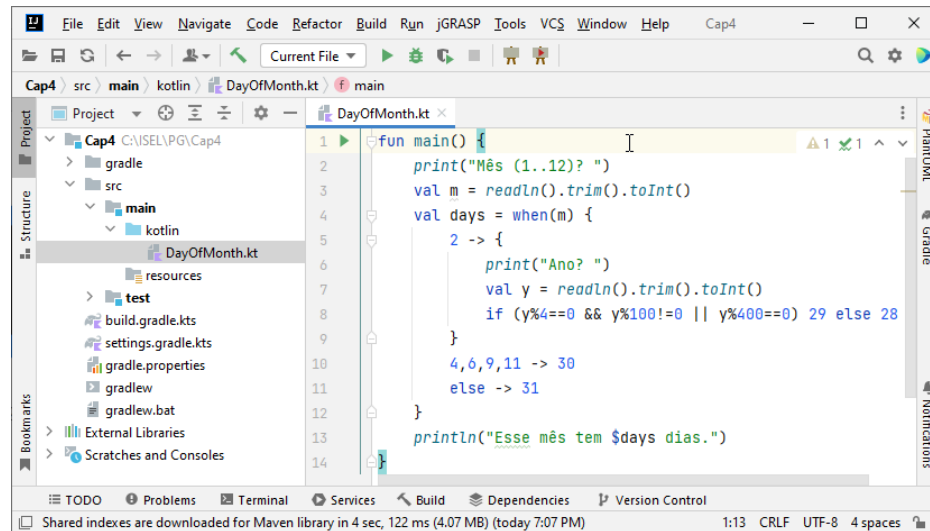



Figura 4.12: Edição do ficheiro *DayOfMonth.kt*

Para executar o programa, clicar em  à esquerda da função `main()` e seleccionar "Run 'DayOfMonthKt' ". Nesse momento, a zona inferior passa a apresentar o resultado da execução do programa, tal como mostra a figura 4.13.

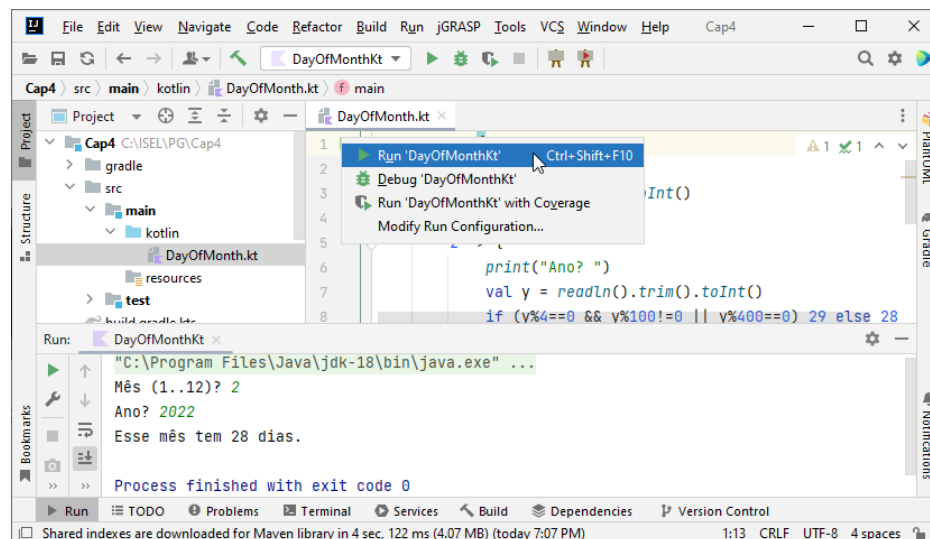




Figura 4.13: Execução de *DayOfMonthKt*

O texto a preto é o que foi escrito pelo programa (*output*) e o texto a verde em itálico é o introduzido pelo utilizador (*input*).

Não foi necessário gravar o ficheiro, porque é gravado automaticamente enquanto o ficheiro é editado. Também não foi necessário compilar o programa, porque a compilação é realizada antes da execução.

4.3.1 Utilização do *debugger*

Para fazer depuração do programa, em primeiro lugar acrescente um ponto de paragem (*break-point*). Para tal, clicar na barra cinzenta no lado esquerdo da zona de edição, na linha da instrução `if (y%4...` até aparecer .

Depois, clicar em  à esquerda da função `main()` e selecionar "Debug 'DayOfMonthKt'". Depois de introduzir os dois valores lidos pelo programa, este fica parado no *breakpoint* e na zona inferior aparece a informação de *Debug*, onde podemos observar os valores de `m` e `y` nesse momento, tal como mostra a figura 4.14.

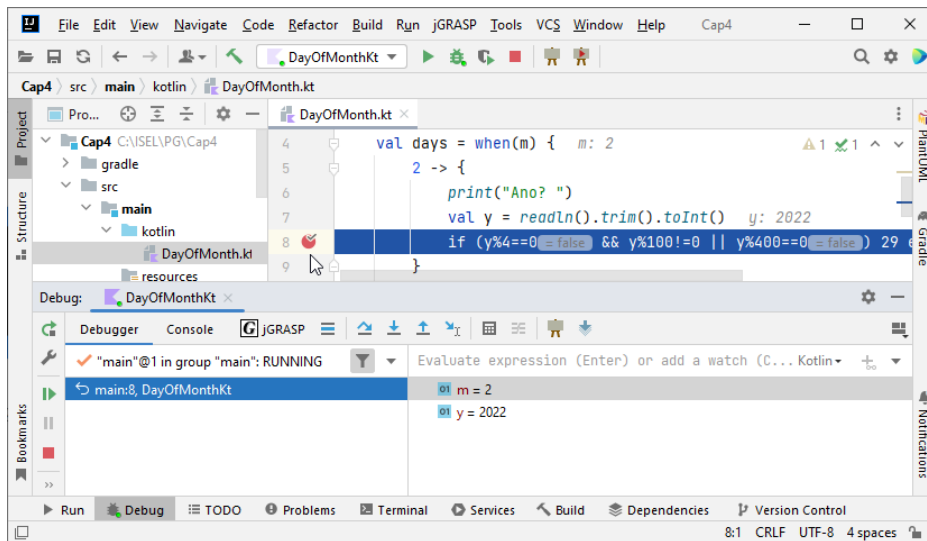



Figura 4.14: Depuração de *DayOfMonthKt*

Para executar apenas uma instrução, clicar em .

Para continuar a execução até ao fim do programa ou parar num *breakpoint*, clicar em .

Um projeto pode conter vários ficheiros fonte e cada um pode ter uma função `main()`. Para criar outro fonte, selecione a pasta "kotlin" abra o menu de contexto e selecione: "New" > "Kotlin Class/File".

Usando o ambiente integrado, também se pode usar o *REPL*, selecionando "Tools" > "Kotlin" > "Kotlin REPL" ou então criando um ficheiro de rascunho (*scratch file*), selecionando "File" > "New" > "Scratch File" > "Kotlin".

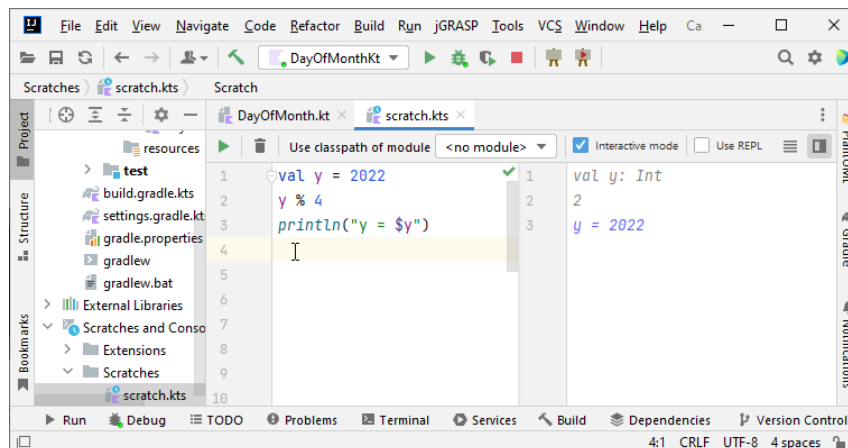


Figura 4.15: Utilização do *Scratch*

A figura 4.15 mostra a utilização do *scratch file* onde cada linha escrita é automaticamente avaliada e o resultado é apresentado no lado direito.

Capítulo 5

Ciclos

Para executar ciclicamente operações são usadas repetições condicionais, também designadas por ciclos.

Dentro do ciclo, as operações são executadas repetidamente enquanto uma condição for verdadeira, podendo esta condição ser avaliada depois ou antes de cada execução.

5.1 Repetição com condição final - *do-while*

```
do { código } while( condição )
```

A instrução **do-while** executa uma ou mais vezes o bloco de código, verificando a condição depois de cada execução.

Sintaticamente, é iniciada pela palavra reservada **do** seguida das instruções, que poderá ser uma só ou um bloco de instruções entre chavetas, e no final a palavra reservada **while** seguida da condição de permanência, entre parêntesis. O diagrama da figura 5.1 descreve a regra sintática do **do-while**.

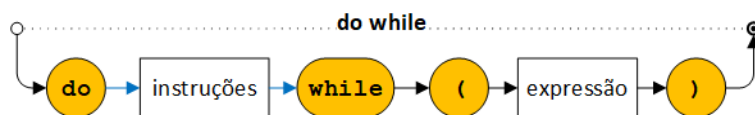


Figura 5.1: Diagrama sintático da repetição *do-while*

O valor resultante da expressão da condição de permanência tem que ser do tipo Boolean.

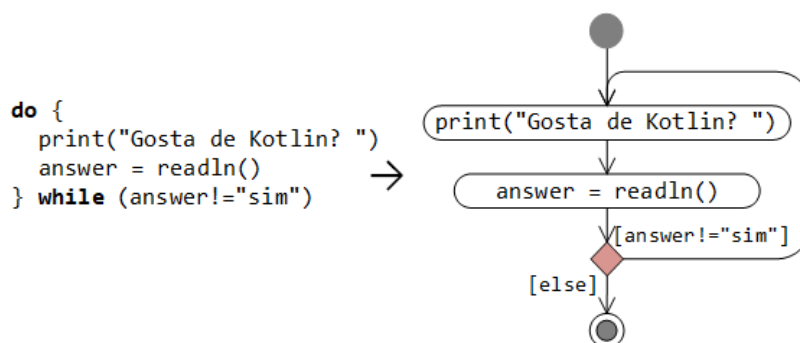


Figura 5.2: Diagrama de atividade do *do-while*

O diagrama da figura 5.2 mostra a execução de um **do-while** que repete a pergunta e lê a resposta até ser dada uma resposta afirmativa.

O programa da listagem 5.1 utiliza dois ciclos **do-while**. O primeiro repete a leitura do número introduzido pelo utilizador enquanto este for maior que 30 ou não for par. O segundo apresenta os números pares desde o número introduzido até 30, inclusive.

```

fun main() {
    var num : Int
    do {
        print("Número par (<30)? ")
        num = readln().toInt()
    } while (num >= 30 || num % 2 != 0)
    do {
        print("$num ")
        num += 2
    } while ( num <= 30 )
}

```

Listagem 5.1: EvenNumbers.kt

Uma possível utilização deste programa seria:

```

C:\ISEL\PG>kotlin EvenNumbersKt ↵
Número par (<30)? 5 ↵
Número par (<30)? 2 ↵
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

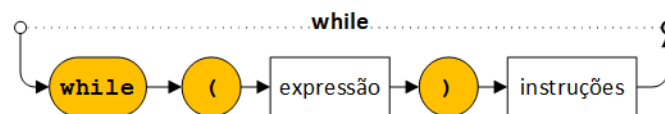
```

5.2 Repetição com condição inicial - *while*

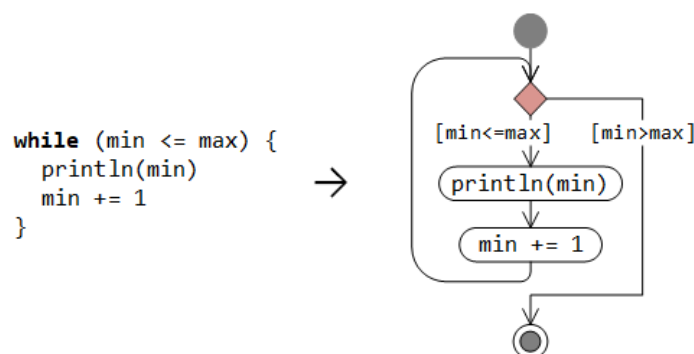
```
while( condição ){ código }
```

A instrução **while** verifica primeiro a condição de permanência e executa zero ou várias vezes o bloco de código.

Sintaticamente, é iniciada pela palavra reservada **while** seguida da expressão entre parêntesis e depois a instrução ou bloco de instruções entre chavetas.

Figura 5.3: Diagrama sintático da repetição *while*

O diagrama da figura 5.4 mostra a execução de um **while** que apresenta todos os valores entre **min** e **max**, avançando o valor de **min** até atingir o valor de **max**.

Figura 5.4: Diagrama de atividade do *while*

O programa da listagem 5.2 utiliza um ciclo **while** para apresentar uma sequência de símbolos, e os respetivos códigos. O primeiro e o último símbolo da sequência são introduzidos pelo utilizador numa só linha.

```

fun main() {
    print("Primeiro símbolo e último símbolo? ")
    val line = readln()           // Lê os dois símbolos
    var min = line[0]             // Extrai o primeiro símbolo
}

```

```

val max = line[1] // Extrai o segundo símbolo
while (min <= max) {
    println("$min - ${min.code}") // símbolo - código
    min+=1;
}
println('.')
}

```

Listagem 5.2: CharSeq.kt

Uma possível utilização deste programa seria:

```

C:\ISEL\PG>kotlin CharSeqKt ↵
Primeiro símbolo e último símbolo? RU ↵
R - 82
S - 83
T - 84
U - 85
.

```

Para esta utilização, a representação dos valores e das sucessivas alterações da variável `min` no final da execução é apresentada na figura 5.5.

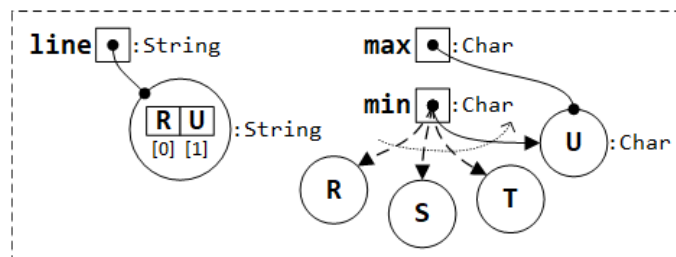


Figura 5.5: Representação da execução de CharSeq

Se o primeiro símbolo introduzido for maior que o último, o ciclo `while` não realiza qualquer iteração, porque a condição é falsa na primeira vez que é verificada.

```

C:\ISEL\PG>kotlin CharSeqKt ↵
Primeiro símbolo e último símbolo? FA ↵
.

```

O programa da listagem 5.3 indica se a palavra introduzida é, ou não, palíndromo. Diz-se que é palíndromo o texto que se lê da mesma forma da esquerda para a direita e da direita para a esquerda.

O programa guarda a palavra em `word` e usa as variáveis `i` e `j` para percorrer os caracteres da palavra. A variável `i` percorre da esquerda para a direita, começando em zero (o local da primeira letra) e vai incrementando. A variável `j` percorre da direita para a esquerda, começando em `word.length-1` (o local da última letra) e vai decrementando.

O ciclo `while` vai comparando a letra esquerda com a direita enquanto o valor do `i` for menor que o `j`, ou seja, enquanto não chegarem ao meio da palavra, e as letras comparadas forem iguais.

```

fun main() {
    print("Palavra? ")
    val word = readln()
    var i=0 // Esquerda -> direita
    var j=word.length-1 // Direita -> esquerda
    while( i<j && word[i]==word[j] ) {
        i += 1 // Próxima à direita
        j -= 1 // Próxima à esquerda
    }
    val res = if (i>=j) "é" else "não é"
    println("$word $res um palíndromo.")
}

```

Listagem 5.3: Palindrome.kt

Duas possíveis utilizações deste programa são:

```
C:\ISEL\PG>kotlin PalindromeKt ↵
Palavra? reviver ↵
reviver é um palíndromo.
C:\ISEL\PG>kotlin PalindromeKt ↵
Palavra? papa ↵
papa não é um palíndromo.
```

Para a primeira utilização, a representação dos valores e das sucessivas alterações das variáveis *i* e *j* no final da execução é apresentada na figura 5.6.

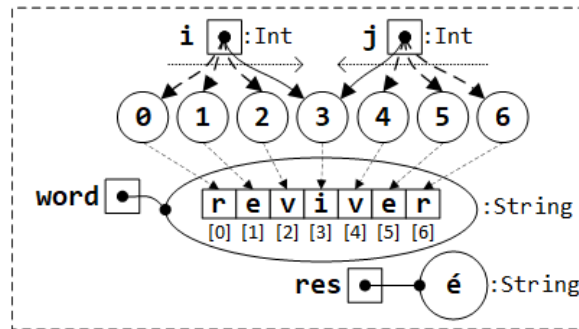


Figura 5.6: Representação da execução de **Palindrome**

Os operadores lógicos `&&` e `||` fazem sempre avaliação parcial. Assim, a condição `i < j && word[i] == word[j]` é avaliada por partes. Se a avaliação de `i < j` der `false` já não é avaliada a parte direita, porque o resultado final é `false` independentemente do valor de `word[i] == word[j]`.

5.3 Incrementar e decrementar

`i++` `j--`

Somar ou subtrair uma unidade ao valor da variável são operações muito frequentes nas instruções de repetição. Para incrementar e decrementar variáveis existem os operadores `++` e `--` em *Kotlin*.

Resumidamente, podemos dizer que: Para qualquer variável *X* de um tipo que suporte a soma aritmética, a expressão `++X` é equivalente à afetação `X+=1`, e `--X` é equivalente a `X-=1`. Mas não é bem assim.

Enquanto que `X+=1` é uma instrução, `++X` é uma expressão, ou seja, da avaliação `++X` resulta um valor que pode ser usado com outras operações numa expressão mais complexa. Por exemplo, supondo que a variável *X* tinha o valor 5, da avaliação da expressão `5+(++X/2)` além de alterar o valor de *X* para 6, resulta o valor 8.

Os operadores `++` e `--` podem ser usados com prefixo (`++X`) ou sufixo (`X++`). Em ambas as situações é alterado o valor da variável, mas o valor que resulta da avaliação é diferente. Do operador prefixo resulta o valor depois da alteração, mas do operador sufixo resulta o valor antes da alteração.

O seguinte troço de código demonstra a utilização dos operadores `++` e `--`, prefixo e sufixo:

```
var x = 0
println(++x)    // -> 1 (depois de incrementar)
println(x++)    // -> 1 (antes de incrementar)
println(x)      // -> 2
println(--x)     // -> 1 (depois de decrementar)
println(x--)    // -> 1 (antes de decrementar)
println(x)      // -> 0
```

No programa da listagem 5.3 as instruções de afetação podem ser substituídas, ficando na expressão do `while` a comparação das letras e a alteração das variáveis *i* e *j*, depois de usar os seus valores, ficando o `while` sem instruções.

```
while( i < j && word[i++] == word[j--] ) { /*EMPTY*/ }
```

Mas atenção, porque desta forma as variáveis são alteradas mesmo que as letras já não sejam iguais.

5.4 Repetição por iteração de sequência - *for*

```
for(i in 1..10){ código }
```

A instrução `for` executa o bloco de código para cada valor de uma sequência.

A sequência pode ser um intervalo, uma string ou, como será apresentado em capítulos mais adiante, pode ser qualquer tipo iterável.

Por exemplo, o seguinte troço de código escreve os valores de 1 a 10 separados por um espaço.

```
for( i in 1..10 )
    print("$i ") // -> 1 2 3 4 5 6 7 8 9 10
```

O diagrama da figura da figura 5.7 mostra a execução deste `for`.

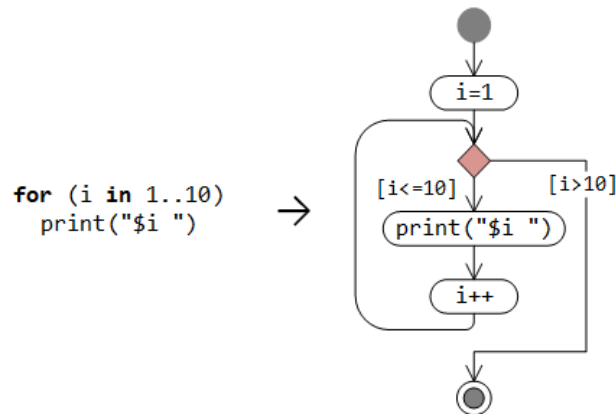


Figura 5.7: Diagrama de atividade do `for` para um intervalo

Sintaticamente, esta instrução é iniciada pela palavra reservada `for` seguida da condição para percorrer a sequência entre parêntesis e depois as instruções executar. A condição é descrita indicando o nome, e opcionalmente o tipo, que terá cada valor da sequência seguido da palavra `in` e depois a expressão da sequência.

O diagrama da figura 5.8 descreve a regra sintática do `for`.

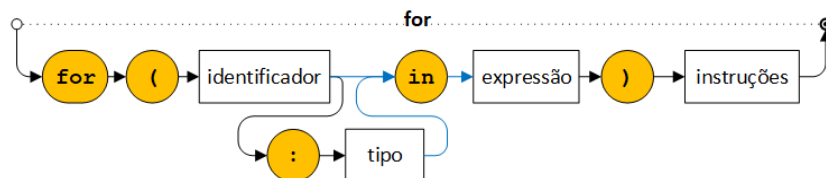


Figura 5.8: Diagrama sintático do `for`

O identificador (nome) do valor a usar na iteração do `for` é automaticamente considerado `val`, logo, não é uma variável e não admite afetações. O tempo de vida deste valor está condicionado a cada iteração do ciclo `for`, ou seja, já não existe nas instruções depois do `for`.

O programa da listagem 5.4 demonstra a utilização do `for` com as diversas formas de definição de intervalos (secção 2.6).

```
fun main() {
    for (c in 'A'..'Z') print(c)           // Intervalo de símbolos
    println()
    for (v in 15 downTo 5) print("$v ") // Intervalo descendente
    println()
    val range = 1 until 21
    for (i in range step 2) print("$i ") // Intervalo com passo 2
    println()
    print("Usando o ")
    for (l in "gps") print(l-1)          // Iteração sobre string
    println()
}
```

}

Listagem 5.4: DemoFor.kt

A utilização deste programa seria:

```
C:\ISEL\PG>kotlin DemoForKt ↵
ABCDEFGHIJKLMNOPQRSTUVWXYZ
15 14 13 12 11 10 9 8 7 6 5
1 3 5 7 9 11 13 15 17 19
Usando o for
```

O programa da listagem 5.2 pode ser reformulado, conforme o da listagem 5.5, usando um `for` em vez do ciclo `while` para apresentar uma sequência de símbolos, em que o primeiro e o último símbolos da sequência são introduzidos pelo utilizador.

```
fun main() {
    print("Primeiro símbolo e último símbolo? ")
    val line = readln()           // Lê os dois símbolos
    for(c in line[0]..line[1])
        println("$c - ${c.code}") // símbolo - código
    println('.')
}
```

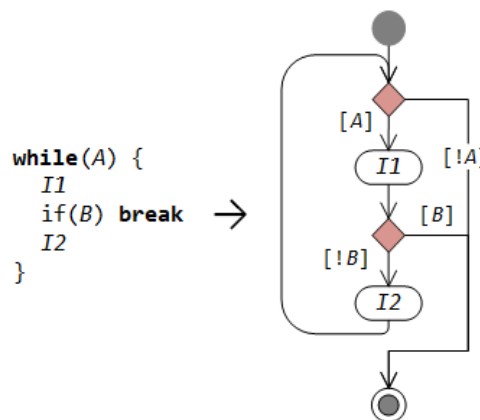
Listagem 5.5: CharSeq2.kt

5.5 Quebra da repetição - *break*

if (condição) break

A expressão `break` termina imediatamente o ciclo (`do-while`, `while` ou `for`) onde está inserido, continuando a execução na instrução depois do ciclo.

Sintaticamente, a versão mais simples é apenas a palavra `break`.

Figura 5.9: Diagrama de atividade com *while* e *break*

O diagrama 5.9 mostra a execução de um `while` com um `break`, em que *A* e *B* são duas condições (expressões booleanas) e *I1* e *I2* são duas instruções.

O programa da listagem 5.6 conta as letras vogais da linha de texto introduzida. Mas, se o texto tiver um ponto final, só conta as vogais até ao ponto final.

```
fun main() {
    print("Linha de texto? ")
    val line = readln()
    var vowels=0;
    for (c in line) when (c) {
        '.' -> break // Termina o for sem chegar ao fim da linha
        in "aeiouAEIOU" -> ++vowels
    }
    println("O texto tem $vowels vogais.")
}
```


}

Listagem 5.6: Vowels.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin VowelsKt ↵
Linha de texto? Texto para contar vogais ↵
0 texto tem 9 vogais.
C:\ISEL\PG>kotlin VowelsKt ↵
Linha de texto? Texto para contar. vogais ↵
0 texto tem 6 vogais.
```

O programa da listagem 5.3 pode ser reformulado, conforme o da listagem 5.7, passando a usar um `for` que percorre a palavra desde os extremos até ao centro. Quando encontra letras diferentes termina a iteração depois de assinalar que não é palíndromo.

```
fun main() {
    print("Palavra? ")
    val word = readln()
    var res = "é"
    for(i in 0..word.length/2)
        if (word[i] != word[word.length-1-i]) {
            res = "não é"
            break
        }
    println("$word $res um palíndromo.")
}
```

Listagem 5.7: Palindrome2.kt

5.6 Valores e variáveis locais

```
for(...) {val v = exp ... }
```

Os blocos de código dos ciclos podem declarar valores ou variáveis, mas esses ficam com tempo de vida condicionado ao bloco, ou seja, já não existem nas instruções depois do bloco.

Por exemplo, o programa da listagem 5.8, que calcula o máximo divisor comum entre dois valores inteiros segundo o algoritmo de *Euclides*, declara o valor `r` no bloco do ciclo `while`. O tempo de vida de `r` termina no fim do bloco.

```
fun main() {
    print("Primeiro valor? ")
    var m = readln().toInt()
    print("Segundo valor? ")
    var n = readln().toInt()
    if (n==0) n=m
    else
        while (true) {
            val r = m % n //Valor r é local ao bloco
            // println("m=$m n=$n r=$r")
            if (r==0) break
            m = n
            n = r
        }
    println("Máximo divisor comum = $n")
}
```

Listagem 5.8: Euclides.kt

Uma possível utilização deste programa seria:

```
C:\ISEL\PG>kotlin EuclidesKt ↵
Primeiro valor? 348 ↵
Segundo valor? 156 ↵
Máximo divisor comum = 12
```

O algoritmo de Euclides descreve uma forma eficiente para calcular o máximo divisor comum (*mdc*) entre dois valores m e n , em que $n \neq 0$:

- Determina-se o resto da divisão fazendo $r \leftarrow m \% n$
- Se $r = 0$ então o *mdc* é o valor de n
- Caso contrário, $m \leftarrow n$, $n \leftarrow r$ e repete-se o processo.

Para o caso inicial em que $n = 0$, não resolvido pelo algoritmo para evitar a divisão por zero, sabe-se que o *mdc* entre 0 e um valor qualquer é esse valor.

A evolução do algoritmo, para o exemplo de utilização, em que $m = 348$ e $n = 156$, é a apresentada na tabela 5.1.

Tabela 5.1: Evolução do algoritmo de Euclides

m	n	$r \leftarrow m \% n$
348	156	36
156	36	12
36	12	0

As linhas desta tabela podem ser apresentadas na execução do próprio programa se for descomentada a linha com `println("m=$m n=$n r=$r")`.

Desta forma, repetindo a utilização do programa seria:

```
C:\ISEL\PG>kotlin EuclidesKt ↵
Primeiro valor? 348 ↵
Segundo valor? 156 ↵
m=348 n=156 r=36
m=156 n=36 r=12
m=36 n=12 r=0
Máximo divisor comum = 12
```

Acrescentar chamadas auxiliares a `println()` para apresentar os valores parciais da execução de um programa, é uma técnica bastante utilizada como forma de compreender e corrigir erros de um programa.

O processo de procurar e corrigir erros é designado por *debugging* ou depuração. Acrescentar chamadas auxiliares a `println()` é uma técnica de *debugging* aceitável, quando não é possível usar ferramentas específicas para depuração.

Capítulo 6

Funções

As funções são blocos de código com um nome associado que podem ser executadas chamando-as por esse nome a partir de outras funções.

Nos pequenos programas realizados nos capítulos anteriores foi declarada apenas a função `main()`, mas foram chamadas outras funções que fazem parte da biblioteca do *Kotlin*.

Para evitar a repetição de código e a complexidade exagerada da função `main()`, os programas devem estar organizado em várias funções.

6.1 Funções sem parâmetros

```
fun nome() { código }
```

O programa da listagem 6.1 apresenta uma grelha com 4 colunas, sendo o número de linhas indicado pelo utilizador. A grelha é apresentada textualmente com linhas de dois padrões de texto. As linhas com o padrão `"+---"` e as linhas com o padrão `"| "`. As funções `printLineDivision()` e `printLineSpace()` escrevem cada uma destas linhas.

O valor `COLS`, declarado fora do corpo das funções, pode ser utilizado por qualquer função.

```
val COLS = 4           // Valor global comum a todas as funções
fun printLineDivision() {           // Declaração de função
    for(cols in 1..COLS) print("+---")
    println('+')
}
fun printLineSpace() {           // Declaração de função
    for(cols in 1..COLS) print("|   ")
    println('|')
}

fun main() {           // Declaração de função
    print("Linhas da grelha? ")
    val lines = readln().toInt()
    for (l in 1..lines) {
        printLineDivision()       // Chamada à função
        printLineSpace()          // Chamada à função
    }
    printLineDivision()           // Chamada à função
}
```

Listagem 6.1: Grid.kt

Uma possível utilização deste programa seria:

```
Linhas da grelha? 2 
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
```

Cada declaração de função é prefixada pela palavra reservada `fun` seguida do nome que se pretende atribuir à função, seguido do par de parêntesis curvos (porque a função não tem parâmetros) seguido do corpo da função, que é indicado num bloco de código entre chavetas.

A ordem da declaração das funções é arbitrária. Poderia estar declarada em primeiro lugar a função `main()` e depois as outras.

Cada chamada a uma função indica o nome da função a chamar seguido do par de parêntesis curvos (quando a função não tem parâmetros).

Neste programa também existem chamadas às funções `print()` e `println()` que fazem parte da biblioteca standard do *Kotlin*, mas nestes casos são funções com parâmetros.

A figura 6.1 mostra as chamadas realizadas entre as funções declaradas no ficheiro `Grid.kt`.

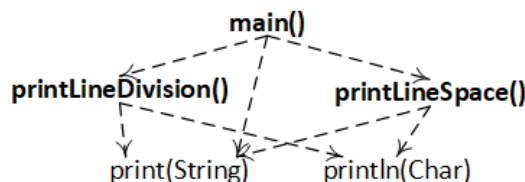


Figura 6.1: Chamadas das funções em `Grid.kt`

6.1.1 Valores e variáveis globais

Os valores (`val`) ou variáveis (`var`) declaradas fora do corpo das funções, são globais e qualquer função pode usar. Estes valores persistem durante toda a execução do programa.

O valor `COLS` da listagem 6.1 é global.

Os valores e as variáveis declaradas no corpo de uma função, são locais à função e só ela pode usar. Estes valores têm o tempo de vida condicionado à execução na função.

O valor `lines` da listagem 6.1 é local à função `main()`.

6.2 Funções com parâmetros

```
fun nome( parâmetros ) { código }
```

As funções podem receber informação que é indicada no momento da chamada da função. Essa informação é passada à função através dos parâmetros.

Sintaticamente, na definição da função, os parâmetros são definidos entre os parêntesis curvos e separados por vírgulas, logo a seguir ao nome da função. Cada parâmetro tem um nome e o seu tipo separados por dois pontos.

Em cada chamada à função é indicado o nome e os argumentos entre parêntesis separados por vírgulas. Os argumentos indicam os valores atuais dos parâmetros no momento da chamada.

O programa da listagem 6.2 apresenta uma grelha com o mesmo aspeto do programa anterior, mas o número de colunas também é introduzido pelo utilizador.

```
// Declaração da função com a assinatura: printLinePattern(Char,Char,Int)
// com os parâmetros: top, middle e times
fun printLinePattern(top: Char, middle: Char, times: Int) {
    for(n in 1..times) print("$top$middle$middle$middle")
    println(top)
}

// Declaração da função com a assinatura: printLineDivision(Int)
fun printLineDivision(cols: Int) {
    printLinePattern('+','- ',cols) // Chamada com argumentos: '+', '- ', cols
}
```

```
// Declaração da função com a assinatura: printGrid(Int,Int)
fun printGrid(lines: Int, cols: Int) {
    for (l in 1..lines) {
        printLineDivision(cols) // Chamada com o argumento: cols
        printLinePattern('|',' ',cols) // Chamada argumentos: '|', ' ', cols
    }
    printLineDivision(cols) // Chamada com o argumento: cols
}

fun main() {
    print("Linhas da grelha? ")
    val numLines = readln().toInt()
    print("Colunas da grelha? ")
    val numCols = readln().toInt()
    printGrid(numLines,numCols) // Chamada argumentos: numLines e numCols
}
```

Listagem 6.2: Grid2.kt

Uma possível utilização deste programa seria:

```
Linhas da grelha? 3 ↵
Colunas da grelha? 6 ↵
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
```

A função `printLinePattern()` tem três parâmetros: `top`, `middle` e `times`. Estes parâmetros recebem informação sobre o símbolo no topo do padrão e no final da linha (`top` do tipo `Char`), o símbolo que se repete 3 vezes no padrão depois do topo (`middle` do tipo `Char`) e o número de vezes que se repete o padrão (`times` do tipo `Int`).

A função escreve uma linha repetindo `times` vezes o padrão formado pelos símbolos `top` e `middle`, escrevendo também o símbolo `top` no final da linha.

A figura 6.2 mostra as chamadas realizadas entre as funções declaradas no ficheiro `Grid2.kt`.

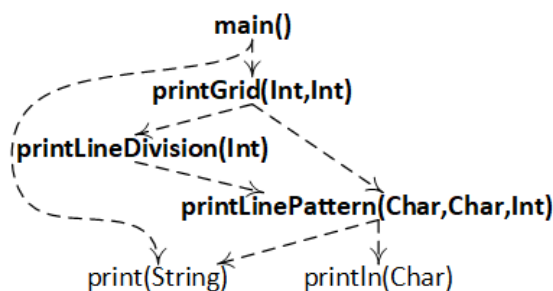


Figura 6.2: Chamadas das funções em Grid2.kt

A função `printLinePattern()` é chamada em dois locais diferentes do programa. Na primeira chamada são passados os argumentos `'+'`, `'-'` e `cols` e na segunda chamada são passados os argumentos `'|'`, `' '` e `cols`, para corresponderem aos valores dos parâmetros `top`, `middle` e `times`.

6.2.1 Argumentos posicionais ou nomeados

A correspondência entre os argumentos passados no momento da chamada e os valores recebidos nos parâmetros é posicional de acordo com a ordem com que são usados.

No entanto, é possível passar os valores dos argumentos independentemente da posição, nomeando os argumentos com o nome do parâmetro. Por exemplo, a primeira chamada a `printLinePattern()` poderia ficar:

```
printLinePattern(middle='-',top='+',times=cols)
```

e a segunda poderia ser:

```
printLinePattern(' | ', times=cols, middle=' ')
```

Na segunda chamada o primeiro argumento é posicional e os restantes são nomeados. Em cada chamada, quando se nomeia um argumento, todos os restantes à direita, têm que ser também nomeados.

Usando argumentos posicionais ou nomeados a sintaxe de chamada a função passa a ser a descrita no diagrama da figura 6.3, substituindo a descrita no diagrama 2.12.

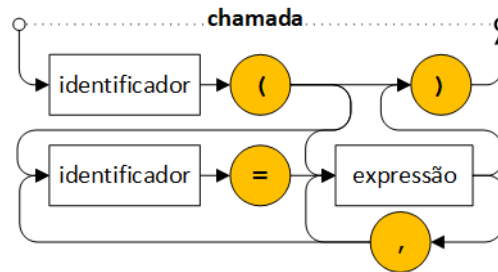


Figura 6.3: Diagrama sintático da chamada a função

6.2.2 Parâmetros não são variáveis

Os parâmetros das funções são valores (`val`) e não variáveis (`var`). Portanto, não é possível alterar os valores dos parâmetros.

Por exemplo, não é possível implementar a função `printLinePattern()` da seguinte forma:

```
fun printLinePattern(top: Char, middle: Char, times: Int) {
    while(times>0) {
        print("$top$middle$middle$middle")
        --times //ERRO: times não é variável
    }
    println(top)
}
```

6.3 Assinatura e sobrecarga

nome(*tipos dos parâmetros*)

A assinatura da função é a combinação do nome da função com os tipos de cada parâmetro. Por exemplo, a função `printLinePattern()` tem a assinatura `printLinePattern(Char,Char,Int)`.

Podem ser declaradas várias funções com o mesmo nome desde que tenham assinaturas distintas. Quando tal acontece, diz-se que a função foi sobrecarregada (*overloaded*).

Por exemplo, as funções `show()`, declaradas a seguir, apesar de terem o mesmo nome e o mesmo número de parâmetros, têm assinaturas distintas porque o tipo do primeiro parâmetro é diferente.

Dependendo do tipo do valor do primeiro argumento passado na chamada é possível distinguir qual a função concreta a chamar.

```
fun show(symb :Char, label :String) { //Assinatura: show(Char,String)
    println("$label = '$symb'")
}
fun show(value :Int, label: String) { //Assinatura: show(Int,String)
    println("$label = $value")
}

fun main() {
    show('A',"Char") //Chama show(Char,String) e escreve: Char = 'A'
    show(27,"Valor") //Chama show(Int,String) e escreve: Valor = 27
}
```

As funções `print()` e `println()`, da biblioteca do *Kotlin*, estão sobrecarregadas para suportarem chamadas com um argumento de qualquer tipo. Por exemplo, a função `print()` foi sobrecarregada com as assinaturas `print(Int)`, `print(Char)`, `print(Double)`, etc.

6.3.1 Argumentos por omissão

Quando existe um valor de um argumento que é frequentemente usado, podemos evitar que argumento seja indicado, no momento da chamada, assumindo esse valor por omissão para o parâmetro.

Por exemplo, a função `show()` do exemplo anterior poderia ser declarada com valores por omissão no parâmetro `label`.

```
fun show(symb :Char, label :String = "Char") {
    println("$label = '$symb'")
}
fun show(value :Int, label: String = "Int") {
    println("$label = $value")
}

fun main() {
    show('A') //Chama show(Char) e escreve: Char = 'A'
    show(27, "Valor") //Chama show(Int,String) e escreve: Valor = 27
}
```

Nesta situação, cada declaração da função `show()` tem duas assinaturas. A primeira tem `show(Char,String)` e `show(Char)`. A segunda tem `show(Int,String)` e `show(Int)`.

6.4 Retorno das funções

```
fun nome(...):tipo
```

As funções podem retornar um valor a quem as chama e o valor retornado pode ser usado pelo chamador.

Por exemplo, a função `power()`, declarada a seguir, recebe a base e o expoente e retorna a potência, fazendo multiplicações sucessivas, admitindo que o expoente é maior ou igual a zero.

```
fun power(base :Int, exp :Int) :Int {
    var value = 1;
    for (e in 1..exp) value *= base
    return value //Termina a execução da função retornando o valor
}

fun main() {
    val res = power(2,8) + power(10,3)
    println("2^8 + 10^3 = $res") // -> 2^8 + 10^3 = 1256
    println("5^0 = ${power(5,0)}") // -> 5^0 = 1
}
```

A execução da instrução `return` termina a execução da função, retornando o valor da expressão indicada a seguir.

Pode existir mais do que um `return` na mesma função se for necessário a função terminar em pontos de execução diferente.

6.4.1 Funções puras

Designam-se como funções puras, as que não produzem nenhum efeito colateral, o valor retornado só depende dos valores dos parâmetros e retorna sempre o mesmo valor para os mesmos argumentos. Um efeito colateral é usar ou alterar algo exterior à função, como por exemplo, chamar `println()` ou `readln()` ou afetar uma variável global.

Segundo este princípio, a função `power()` é pura. Se a função for chamada várias vezes com `power(2,10)` retornará sempre 1024 e não produz mais nenhum efeito.

As funções que não retornam valores não são puras porque produzem algum efeito colateral, caso contrário não serviriam para nada.

Sempre que possível, devemos implementar funções puras, porque são fáceis de testar, chamando várias vezes com determinados argumentos e verificando o valor retornado. Também não são propícias a gerar erros inadvertidos, por não terem efeitos colaterais.

A função `validInt()` da listagem 6.3 tem três pontos de retorno. Dois que retornam `false` e um, no final, que retorna `true`.

Esta função verifica se o texto indicado no parâmetro `number` descreve um valor inteiro de acordo com o diagrama sintático da figura 6.4. A função retorna `true` se a string `number` contém apenas dígitos (de 0 até 9) prefixados ou não com um sinal `+` ou `-`, sendo portanto uma função pura.

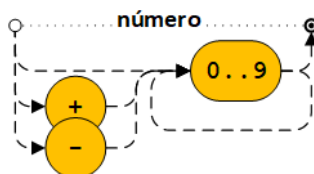


Figura 6.4: Número válido

```
fun validInt(number :String) :Boolean {
    val idx = if(number.length>0 && number[0] in "+-") 1 else 0
    if (idx == number.length) // string vazia ou só com o sinal?
        return false
    for(i in idx until number.length) {
        if (number[i] !in '0'..'9') // não é dígito?
            return false
    }
    return true
}

fun main() {
    println(validInt("123")) // -> true
    println(validInt("-57")) // -> true
    println(validInt("+22")) // -> true
    println(validInt("")) // -> false
    println(validInt("+")) // -> false
    println(validInt("-")) // -> false
    println(validInt("12a23")) // -> false
    println(validInt("-23+")) // -> false
}
```

Listagem 6.3: Função `validInt()`

A função `max()` declarada a seguir é pura e retorna o maior dos dois valores indicados como parâmetros, por isso, retorna em dois pontos diferentes.

```
fun max(a :Int, b :Int) :Int {
    if (a>b) return a // Ponto de retorno
    return b // Outro ponto de retorno
}
```

De facto, pode existir um só `return` usando o `if` como expressão, como mostra a declaração seguinte:

```
fun max(a :Int, b :Int) :Int {
    return if (a>b) a else b // Um só ponto de retorno
}
```

6.4.2 Expressão como corpo da função

fun nome(...) = exp

Quando uma função tem apenas uma instrução que faz `return` de uma expressão no seu corpo (bloco de código), podemos substituir o bloco pelo símbolo `=` seguido da expressão retornada.


```
fun max(a :Int, b :Int) :Int = if (a>b) a else b
```

Neste caso, tal como na declaração de valores (`val`) e de variáveis (`var`), o tipo pode ser omitido porque é inferido a partir da expressão, ficando da seguinte forma:

```
fun max(a :Int, b :Int) = if (a>b) a else b
```

Como regra geral, podemos estabelecer a seguinte equivalência sintática:

$$\text{fun nome}(\dots) : \text{type} \{ \text{return exp} \} \Rightarrow \text{fun nome}(\dots) = \text{exp}$$

O diagrama da figura 6.5 descreve a regra sintática da declaração de funções, com tudo o que foi descrito até este momento.

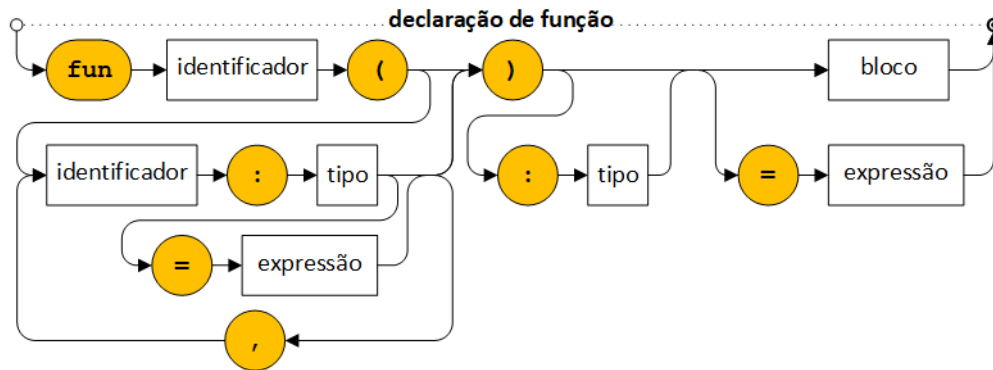


Figura 6.5: Diagrama sintático da declaração de funções

6.4.3 Retorno do tipo *Unit*

De facto, todas as funções em *Kotlin* têm um tipo de retorno. Quando a função não tem resultado útil, o seu tipo de retorno é `Unit`. O tipo de retorno `Unit` pode estar explícito ou ser omitido.

Quando é indicado o corpo da função entre chavetas e se omite o tipo de retorno, é assumido o tipo `Unit`. Por exemplo, a função `main()` retorna `Unit` e poderia ser declarada da seguinte forma:

```
fun main() :Unit { ... }
```

O `return` também pode ser usado em funções do tipo `Unit` para terminar a execução antes da última instrução. Nesse caso não existe a expressão depois da palavra reservada `return`.

No programa da listagem 6.4, a função `readValues()` lê os valores inteiros introduzidos pelo utilizador até ser introduzida a palavra FIM. Esta função faz um `return` sem expressão para terminar a execução dentro de um ciclo `while`, quando lê a palavra FIM.

O somatório dos valores lidos é acumulado na variável global `sum` e o número de valores lidos é atualizado na variável global `count`.

Obviamente que esta função não é pura porque, para além de ler os valores introduzidos no *input*, afeta as variáveis `sum` e `count` exteriores á função.

```
var sum = 0    //Variável global com a soma dos valores lidos
var count = 0 //Variável global com a contagem dos valores lidos

fun readValues() {
    while(true) {
        val txt = readln().trim()
        if (txt=="FIM") return //Terminação da função
        sum += txt.toInt()
        ++count
    }
}

fun main() {
    println("Escreva um valor inteiro por linha.")
    println("Para terminar escreva FIM.")
    readValues()
}
```

```

    val avg = sum.toDouble() / count    //Divisão de Double
    println("Média = $avg")
}

```

Listagem 6.4: Average.kt

O programa da listagem 6.4 apresenta a média de todos os valores inteiros introduzidos pelo utilizador. Como se pretende que a média seja um valor real, o somatório dos valores é convertido para `Double` para garantir que a divisão é de `Double`. Sem esta conversão, como `sum` e `count` são do tipo `Int` a divisão seria de valores inteiros, de onde resultaria o quociente da divisão e `avg` seria também do tipo `Int`.

Uma possível utilização deste programa seria:

```

Escreva um valor inteiro por linha.
Para terminar escreva FIM.
30 ↵
23 ↵
45 ↵
65 ↵
123 ↵
FIM ↵
Média = 57.2

```

6.5 Funções como valores

```
val fx : (Int, Int) -> String
```

As funções também podem ser usadas como valores de um determinado tipo. Tal como os restantes valores em *Kotlin*, também é possível, por exemplo, ter valores que são funções e passar uma função como argumento a outra função.

Por exemplo, consideremos as funções `upperCase()` e `lowerCase()`. A função `upperCase()` transforma uma letra minúscula em maiúscula, caso seja minúscula. A função `lowerCase()` transforma uma letra maiúscula em minúscula, caso seja maiúscula. Ambas são do tipo `(Char) -> Char`, ou seja, funções com um parâmetro do tipo `Char` que retornam um `Char`.

```

fun upperCase(c: Char) = if (c in 'a'..'z') c+('A'-'a') else c
fun lowerCase(c: Char) = if (c in 'A'..'Z') c+('a'-'A') else c

```

Podemos declarar uma variável do tipo `(Char) -> Char` e, usando o operador `::`, afetamos a variável com a função `upperCase()` ou `lowerCase()`.

Para chamar a função da variável, realiza-se uma chamada à função usando o nome da variável como se fosse o nome da função.

```

var fx : (Char) -> Char // Variável fx que refere uma função
fx = ::upperCase        // fx fica a referir a função upperCase
val c = fx('h')         // Chama a função referida por fx
println(c)              // -> H
fx = ::lowerCase        // fx fica a referir a função lowerCase
println(fx('M'))        // -> m

```

As funções `printInUpperCase()` e `printInLowerCase()`, apresentadas a seguir, têm um padrão de código muito semelhante, porque só diferem na função que faz a transformação de cada letra.

```

fun printInLowerCase(txt: String) {
    for (c in txt) print( lowerCase(c) )
    println()
}
fun printInUpperCase(txt: String) {
    for (c in txt) print( upperCase(c) )
    println()
}
fun main() {
    printInLowerCase("AbCdEfGhIJKLmnop") // -> abcdefghijklmnop
    printInUpperCase("AbCdEfGhIJKLmnop") // -> ABCDEFGHIJKLMNOP
}

```

Usando um parâmetro adicional do tipo `(Char)->Char`, podemos declarar apenas uma função `printInFx()` que recebe a função de transformação como parâmetro.

```

fun printInFx(txt: String, fx: (Char)->Char) {
    for(c in txt) print( fx(c) ) // Chama a função passada como parâmetro
    println()
}
fun main() {
    printInFx("AbCdEfGhIJKLmnop", ::lowerCase) // -> abcdefghijklmnop
    printInFx("AbCdEfGhIJKLmnop", ::upperCase) // -> ABCDEFGHIJKLMNOP
}

```

Não é importante para o exemplo, mas a função `printInFx()` é do tipo `(String, (Char)->Char)->Unit` e a função `main()` é do tipo `()->Unit`.

A sintaxe dos tipos das funções é a descrita no diagrama da figura 6.6, onde é possível indicar os nomes dos parâmetros, caso torne o tipo da função mais compreensível.

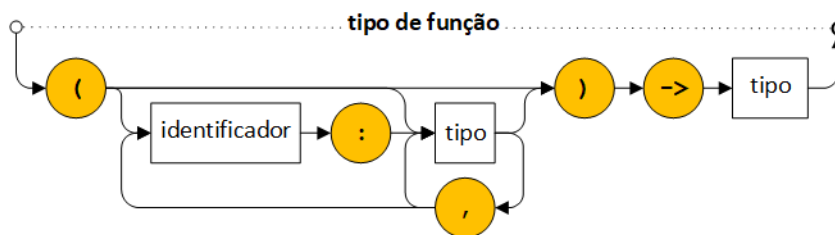


Figura 6.6: Diagrama sintático dos tipos de funções

6.5.1 Expressões *lambda*

É possível definir funções sem nome, numa expressão, designadas por expressões lambda.

Por exemplo, a expressão `{ c:Char -> upperCase(c)+delta }` define uma função com um parâmetro `c` do tipo `Char` que retorna o valor da expressão `upperCase(c)+delta`.

As expressões lambda podem usar tudo o que a função chamadora tem acessível. Portanto, usando as funções `upperCase` e `printInFx` definidas anteriormente, é possível fazer:

```

fun main() {
    val delta = 2 // valor usado na expressão lambda
    val fx = { c:Char -> upperCase(c)+delta } // O tipo da função é inferido
    printInFx("AbCdEfGhIJKLmnop", fx) // -> CDEFGHIJKLMNOPQR
}

```

Ou ainda, passando diretamente a expressão lambda como argumento na chamada à função `printInFx()`:

```

fun main() {
    val delta = 2 // valor usado na expressão lambda
    printInFx("AbCdEfGhIJKLmnop", { c:Char -> upperCase(c)+delta })
}

```

Como o segundo parâmetro da função `printInFx()` é do tipo `(Char)->Char`, já é conhecido o tipo do parâmetro da expressão lambda e este pode ser omitido, ficando:

```

printInFx("AbCdEfGhIJKLmnop", { c -> upperCase(c)+delta })

```

Quando o último parâmetro da função chamada é outra função, então existe a alternativa sintática de poder indicar a expressão lambda depois dos restantes parâmetros à esquerda, ficando:

```
printlnFx("AbCdEfGhIJKLmnop") { c -> upperCase(c)+delta }
```

Finalmente, porque a função da expressão lambda só tem um parâmetro, podemos omitir o parâmetro assumindo que o seu nome é `it`, ficando:

```
printlnFx("AbCdEfGhIJKLmnop") { upperCase(it)+delta }
```

Se a expressão lambda tiver várias instruções no corpo da função, então o valor retornado é o resultado da avaliação da última expressão.

Por exemplo, chamando a função `printlnFx()` anterior, pode ser passada uma expressão lambda que escreve um ponto cada vez que for chamada e vai incrementando o valor que adiciona no código de cada letra, alterando a variável local da função `main()`.

```
fun main() {
    var d = 0
    printlnFx("aBc") { // -> .B.D.F
        print('.')
        ++d           // Incrementa a variável local de main()
        upperCase(it)+d // Valor retornado pela função
    }
}
```

O diagrama da figura 6.7 descreve a regra sintática das expressões lambda.

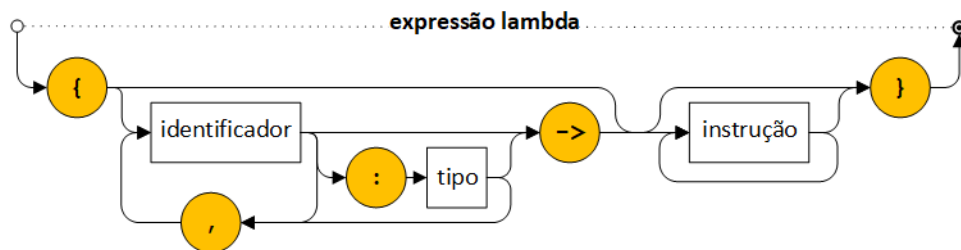


Figura 6.7: Diagrama sintático da expressão lambda

6.5.2 Função *repeat*

Usando as características das expressões lambda, está definida a função `repeat()` na biblioteca standard do *Kotlin*:

```
fun repeat(times: Int, action: (Int)->Unit) {
    for (index in 0 until times)
        action(index)
}
```

Dando a ilusão que existe mais uma instrução de repetição na linguagem, que nos permite escrever código como:

```
repeat(10) {
    print("$it ") // -> 0 1 2 3 4 5 6 7 8 9
}
```

6.6 Múltiplos ficheiros fonte

Um programa pode usar funções que estão distribuídas por vários ficheiros fonte (com extensão `.kt`). Cada ficheiro fonte pode ter várias funções. Pelo menos um dos ficheiros terá a função `main()` do programa.

Admitindo que a função `readInt()` e outras funções que realizem leituras de valores do input serão utilizadas em vários programas, é boa técnica definir estas funções num ficheiro fonte com o nome `Input.kt`. A listagem 6.5 apresenta parcialmente este ficheiro.

```

fun readInt(quest :String) :Int {
    print("$quest? ")
    return readln().trim().toInt()
}
// ... Outras funções com leituras do input.

```

Listagem 6.5: Ficheiro Input.kt

```

fun main() {
    val hours = readInt("Horas")
    val minutes = readInt("Minutos")
    val seconds = readInt("Segundos")
    val totalSeconds = hours * 3600 + minutes * 60 + seconds
    println("$hours:$minutes:$seconds = $totalSeconds segundos.")
}

```

Listagem 6.6: Ficheiro TotalSecs.kt

Desta forma, o programa da listagem 6.6 que usa a função `readInt()` definida em `Input.kt` deve ser compilado conjuntamente com `Input.kt`, usando o comando:

```
C:\ISEL\PG>kotlinc TotalSecs.kt Input.kt ↵
```

Assim, foram produzidos os ficheiros `InputKt.class` e `TotalSecsKt.class`. Para ser executado o programa, usa-se o comando habitual:

```

C:\ISEL\PG>kotlin TotalSecsKt ↵
Horas? 1
Minutos? 11
Segundos? 40
1:11:40 = 4300 segundos.

```

Usando o ambiente integrado *IntelliJ*, podemos definir vários ficheiros fonte no mesmo projeto, cada um com várias funções, e todos farão parte do programa resultante.

Deve existir um critério lógico para a divisão das funções por vários ficheiros fonte. Por exemplo, o ficheiro `Input.kt` tem funções que fazem leituras de informação introduzidas pelo utilizador, o ficheiro `Output.kt` tem as funções que fazem escritas, o ficheiro `Math.kt` tem as funções que realizam cálculos matemáticos, etc.

Capítulo 7

Definição de tipos

No capítulo 2 foram descritos os tipos principais que estão pré-definidos na linguagem *Kotlin*. Existem tipos para armazenar valores inteiros, valores reais, valores lógicos, símbolos, texto, etc.

Para definir novos tipos em *Kotlin*, são declaradas classes. Neste capítulo vamos usar a declaração de classes para os casos particulares da definição simplificada de valores enumerados e de tipos agregados.

A declaração de classes é iniciada com a palavra reservada `class` podendo ser prefixada por outra palavra reservada (um modificador) para indicar um caso particular de classe, dos quais apenas vamos usar `data` para tipos agregados e `enum` para enumerados.

O nome indicado na declaração da classe constitui um novo tipo no sistema de tipos da linguagem.

7.1 Tipos agregados

```
data class Point(val x:Int, val y:Int)
```

Os tipos agregados servem para representar informação com vários componentes ou propriedades, em que cada propriedade pode ser um valor ou uma variável com um nome e o respetivo tipo.

Sintaticamente, a declaração de um tipo agregado simples começa pelas palavras reservadas `data` `class` seguidas pelo nome do tipo e as propriedades entre parêntesis curvos e separadas por vírgulas. Cada propriedade é declarada tal como os valores e as variáveis, mas o tipo não pode ser omitido.

O diagrama da figura 7.1 descreve a regra sintática da declaração mais simples de um tipo agregado.

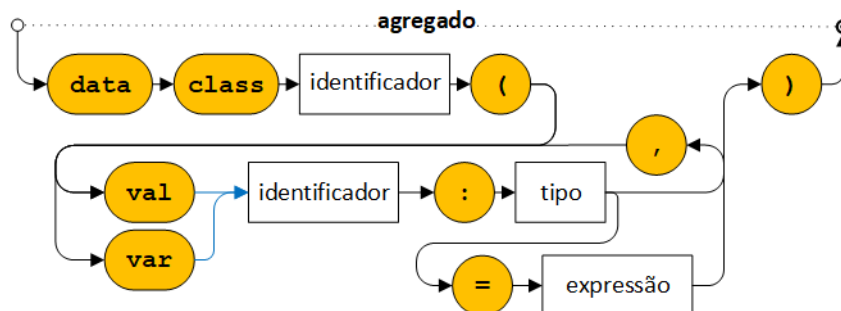


Figura 7.1: Diagrama sintático da definição de tipo agregado

No contexto completo da linguagem *Kotlin*, esta declaração simplificada é uma classe definida apenas com o construtor primário.

Por exemplo, a seguinte declaração define o tipo `Point` com as propriedades `x` e `y` para representar um ponto num plano cartesiano.

```
data class Point(val x:Int =0, val y:Int =0)
```

Cada valor de um tipo definido com uma classe (`class`) é designado como um objeto. Cada objeto tem que ser construído (ou instanciado) explicitamente indicando o tipo e um valor para cada uma das suas propriedades.

Sintaticamente, a construção de um objeto é equivalente a chamar uma função com o nome do tipo em que os parâmetros são os valores iniciais das propriedades.

Por exemplo, a seguinte expressão constrói um ponto na coordenada ($x=20$, $y=15$), sendo a sua avaliação um valor do tipo `Point`.

```
Point(20,15)
```

Os argumentos na construção são posicionais ou nomeados. As propriedades podem ter valores por omissão, assumidos quando são omitidos na construção. O tipo `Point` tem zero por omissão nas duas propriedades.

```
Point()           // ==> Point(0,0)
Point(y=15,x=20)  // ==> Point(20,15)
Point(10)         // ==> Point(10,0)
Point(y=30)       // ==> Point(0,30)
```

A seguinte declaração, constrói um ponto que é armazenado no valor `pt`. A figura 7.2 é a representação dos elementos envolvidos nesta declaração.

```
val pt = Point(20,15)
```

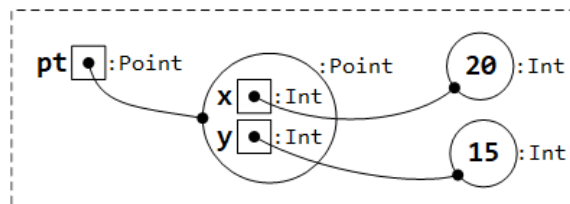


Figura 7.2: Representação de um valor do tipo `Point`

Para aceder a cada propriedade indica-se o objeto e o nome da propriedade pretendida separados por um `.` (ponto).

Por exemplo, as seguintes expressões obtêm os valores das propriedades `x` e `y` de vários objetos do tipo `Point`, usando também o objeto armazenado em `pt`.

```
pt.x           // 20
pt.y           // 15
Point(10,20).x // 10
Point().x      // 0
```

7.1.1 Função `toString()`

A função de conversão `toString()` aplicada a um objeto (`data class`) retorna uma descrição do objeto no formato `Tipo(propriedade1=valor1, propriedade2=valor2, ...)`.

```
println(pt.toString()) // -> Point(x=20, y=15)
```

A função de conversão `toString()` é chamada implicitamente nas expressões embutidas, nas operações de concatenação e nas funções `println()` e `print()` sempre que é necessário apresentar textualmente um objeto.

```
println(pt)           // -> Point(x=20, y=15)
println("pt = $pt")   // -> pt = Point(x=20, y=15)
println("point:" + pt) // -> point:Point(x=20, y=15)
```

7.1.2 Comparação de objetos

O operador `==` entre objetos faz a comparação estrutural dos objetos, comparando estruturalmente o valor de cada propriedade. Só resulta `true` se os dois objetos têm valores iguais em todas as propriedades correspondentes.

O operador `==` só pode ser usado entre objetos e faz a comparação de identidade. Dois valores só são iguais se referirem o mesmo objeto.

O troço de código da listagem 7.1 usa o tipo `Point` para demonstrar as principais características dos tipos agregados. A figura 7.3 é a representação dos elementos envolvidos neste troço de código.

```
val pt = Point(20,15)
println(pt.x)           // -> 20
println(pt.y)           // -> 15
println(pt.toString())  // -> Point(x=20, y=15)
val p1 = Point(pt.x+10)
println("p1 = $p1")      // -> p1 = Point(x=30, y=0)
var p2 = Point(30)
println( p1 == p2 )      // comparação estrutural -> true
println( p1 === p2 )     // comparação identidade -> false
p2 = p1                  // afetação
println( p1 === p2 )     // comparação identidade -> true
```

Listagem 7.1: Utilização de tipos agregados

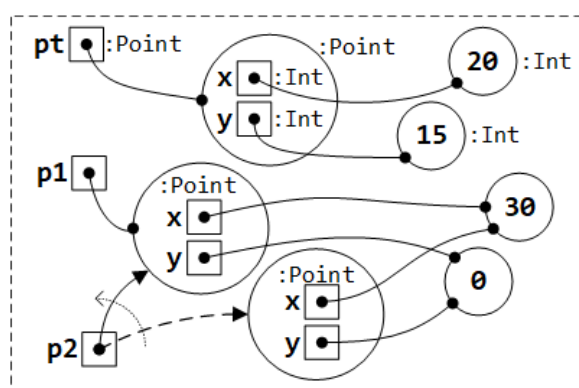


Figura 7.3: Representação dos elementos da listagem 7.1

Usando o tipo `Point` podemos fazer uma função que adiciona dois vetores (parâmetros do tipo `Point`) e outra função que multiplica um vetor por um valor inteiro, ambas retornam um novo objeto `Point` com o vetor resultante.

```
fun addPoints(p1 :Point, p2 :Point) = Point(p1.x+p2.x, p1.y+p2.y)
fun multPoint(p :Point, n :Int) = Point(p.x*n, p.y*n)
```

Uma possível utilização destas funções seria:

```
val pt = Point(20,15)
val add = addPoints(pt, Point(15,10))
val res = multPoint(add, 2)
println( res ) // -> Point(x=70, y=50)
println(multPoint(addPoints(pt,Point(15,10)),2)) // -> Point(x=70, y=50)
```

A seguinte declaração define o tipo `Circle` com as propriedades **radius**, **center** e **color** para representar um círculo numa figura, em que a cor é representada por um valor inteiro.

```
data class Circle(val radius:Int, val center:Point, val color:Int)
```

A seguinte função recebe um círculo e um deslocamento e retorna um novo círculo movido pelo deslocamento indicado.

```
fun moveCircle(circle :Circle, dx :Int, dy :Int) :Circle {
    val center = addPoints(circle.center, Point(dx,dy))
    return Circle(circle.radius, center, circle.color)
}
```

Uma possível utilização desta função seria:

```
val c1 = Circle(12, Point(20,15), 0x0000FF)
println(c1) // -> Circle(radius=12, center=Point(x=20, y=15), color=255)
val c2 = moveCircle(c1, 7, -3)
println(c2) // -> Circle(radius=12, center=Point(x=27, y=12), color=255)
```

7.2 Funções extensão

```
fun Circle.move(dx :Int, dy :Int) :Circle
```

É possível definir funções que são aplicadas a um valor recetor de um determinado tipo, em vez do valor ser um parâmetro. Estas funções são designadas por funções extensão desse tipo e o valor recetor é acedido com `this`.

Por exemplo, em vez das funções `addPoints` e `moveCircle`, apresentada anteriormente, podem ser definidas as seguintes funções extensão de `Point` e de `Circle` apenas com os nomes `add` e `move`.

```
fun Point.add(p2 :Point) = Point(this.x+p2.x, this.y+p2.y)

fun Circle.move(dx :Int, dy :Int) :Circle {
    val c = this.center.add(Point(dx,dy))
    return Circle(this.radius, c, this.color)
}
```

As funções extensão são chamadas usando como prefixo o valor recetor do tipo ao qual se pretende aplicar a função. Assim, no exemplo anterior, a terceira instrução passará a ser:

```
val c2 = c1.move(7, -3)
```

Na implementação das funções extensão, em cada acesso ao valor recetor não é necessário o prefixo `this` quando não existir colisão com outro nome. Assim, as funções anteriores podem ser escritas de forma mais compacta.

```
fun Point.add(p2 :Point) = Point(x+p2.x, y+p2.y)

fun Circle.move(dx :Int, dy :Int) =
    Circle(radius, center.add(Point(dx,dy)), color)
```

As funções extensão podem ser de tipos já definidos. Por exemplo, é possível definir uma função `toPoint` aplicada ao tipo `Int` para converter um valor inteiro num vetor (`Point`) com as propriedades `x` e `y` iguais a esse valor. Neste caso, o `this` é indispensável para aceder ao próprio valor inteiro do recetor.

```
fun Int.toPoint() = Point(this,this)
```

Uma possível utilização desta função poderá ser:

```
println(25.toPoint()) // -> Point(x=25, y=25)
```

7.2.1 Função extensão *copy*

Para facilitar a construção de cópias de objetos, o compilador gera automaticamente a função extensão `copy` para cada `data class`. Esta função tem parâmetros iguais às propriedades definidas com valores por omissão iguais aos valores das propriedades do objeto recetor.

Para o caso do tipo `Circle` a função extensão `copy` gerada automaticamente é:

```
fun Circle.copy(radius=this.radius, center=this.center, color=this.color) =
    Circle(radius, center, color)
```

Chamando a função `copy`, a função `move` pode ser reescrita, passando apenas o parâmetro `center` e deixando os restantes com os valores por omissão:

```
fun Circle.move(dx :Int, dy :Int) =
    this.copy(center= this.center.add(Point(dx,dy)))
```

Omitindo a utilização do `this` fica simplesmente:

```
fun Circle.move(dx :Int, dy :Int) = copy(center= center.add(Point(dx,dy)))
```

7.2.2 Sobrecarga de operadores

A maioria dos operadores definidos em *Kotlin* podem ser sobrecarregados para um determinado tipo usando funções extensão desse tipo.

Por exemplo, é possível sobrecarregar o operador `+` para fazer a soma de dois pontos `p1` e `p2` escrevendo apenas `p1+p2` em vez de chamar a função `add` com `p1.add(p2)`.

As funções que sobrecarregam operadores são prefixadas pela palavra reservada `operator` e têm que ter exatamente o nome associado ao operador em causa. Por exemplo, a sobrecarga dos operadores `+` e `*` são realizadas com as funções `plus` e `times` respetivamente.

As funções `addPoints` e `multPoint`, apresentadas anteriormente, podem ser substituídas pelas funções extensão `plus` e `times` que fazem sobrecarga dos operadores `+` entre dois valores do tipo `Point` e `*` entre `Point` e `Int` ou entre `Int` e `Point`.

```
operator fun Point.plus(p :Point) = Point(x+p.x, y+p.y)
operator fun Point.times(n :Int) = Point(x*n, y*n)
operator fun Int.times(p :Point) = p * this
```

Estas funções são chamadas nas expressões que usam os operadores em causa, mantendo as prioridades definidas na linguagem. Neste exemplo o operador `*` é mais prioritário e por isso é avaliado antes do operador `+`.

```
val pt = Point(20,15)
println( pt + Point(15,10) * 2 ) // -> Point(x=50, y=35)
println( 3 * pt )                // -> Point(x=60, y=45)
```

De facto, as duas últimas instruções são automaticamente transformadas pelo compilador em:

```
println( pt.plus( Point(15,10).times(2) ) ) // -> Point(x=50, y=35)
println( 3.times(pt) )                     // -> Point(x=60, y=45)
```

ou seja, expressões na forma `a + b` são transformadas em `a.plus(b)` e expressões na forma `a * b` são transformadas em `a.times(b)`, desde que existam as funções extensão prefixadas com `operator` para os tipos de valores envolvidos.

A tabela 7.1 resume as transformações que são realizadas usando as funções de sobrecarga de operadores.

Tabela 7.1: Operadores sobrecarregados

Expressão	Transformada em	Expressão	Transformada em
<code>a + b</code>	<code>a.plus(b)</code>	<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>	<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a * b</code>	<code>a.times(b)</code>	<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a / b</code>	<code>a.div(b)</code>	<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a % b</code>	<code>a.rem(b)</code>	<code>a %= b</code>	<code>a.remAssign(b)</code>
<code>a .. b</code>	<code>a.rangeTo(b)</code>		
<code>+a</code>	<code>a.unaryPlus()</code>	<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>		
<code>a > b</code>	<code>a.compareTo(b)>0</code>	<code>a < b</code>	<code>a.compareTo(b)<0</code>
<code>a >= b</code>	<code>a.compareTo(b)>=0</code>	<code>a <= b</code>	<code>a.compareTo(b)<=0</code>
<code>a == b</code>	<code>a?.equals(b)?:b==null</code>	<code>a != b</code>	<code>!(a?.equals(b)?:b==null)</code>
<code>++a</code>	<code>{a=a.inc(); a}</code>	<code>a++</code>	<code>{val a0=a; a=a.inc(); a0}</code>
<code>--a</code>	<code>{a=a.dec(); a}</code>	<code>a--</code>	<code>{val a0=a; a=a.dec(); a0}</code>
<code>a in b</code>	<code>b.contains(a)</code>	<code>a !in b</code>	<code>!b.contains(a)</code>
<code>a[i]</code>	<code>a.get(i)</code>	<code>a[i,...,j]</code>	<code>a.get(i,...,j)</code>
<code>a[i]=b</code>	<code>a.set(i,b)</code>	<code>a[i,...,j]=b</code>	<code>a.set(i,...,j,b)</code>
<code>a()</code>	<code>a.invoke()</code>		
<code>a(i)</code>	<code>a.invoke(i)</code>	<code>a(i,...,j)</code>	<code>a.invoke(i,...,j)</code>

7.3 Enumerados

```
enum class Signal{GREEN, YELLOW, RED}
```

Um enumerado é um tipo que tem um conjunto limitado de valores. Cada valor tem um nome e o respetivo número de ordem na definição.

A declaração de um tipo enumerado simples começa pelas palavras reservadas `enum class` seguidas pelo nome do tipo enumerado e os nomes dos valores possíveis entre chavetas e separados por vírgulas.

O diagrama da figura 7.4 descreve a regra sintática da declaração mais simples de um enumerado.

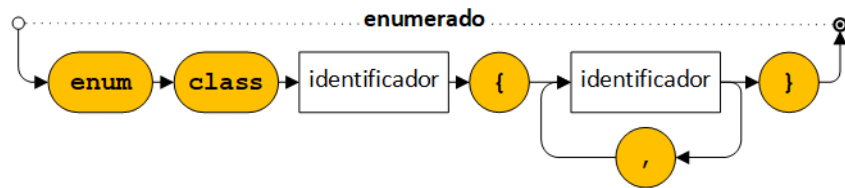


Figura 7.4: Diagrama sintático da definição de enumerado simples

A seguinte declaração define o tipo `Dir` para representar as quatro direções possíveis de um deslocamento

```
enum class Dir { UP, RIGHT, DOWN, LEFT }
```

O nome do enumerado é o nome do tipo definido. Neste caso, o tipo `Dir`.

Cada valor é do tipo do enumerado. Na sua utilização, os valores são prefixados com o nome do tipo do enumerado. Por exemplo, `Dir.LEFT`.

A variável `motion` declarada a seguir só pode ter um dos 4 possíveis valores do tipo `Dir`, sendo `RIGHT` o seu valor inicial. A figura 7.5 é uma representação da variável `motion` e dos 4 diferentes valores do tipo `Dir`.

```
var motion: Dir = Dir.RIGHT
```

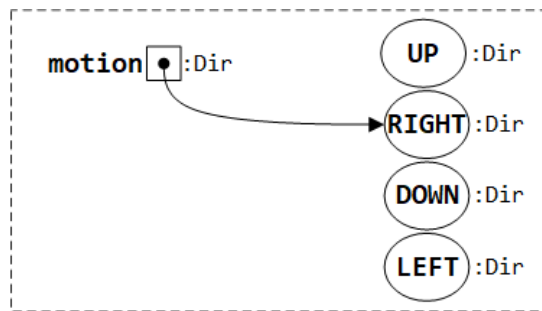


Figura 7.5: Representação de uma variável de um tipo enumerado

O troço de código da listagem 7.2 usa o tipo `Dir` para demonstrar as principais características dos enumerados.

```
var motion = Dir.RIGHT
println(motion.name)           // -> RIGHT
motion = Dir.UP
println("$motion = ${motion.ordinal}") // -> UP = 0
println( motion == Dir.LEFT )   // -> false
println( Dir.DOWN > motion )    // -> true
val deltaY = when(motion) {
    Dir.UP -> +1
    Dir.DOWN -> -1
    else -> 0
}
println("deltaY de $motion = $deltaY") // -> deltaY de UP = 1
for(d in Dir.values())
    print("$d=${d.ordinal} ")          // -> UP=0 RIGHT=1 DOWN=2 LEFT=3
```

Listagem 7.2: Utilização de enumerados

A propriedade `name` de cada valor do enumerado é do tipo `String` e corresponde ao seu nome. Por exemplo, `Dir.LEFT.name` tem o valor `"LEFT"`.

A função de conversão `toString()` aplicada a um valor do enumerado retorna a propriedade `name`. Esta função é chamada nas expressões embutidas e nas funções `println()` e `print()` sempre que é necessário apresentar textualmente um enumerado.

A propriedade `ordinal` de cada valor do enumerado é do tipo `Int` e corresponde ao número de ordem na definição, tendo o primeiro valor o número de ordem zero. Por exemplo, `Dir.LEFT.ordinal` tem o valor 3.

Os valores do enumerado são comparáveis de acordo com o seu número de ordem. Por exemplo, `Dir.LEFT` é maior que `Dir.DOWN`.

A função `values()` aplicada ao tipo do enumerado retorna um iterável que permite percorrer todos os seus valores.

A função extensão seguinte retorna a direção inversa de uma direção

```
fun Dir.reverse() = when(this) {
    Dir.LEFT   -> Dir.RIGHT
    Dir.RIGHT  -> Dir.LEFT
    Dir.UP     -> Dir.DOWN
    Dir.DOWN   -> Dir.UP
}
```

Usando os tipos `Dir` e `Point` é possível fazer uma função extensão de `Dir` que converte uma direção no vetor deslocamento dessa direção.

```
fun Dir.toPoint() = when(this) {
    Dir.LEFT   -> Point(-1,0)
    Dir.RIGHT  -> Point(+1,0)
    Dir.UP     -> Point(0,+1)
    Dir.DOWN   -> Point(0,-1)
}
```

O seguinte troço de código testa o funcionamento destas funções.

```
for(d in Dir.values())
    println("$d: ${d.toPoint()} rev=${d.reverse()}")
/* Output:
UP: Point(x=0, y=1) rev=DOWN
RIGHT: Point(x=1, y=0) rev=LEFT
DOWN: Point(x=0, y=-1) rev=UP
LEFT: Point(x=-1, y=0) rev=RIGHT
*/
```

Usando a função anterior e a soma de valores do tipo `Point` já definida, implementa-se uma função que define o operador soma de um ponto com uma direção, retornando um novo ponto deslocado nessa direção.

```
operator fun Point.plus(d :Dir) = this + d.toPoint()
```

O seguinte troço de código demonstra a sua utilização.

```
println( Point(3,10) + Dir.RIGHT ) // -> Point(x=4, y=10)
```

7.3.1 Enumerados com propriedades

Os enumerados podem ter propriedades agregadas. Para tal define-se as propriedades, tal como nos tipos agregados, e cada valor do enumerado indica logo os valores de cada propriedade.

Por exemplo, o enumerado `Dir` declarado anteriormente pode agregar um símbolo na propriedade `arrow` com a seta de direção e o vetor deslocamento dessa direção na propriedade `delta`.

```
enum class Dir(val arrow: Char, val delta: Point) {
    UP( '\u2191', Point(0,+1) ),
    RIGHT( '\u2192', Point(+1,0) ),
    DOWN( '\u2193', Point(0,-1) ),
    LEFT( '\u2190', Point(-1,0) )
}
```

Usando esta nova definição do tipo `Dir` podemos fazer uma função que apresenta a informação de uma direção.

```
fun Dir.show() {
    println("$name: ($arrow) $delta")
}
```

E já agora, a função `toPoint` apresentada anteriormente passa a ser simplesmente.

```
fun Dir.toPoint() = delta
```

O seguinte troço de código apresenta a informação de cada valor do enumerado `Dir`.

```
for( d in Dir.values() )
    d.show()
/* Output:
UP: (↑) Point(x=0, y=1)
RIGHT: (→) Point(x=1, y=0)
DOWN: (↓) Point(x=0, y=1)
LEFT: (←) Point(x=-1, y=0)
*/
```

7.4 Tipos anuláveis

```
var maybeLine :String? = null
```

O valor especial `null` é tipicamente utilizado para representar a ausência de valor. O sistema de tipos do *Kotlin* distingue explicitamente os tipos que podem ou não conter o valor `null`.

Qualquer tipo `T` com o sufixo `?` é outro tipo `T?` que pode também conter o valor `null`, sendo designado por como um tipo anulável.

Por exemplo, uma variável do tipo `String` não pode conter o valor `null`, sendo essa verificação realizada em tempo de compilação.

```
var line: String = "txt" // Tipo não anulável
println(line.length) // -> 3
line = null // Erro de compilação: não pode conter null
```

Para permitir texto e o valor `null`, a variável tem que ser do tipo `String?`.

```
var maybeLine: String? = "txt" // Tipo anulável
println(maybeLine) // -> txt
maybeLine = null // Ok
println(maybeLine) // -> null
```

No entanto, o compilador não permite usar um valor, que poderá ser `null`, para aceder a uma propriedade ou aplicar uma função.

```
var maybeLine: String? = ... // Tipo anulável
println(maybeLine.length) // Erro de compilação: maybeLine pode ser null
```

A última instrução não é válida porque não é possível aceder à propriedade `length` se o valor de `maybeLine` for `null`.

Para ter um exemplo concreto onde faz sentido usar tipos anuláveis, vamos fazer uma função `dirOfDelta` que retorna a direção associada a determinado deslocamento. Esta seria chamada assim:

```
val dir = dirOfDelta( Point(-1,0) )
println( dir ) // -> LEFT
```

O que é que esta função deve retornar se o parâmetro for um deslocamento, por exemplo `Point(+1,-1)`, que não corresponde a nenhuma direção? Para representar a ausência de direção podemos usar o valor `null`. Portanto, esta função retorna um valor do tipo `Dir?`.

```
fun dirOfDelta(delta: Point): Dir? {
    for (d in Dir.values())
        if (d.delta==delta) return d
    return null
}
```

7.4.1 Verificar condições com *null*

Antes de utilizar um valor que poderá ser *null*, para aplicar uma função ou aceder a uma propriedade, é obrigatório verificar se o valor não é *null*.

```
val dir :Dir? = dirOfDelta( Point(-1,+1) )
dir.show() // Erro de compilação: dir pode ser null
if (dir!=null)
    dir.show() // OK: não é null
```

Depois de qualquer teste que garanta que o valor não é *null*, já é possível usar o valor como sendo de um tipo não anulável.

```
var maybeLine: String? = ...
if (maybeLine!=null) {
    val line: String = maybeLine // OK: não é null
    println(line.length)
}
if (maybeLine==null || maybeLine.isEmpty()) // OK: não é null
    println("No line.")
```

Operador *?.*

O operador *?.* faz o teste implicitamente, só acedendo à propriedade ou à função se o valor não for *null*, caso contrário a expressão tem o valor *null*. Ou seja, a expressão *a?.b* é equivalente a *if(a!=null) a.b else null*.

Por exemplo, o seguinte troço de código apresenta o comprimento da *string* ou *null* e apresenta o conteúdo da *string* em maiúsculas ou *null*.

```
var maybeLine: String? = ...
println(maybeLine?.length) // Parâmetro é do tipo Int?
println(maybeLine?.toUpperCase()) // Parâmetro é do tipo String?
```

Usando o operador *?.* temos uma forma mais compacta de chamar a função *show* apenas se *dir* não for *null*.

```
val dir: Dir? = ...
dir?.show() // OK: só se não for null
```

Operador *?:*

O operador *?:*, também conhecido como o operador *elvis*, avalia uma expressão, mas caso o valor seja *null* o resultado é um valor alternativo. Ou seja, a expressão *a?:b* é equivalente a *if(a!=null) a else b*. Por exemplo, a seguinte instrução apresenta o comprimento da *string* ou 0.

```
var maybeLine: String? = ...
println(maybeLine?.length?:0) // Parâmetro é do tipo Int
// O mesmo que:
println( if (maybeLine!=null) maybeLine.length else 0 )
```

Usando a função *dirOfDelta*, mas admitindo que caso não exista uma direção para um determinado deslocamento, queremos a direção UP, podemos usar o operador *?:* da seguinte forma:

```
val delta: Point = ...
val dir: Dir = dirOfDelta( delta ) ?: Dir.UP
dir.show()
```

Operador *!!*

O operador *!!* transforma um tipo anulável e não anulável, mas caso o valor seja *null*, é lançada a exceção *NullPointerException* (*NPE*) e o programa termina abruptamente.

É muito raro fazer sentido utilizar este operador. Em vez disso deve ser usado um dos outros operadores fazendo com que o código tenha um comportamento alternativo quando o valor é *null*.

Este operador só deve ser usado quando, apesar do tipo ser anulável, temos a certeza que o valor não será *null*. Caso seja *null*, consideramos um erro e até é recomendável que o programa termine assinalando tal facto com uma *NPE*.

Capítulo 8

Listas

As listas são coleções ordenadas de elementos do mesmo tipo, com acesso por índice.

Os índices são números inteiros que representam a posição de cada elemento. Uma lista com n elementos tem índices de 0 a $n - 1$.

A lista pode ser vazia ou ter vários elementos, podendo haver repetições.

Por exemplo, uma frase é uma lista de palavras, a ordem delas é importante e podem existir palavras repetidas numa frase.

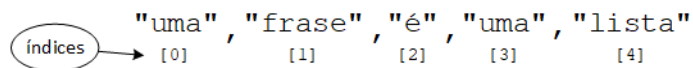


Figura 8.1: Representação de uma lista de palavras

Cada lista é do tipo `List<T>`, em que T é o tipo de cada elemento contido na lista. Por exemplo, uma lista de valores inteiros é do tipo `List<Int>`.

Designam-se como **genéricos**, os tipos que têm parâmetros formais com outros tipos. Em *Kotlin*, estes parâmetros são indicados entre `< e >` (parêntesis angulares). Assim, diz-se que `List` é um tipo genérico e `List<Int>` é um tipo parametrizado.

A função `listOf()`, da biblioteca do *Kotlin*, retorna uma lista, do tipo parametrizado `List<T>`, com os elementos passados como parâmetros, sendo T o tipo dos argumentos passados à função.

```
val phrase: List<String> = listOf("uma", "frase", "é", "uma", "lista")
```

Como a inferência de tipos do *Kotlin* também pode ser utilizada para tipos parametrizados, podemos omitir o tipo na declaração anterior.

```
val phrase = listOf("uma", "frase", "é", "uma", "lista")
```

8.1 Operadores e propriedades das listas

```
phrase[ phrase.size-1 ]
```

A propriedade `size`, do tipo `Int`, tem o número de elementos contidos na lista.

O operador `[]` (indexação) permite obter cada elemento pelo seu índice.

Usando estas características, é possível escrever todos os elementos de uma lista com um `for`.

```
for(i in 0..phrase.size-1)
    print(phrase[i]+' ' ) //Escreve: uma frase é uma lista
```

A propriedade `indices` é o intervalo dos índices dos elementos. Ou seja, `1.indices` é equivalente a 0 até `1.size`. Usando a propriedade `indices`, o `for` anterior pode ser reescrito para:

```
for(i in phrase.indices) print(phrase[i]+' ')
```

Como as listas são iteráveis, tal como os intervalos e as *strings*, a instrução `for` pode ser usada para percorrer diretamente todos os elementos da lista.

```
for(word in phrase) print(word+' ' ) // Ecreve: uma frase é uma lista
```

A função de conversão `toString()` aplicada a uma lista retorna uma *string* delimitada por parêntesis retos com os elementos separados por uma vírgula e um espaço. Por exemplo:

```
val txt = phrase.toString()
println(txt) //Escreve: [uma, frase, é, uma, lista]
```

Como a função de conversão `toString()` é chamada implicitamente quando é necessário escrever textualmente qualquer valor, com as funções `print()` ou `println()` e também nas *string* embutidas. É possível apresentar os elementos de uma lista da seguinte forma:

```
println(phrase) //Escreve: [uma, frase, é, uma, lista]
println("phrase=$phrase") //Escreve: phrase=[uma, frase, é, uma, lista]
```

As listas são imutáveis, ou seja, não é possível alterar os seus elementos. No entanto, podemos produzir outra lista como resultado de uma operação numa lista.

O operador `+` adiciona duas listas ou uma lista com um valor, produzindo uma nova lista.

```
val phrase1 = phrase + listOf("de", "palavras")
println(phrase1) // [uma, frase, é, uma, lista, de, palavras]
println(phrase + "fim.") // [uma, frase, é, uma, lista, fim.]
```

O operador `-` permite subtrair duas listas ou subtrair um valor de uma lista, produzindo uma lista sem os elementos subtraídos.

```
val phrase2 = phrase - listOf("lista", "frase")
println(phrase2) // [uma, é, uma]
println(phrase - "é") // [uma, frase, uma, lista]
```

Os operadores `in` e `!in` são utilizados para saber se um elemento faz parte da lista. São aplicados entre um valor e uma lista e produzem um valor lógico.

```
if ("uma" in phrase) println("Existe") // Existe
if ("ab" !in phrase) println("Não existe") // Não existe
```

O programa da listagem 8.1 converte um número em numeração romana para um número em numeração decimal. Para tal, é usada a lista `romanLetters`, do tipo `List<Char>`, com as letras usadas na numeração romana e é usada a lista `romanValues`, do tipo `List<Int>`, com os valores correspondentes a cada letra.

Nesta versão do programa, são ignoradas as letras que não pertençam à numeração romana, cada letra pode ser repetida mais do que 3 vezes e não são consideradas as subtrações. Por exemplo, `XXIX` é considerado 31 e não 29, `XIIIIII` é considerado 14 porque o `A` é ignorado.

A função `letterToValue()` retorna o valor correspondente à letra passada como parâmetro. Procura a letra na lista `romanLetters` e retorna o valor da lista `romanValues` no mesmo índice onde encontrou a letra, retornando 0 (zero) caso a letra não pertença à numeração romana.

A função `romanToInt()` retorna o valor correspondente ao número romano recebido como parâmetro numa `String`, fazendo o soma de todos os valores correspondentes de cada letra.

```
val romanLetters = listOf('M', 'D', 'C', 'L', 'X', 'V', 'I')
val romanValues = listOf(1000, 500, 100, 50, 10, 5, 1)

fun letterToValue(letter: Char): Int {
    for(idx in romanLetters.indices)
        if (letter == romanLetters[idx]) // Letra encontrada?
            return romanValues[idx] // Retorna valor da letra
    return 0 // Letra não encontrada
}

fun romanToInt(roman: String): Int {
    var number = 0
    for(r in roman)
        number += letterToValue(r)
    return number
}
```

```
fun main() {
    print("Número romano? ")
    val romanNumber = readln().trim().uppercase()
    val number = romanToInt(romanNumber)
    println("$romanNumber = $number")
}
```

Listagem 8.1: Roman.kt

Uma possível utilização deste programa seria:

```
Número romano? MMXXI ↵
MMXXI = 2021
```

Sem alterar nada no resto do programa, a primeira linha poderia ser substituída por:

```
val romanLetters = "MDCLXVI"
```

passando `romanLetters` a ser do tipo `String` em vez de `List<Char>`. De facto, existem muitas semelhanças entre os tipos `String` e `List<Char>`. Das características descritas até ao momento, a única diferença é que a propriedade que indica o número de elementos é `length` na `String` e `size` em `List<Char>`.

8.2 Operações sobre listas

```
list.sorted().last()
```

Existem várias funções que se podem aplicar a qualquer lista. Por exemplo, a função `indexOf()` procura na lista o elemento indicado como parâmetro e retorna o índice onde o encontrou, ou retorna `-1` se não encontrou.

Usando a função `indexOf()`, a função `letterToValue()` da listagem 8.1 pode ser reescrita para:

```
fun letterToValue(letter: Char): Int {
    val idx = romanLetters.indexOf(letter)
    return if (idx == -1) 0 else romanValues[idx]
}
```

Existem muitas operações sobre listas disponíveis na biblioteca. As mais comuns podem ser divididas em operações de consulta, de filtragem e de transformação.

O seguinte troço de código demonstra a aplicação de diversas funções sobre a lista `digits`.

```
val digits = listOf(3,1,4,1,5,9,2,6) // Primeiros 8 algarismos de PI
// Consulta
println(digits.isEmpty()) // false
println(digits.first()) // 3
println(digits.last()) // 6
println(digits.indexOf(5)) // 4
println(digits.lastIndexOf(1)) // 3
// Filtragem
println(digits.subList(2,6)) // [4, 1, 5, 9]
println(digits.drop(2)) // [4, 1, 5, 9, 2, 6]
println(digits.dropLast(2)) // [3, 1, 4, 1, 5, 9]
println(digits.take(3)) // [3, 1, 4]
println(digits.takeLast(3)) // [9, 2, 6]
println(digits.distinct()) // [3, 1, 4, 5, 9, 2, 6]
// Transformação
println(digits.reversed()) // [6, 2, 9, 5, 1, 4, 1, 3]
println(digits.sorted()) // [1, 1, 2, 3, 4, 5, 6, 9]
```

A função `isEmpty()` retorna um valor lógico que indica se a lista está vazia. Ou seja, `l.isEmpty()` é equivalente a `l.size==0`.

As funções `first()` e `last()` retornam o primeiro e o último elemento, respetivamente. Ou seja, a expressão `l.first()` é equivalente a `l[0]` e `l.last()` é equivalente a `l[l.size-1]`.

As funções `indexOf()` e `lastIndexOf()` retornam o índice onde encontram o elemento indicado, ou `-1` caso não exista na lista, mas `lastIndexOf()` procura pela ordem inversa.

A função `subList()` retorna uma sub-lista com os elementos entre os índices indicados, incluindo o primeiro mas excluindo o último.

As funções `drop()` e `dropLast()` retornam uma sub-lista sem o número de elementos indicados, no início ou no fim, respetivamente. Ou seja, `l.drop(n)` é equivalente a `l.subList(n,l.size)` e `l.dropLast(n)` é equivalente a `l.subList(0,l.size-n)`.

As funções `take()` e `takeLast()` retornam uma sub-lista só com o número de elementos indicados, no início ou no fim, respetivamente. Ou seja, `l.take(n)` é equivalente a `l.dropLast(l.size-n)` e `l.takeLast(n)` é equivalente a `l.drop(l.size-n)`.

A função `distinct()` retorna outra lista com os elementos pela mesma ordem mas sem repetições.

A função `reversed()` retorna outra lista com os elementos por ordem inversa.

A função `sorted()` retorna outra lista com os elementos por ordem crescente do seu valor.

Quase todas as funções descritas, aplicáveis a qualquer lista, também podem ser aplicadas a qualquer tipo iterável, como é o caso de `String`. Neste caso, não existem as funções `sorted()` e `distinct()` e em vez da função `subList()` existe a função `substring()`. O seguinte troço de código demonstra a aplicação destas funções à *string* `str`.

```
val str = "Kotlin"
println(str.isEmpty())           // false
println(str.first())             // K
println(str.last())              // n
println(str.indexOf('t'))        // 2
println(str.lastIndexOf('i'))    // 4
println(str.substring(2,4))      // tl
println(str.drop(2))              // tlin
println(str.take(2))              // Ko
println(str.reversed())          // niltoK
```

Na biblioteca existem ainda algumas funções que só podem ser aplicadas a iteráveis de valores numéricos. O troço de código seguinte demonstra a utilização das funções `min()`, `max()`, `sum()` e `average()`.

```
val numbers = listOf(12, 4, 9, 2, 15, 8)
println(numbers.min())           // 2
println(numbers.max())           // 15
println(numbers.sum())           // 50
println(numbers.average())       // 8.333333333333334
val range = 0..22 step 2
println(range.min())             // 0
println(range.max())             // 22
println(range.sum())             // 132
println(range.average())         // 11.0
```

8.3 Filtragens

```
list.filter { selecção }
```

A função `filter()` permite fazer filtragens customizáveis. É aplicável a qualquer iterável e recebe como parâmetro a função que seleciona cada elemento a passar na filtragem. Retorna uma nova lista contendo apenas os elementos selecionados.

A função de seleção recebe como parâmetro cada elemento da lista original e retorna o valor lógico `true` se o elemento for selecionado ou `false` caso não passe no filtro.

O seguinte troço de código, chama a função `filter()` aplicada à lista `digits`, que é do tipo `List<Int>`, passando como parâmetro uma expressão lambda do tipo `(Int)->Boolean`. A lista resultante fica apenas com os elementos que são números pares.

```
val digits = listOf(3,1,4,1,5,9,2,6) // Primeiros 8 algarismos de PI
val evenDigits = digits.filter( { e:Int -> e%2==0 } )
println(evenDigits) // [4, 2, 6]
```

Usando as simplificações descritas em 6.5.1 a chamada à função `filter()` pode ficar apenas:

```
val evenDigits = digits.filter { it%2==0 }
```

O seguinte troço de código apresenta as palavras com mais dos que 3 letras da lista `phrase`.

```
val phrase = listOf("uma", "frase", "é", "uma", "lista")
println( phrase.filter { it.length>3 } )    // [frase, lista]
```

Uma implementação simplificada da função `filter` para listas de *strings* poderia ser:

```
fun List<String>.filter(condition: (String)->Boolean): List<String> {
    var result: List<String> = listOf()
    for(element in this)
        if (condition(element)) result = result + element
    return result
}
```

A função `filterIndexed` da biblioteca recebe uma função de seleção com dois parâmetros, o índice de cada elemento e o elemento. Por exemplo, a seguinte código apresenta as quatro primeiras palavras da frase com menos do que quatro letras.

```
val res = phrase.filterIndexed { idx, word -> idx<4 && word.length<4 }
println(res) // [uma, é, uma]
```

A funções de fitragem também podem ser aplicadas as strings e a intervalos, mas no segundo caso retornam listas.

O seguinte código apresenta as vogais minúsculas de "Kotlin" e os valores múltiplos de 3 no intervalo de 1 até 10.

```
println("Kotlin".filter { it in "aeiou" } ) // oi
println((1..10).filter { it % 3 == 0 } ) // [3, 6, 9]
```

8.4 Transformações

`list.map { transformação }`

Para fazer transformações (mapeamentos) customizáveis deve ser usada a função `map()`. Esta é aplicável a qualquer iterável e recebe como parâmetro a função de transformação que é chamada para cada elemento e retorna uma nova lista contendo os elementos transformados.

A função de transformação recebe como parâmetro cada elemento da lista original e retorna um valor para cada elemento da lista a retornar.

A lista retornada tem a mesma dimensão da lista original mas pode ser de um tipo diferente.

O seguinte troço de código, chama a função `map()` aplicada à lista `phrase`, que é do tipo `List<String>`, passando como parâmetro uma expressão lambda do tipo `(String)->Int`. A lista resultante é do tipo `List<Int>`, ficando com os comprimentos das palavras da frase.

```
val phrase = listOf("uma", "frase", "é", "uma", "lista")
println(phrase.map { it.length } ) // [3, 5, 1, 3, 5]
```

É comum chamar em cascata várias funções sobre listas. Por exemplo, para apresentar a média dos comprimentos das palavras de `phrase`, podemos fazer:

```
println(phrase.map { it.length }.average()) // 3.4
```

Análoga à função `filterIndexed`, a função `mapIndexed` da também recebe uma função de transformação com dois parâmetros. Por exemplo, a seguinte expressão faz a soma dos comprimentos das palavras multiplicado pela respetiva posição (índice).

```
println(phrase.mapIndexed { i,w -> w.length * i }.sum() ) // 36
```

A função `map()` também pode ser aplicada a *strings* e a intervalos, mas retorna uma lista. Por exemplo, o seguinte troço de código apresenta o código de cada carácter usado no texto "Kotlin" e o somatório dos códigos de todas as letras maiúsculas.

```
println("Kotlin".map{ it.code }) // [75, 111, 116, 108, 105, 110]
println(('A'..'Z').map { it.code }.sum()) // 2015
```

A função `romanToInt()` da listagem 8.1, que anteriormente foi escrita assim:

```
fun romanToInt(roman :String): Int {
    var number = 0
    for(r in roman) number += letterToValue(r)
    return number
}
```

pode ser reescrita, usando as funções `map()` e `sum()`, para:

```
fun romanToInt(roman: String) = roman.map { letterToValue(it) }.sum()
```

Esta nova versão, faz uma lista com os valores de cada letra e retorna a soma desses valores.

Uma implementação simplificada da função `map` para transformar uma lista de *strings* numa lista de inteiros, poderia ser:

```
fun List<String>.map(transform: (String)->Int): List<Int> {
    var result: List<Int> = listOf()
    for(element in this)
        result = result + transform(element)
    return result
}
```

8.5 Iterações

```
list.forEach { operação }
```

Para percorrer todos os elementos de uma lista ou de qualquer coleção iterável, podem ser usadas diferentes formas de iteração.

As coleções (listas ou qualquer iterável) fornecem acesso sequencial aos seus elementos sem expor a estrutura interna.

A forma mais elementar de iteração é usando o conceito de iterador. Primeiro obtém-se o iterador chamando a função `iterator()` da coleção, depois, enquanto a função `hasNext()` do iterador retornar `true`, obtém-se o próximo elemento chamando a função `next()` do iterador. O troço de código seguinte mostra esta forma de iterar.

```
val collection = listOf("uma", "frase", "é", "uma", "lista")
val iter = collection.iterator() // Pedir iterador
while ( iter.hasNext() ) {      // Existe o próximo ?
    val element = iter.next()    // Obter o próximo
    println( element )
}
```

Se a primeira linha for substituída por:

```
val collection = "Kotlin"
```

ou por:

```
val collection = 1..9
```

as restantes linhas seriam iguais para percorrer todos os elementos da respetiva coleção.

Usando um ciclo `for` para obter cada elemento da coleção o código gerado é equivalente. É obtido implicitamente o iterador e são usadas implicitamente as funções `hasNext()` e `next()` do iterador.

```
for(element in collection)
    println( element )
```

Com o mesmo propósito existe a função `forEach()` que recebe como parâmetro uma função para realizar uma operação para cada elemento. A função da operação recebe cada elemento como parâmetro.

```
collection.forEach { println(it) }
```

8.5.1 Iteração com índice

Caso a coleção tenha acesso a cada elemento por índice (que não é o caso dos intervalos) é possível usar um ciclo para percorrer os índices e depois obter cada elemento.

```
val collection = listOf("uma", "frase", "é", "uma", "lista")
for(idx in collection.indices) {
    val element = collection[idx]
    println("[ $idx ] = $element")
}
```

Para os intervalos, como não têm acesso por índice, não é possível usar esta forma de iteração. Mas para *strings* é possível, percorrendo o intervalo `0..collection.length-1`.

```
val collection = "Kotlin"
for(idx in 0 until collection.length) {
    val element = collection[idx]
    println("[ $idx ] = $element")
}
```

Na biblioteca do *Kotlin* também existe a função `forEachIndexed()` em que a função passada como argumento recebe o índice e o elemento como parâmetros. A função `forEachIndexed()` está disponível para qualquer iterável, mesmo para intervalos.

```
collection.forEachIndexed { idx, element ->
    println("[ $idx ] = $element")
}
```

Uma implementação simplificada da função `forEachIndexed` para percorrer uma lista de *strings*, poderia ser:

```
fun List<String>.forEachIndexed(operation: (Int, String)->Unit) {
    var idx = 0
    for(element in this)
        operation(idx++, element)
}
```

8.6 Listas mutáveis

```
mutLlist.add( elemento )
```

Enquanto que as listas imutáveis, do tipo `List<T>`, são usadas para armazenar elementos que não mudam após a criação, as listas mutáveis, do tipo `MutableList<T>`, permitem a adição, remoção e atualização de elementos.

Para criar uma lista mutável, podemos usar a função `mutableListOf()`. Essa função cria uma lista iniciada com elementos indicados.

```
val mutPhrase: MutableList<String> = mutableListOf(
    "Uma", "frase", "é", "lista", "de"
)
println(mutPhrase.size) // 5
println(mutPhrase) // [Uma, frase, é, lista, de]
```

Posteriormente podem ser adicionados elementos usando a função `add()` para acrescentar no fim da lista ou inserir num índice específico, deslocando os seguintes.

```
mutPhrase.add("palavras")
mutPhrase.add(3, "uma")
println(mutPhrase.size) // 7
println(mutPhrase) // [Uma, frase, é, uma, lista, de, palavras]
```

Podem ser removidos elementos usando a função `remove()` ou `removeAt()`.

```
mutPhrase.remove("lista")
mutPhrase.removeAt(2)
mutPhrase.removeAt(3)
println(mutPhrase) // [Uma, frase, uma, palavras]
```

Os elementos podem ser alterados usando a função `set()` ou simplesmente usando a afetação.

```
mutPhrase.set(2, "são")
mutPhrase[3] = "strings"
mutPhrase.add(3, "várias")
println(mutPhrase) // [Uma, frase, são, várias, strings]
```

Existe conversão direta de listas mutáveis para listas imutáveis, mas o contrário não é possível.

```
// OK: MutableList<String> é uma List<String>
val phrase: List<String> = mutPhrase

// ERRO: List<String> não é uma MutableList<String>
val mutable: MutableList<String> = phrase
```

A função `toMutableList()` cria uma nova lista mutável a partir de uma lista imutável.

```
val phrase = listOf("uma", "frase", "é", "uma", "lista")
val mutable = phrase.toMutableList()
mutable.add("mutável.")
println(mutable) // [uma, frase, é, uma, lista, mutável.]
println(phrase) // [uma, frase, é, uma, lista]
```

Em *Kotlin*, prefira sempre usar listas do tipo `List<T>` em vez de `MutableList<T>` sempre que possível, para garantir a imutabilidade dos dados. Listas imutáveis são mais seguras e menos propensas a erros, e também promovem um código mais simples e previsível.

8.7 Criação de listas iniciadas

```
val list = List(size) { init }
```

Por vezes é necessário criar listas imutáveis com uma sequência lógica de valores. Por exemplo, uma lista com todas as letras maiúsculas do alfabeto.

```
val alphabet = listOf(
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
)
println(alphabet.size) // 26
val text = alphabet.joinToString(separator = "")
println(text) // ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Por curiosidade, a função `joinToString()` retorna uma *string* com todos os elementos de uma coleção. Esta função tem vários parâmetros, sendo um deles o `separator` que define o texto que fica a separar os elementos que por omissão é ", ".

Obviamente que, a melhor forma de criar uma lista de valores que seguem uma sequência lógica não é com a função `listOf()`, principalmente se a lista for de muitos elementos. Neste caso, pode ser definido um intervalo e fazer uma transformação (`map`).

```
val alphabet = (0..25).map { 'A' + it }
```

Para não ficar a "constante mágica" 25, ficaria melhor:

```
val alphabet = (0..'Z'-'A').map { 'A' + it }
```

Uma alternativa ainda mais evidente seria a conversão direta de um intervalo de letras na lista.

```
val alphabet = ('A'..'Z').toList()
```

Mas também é possível criar a lista diretamente usando a função de construção `List()`. Esta função recebe dois parâmetros, o número de elementos da lista (`size`) e uma função de iniciação de cada elemento que recebe o número de ordem do elemento a iniciar (de 0 a `size-1`)

```
val alphabet = List('Z'-'A'+1) { 'A' + it }
```


A alternativa que deve ser evitada, mas que por vezes é necessária, é a utilização de uma lista mutável auxiliar que é preenchida e depois é convertida em imutável.

```
val alphaAux = mutableListOf<Char>()
for (letter in 'A'..'Z') {
    alphaAux.add(letter)
}
val alphabet: List<Char> = alphaAux
```

Na biblioteca existe a função `buildList()` que usa uma lista mutável internamente e que recebe como parâmetro uma função extensão da lista mutável que é chamada para iniciar a lista.

```
val alphabet = buildList {
    for (letter in 'A'..'Z') add(letter)
}
```

Uma implementação simplificada da função `buildList()` para criar uma lista de `Char`, seria:

```
fun buildList(init: MutableList<Char>().() -> Unit): List<Char> {
    val result = mutableListOf<Char>()
    result.init()
    return result
}
```

Para tornar mais evidente a necessidade da criação de listas iniciadas, vamos supor que era necessário criar a lista de todos os pontos possíveis num sistema de coordenadas de 10 por 10.

```
data class Point(val x: Int, val y: Int)

val allPoints = listOf(
    Point(0,0), Point(0,1), ... Point(0,9),
    Point(1,0), Point(1,1), ... Point(1,9),
    ...
    Point(9,0), Point(9,1), ... Point(9,9)
)
```

Em vez desta iniciação, poderia ser realizada a seguinte:

```
val allPoints = buildList {
    for (x in 0 until 10)
        for (y in 0 until 10)
            add(Point(x, y))
}
```

Ou então simplesmente:

```
val allPoints = List(10*10) { Point(x= it/10, y= it%10) }
```

8.8 Arrays

`array[index] = value`

Os arrays são coleções sequenciais com dimensão fixa (imutável) mas os elementos são modificáveis (mutáveis). Enquanto as listas mutáveis, do tipo `MutableList<T>`, permitem a adição, remoção e atualização de elementos, os arrays `Array<T>` só permitem a atualização dos elementos.

Para criar um array, podemos usar a função `arrayOf()` ou a funções construtora `Array()`.

A função `arrayOf()` cria um array com elementos indicados em que a dimensão do array é igual ao número elementos indicados. Por exemplo, o seguinte troços de código cria um array de palavras e depois altera a segunda palavra.

```
val arrPhrase: Array<String> = arrayOf("PG", "LEIC", "DEETC", "ISEL")
println(arrPhrase.size) // 4
println(arrPhrase[1]) // LEIC
arrPhrase[1] = "Informática"
println(arrPhrase[1]) // Informática
```

A função `Array()` recebe dois parâmetros, a dimensão do array e a função de iniciação de cada elemento. Por exemplo, para criar um array com dez contadores iniciados a zero, seria:

```
val counters = Array(10) { 0 }
println(counters[3]) // 0
counters[3]++
println(counters[3]) // 1
```

Cada array tem um espaço de memória contígua com dimensão fixa para armazenar os elementos. A figura 8.2 é uma representação do array `arrPhrase` do exemplo anterior.

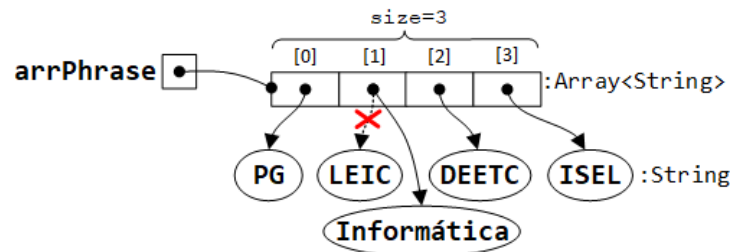


Figura 8.2: Representação em memória de um array

A função `main()`, que é a primeira função chamada na execução de um programa, pode ter um parâmetro do tipo `Array<String>` onde são passadas as palavras que foram indicadas na linha de comando. Por exemplo, dado o seguinte programa da listagem 8.2 que estaria no ficheiro fonte `Args.kt`.

```
fun main(args: Array<String>) {
    for(i in args.indices) {
        println("args[$i] = ${args[i]}")
    }
}
```

Listagem 8.2: `Args.kt`

A execução do programa com o comando: `kotlin ArgsKt PG ISEL LEIC`, dará o seguinte resultado.

```
C:\ISEL\PG>kotlin ArgsKt PG ISEL LEIC ↵
args[0] = PG
args[1] = ISEL
args[2] = LEIC
```

A tabela 8.1 resume e compara as características principais de `List<T>`, `Array<T>` e `MutableList<T>`.

Tabela 8.1: `List`, `Array` e `MutableList`

Tipo	Dimensão	Elementos
<code>List<T></code>	Fixa	Imutáveis
<code>Array<T></code>	Fixa	Mutáveis
<code>MutableList<T></code>	Variável	Mutáveis

8.9 Reconhecimento e conversão de numeração romana

Vamos reformular o programa apresentado na listagem 8.1, que convertia um número em numeração romana para um número em numeração decimal, mas com algumas simplificações. Agora, o programa usará todas as regras da numeração romana, passará a receber vários números indicados na linha de comando e cada número é validado antes de ser convertido.

A função `main()` será reescrita para:

```
fun main(numbers: Array<String>) {
    for(r in numbers) {
        val roman = r.trim().uppercase()
        if (isValidRoman(roman)){
            val number = romanToInt(roman)
            println("$roman = $number")
        } else
            println("$roman não é válido em numeração romana")
    }
}
```

Listagem 8.3: Função `main()` de `Roman.kt`

Se o programa for executado com o comando `kotlin RomanKt MMXXIII XXVVI MCMLL` produzirá o resultado:

```
MMXXIII = 2023
XXVVI não é válido em numeração romana
MCMLL não é válido em numeração romana
```

A função `isValidRoman()` verifica se o número é válido e a função `romanToInt()` retorna o valor correspondente ao número, assumindo que é válido.

Um número válido, em numeração romana, cumpre o diagrama sintático da figura 8.3, onde se identifica uma repetição estrutural nas centenas (C), nas dezenas (X) e nas unidades (I).

1. Um número, pode ter no início zero ou vários M, mas não podem ser mais do que 3.
2. Depois pode existir, ou não, um só D seguido de 0 até 3 C ou em alternativa as combinações para realizar as subtrações CM ou CD.
3. Depois pode existir um só L seguido de 0 até 3 X ou em alternativa XC ou XL.
4. Finalmente, pode existir um só V seguido de 0 até 3 I ou em alternativa IX ou IV.

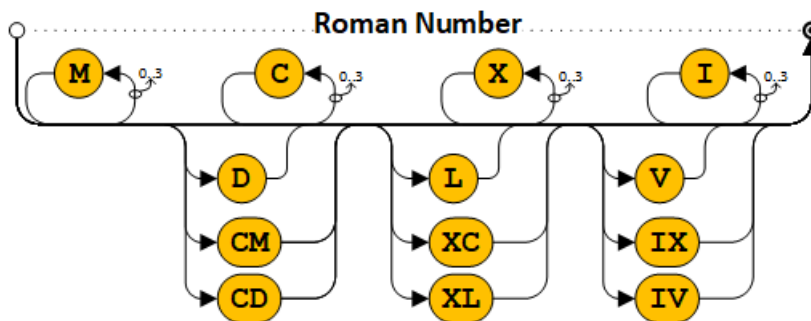


Figura 8.3: Diagrama sintático de um número em numeração romana

Por exemplo: `MMDDXXII` não é válido porque o D não pode ser repetido; `MCCCCVI` não é válido porque o C não pode ser repetido mais do que 3 vezes; `MCCDXI` não é válido porque depois de CC já não pode existir um D; `MCMLXXIX` é válido, representa o valor 1979 e cumpre o percurso `M→CM→L→X→X→IX`.

Seguindo o diagrama da figura 8.3, uma implementação da função `isValidRoman()` poderá a da listagem 8.4.

Esta função usa a variável `i` com índice do símbolo corrente em análise da `string roman`. Cada vez que reconhece um símbolo válido avança o índice. No final, o número é reconhecido com válido se `i` for igual à dimensão da `string roman`, ou seja, não sobram símbolos por reconhecer.

As funções `isChar()`, `isSubtract()` e `repeatChar()` são funções locais a `isValidRoman` para evitar a repetição de código e, como tal, podem usar diretamente a variável `i` e o parâmetro `roman`.

A função `isChar()` verifica se símbolo em análise existe e é igual ao indicado como parâmetro. A função `isSubtract()` verifica se os próximos dois símbolo em análise existem e são iguais aos indicados como parâmetro. A função `repeatChar()` verifica se os próximos símbolos são zero até três símbolos iguais ao parâmetro e, neste caso, vai avançando o índice corrente.

```
fun isValidRoman(roman: String): Boolean {
    var i = 0
    fun isChar(c: Char) =
        i < roman.length && roman[i] == c
    fun isSubtract(cc: String) =
        i < roman.length-1 && roman[i] == cc[0] && roman[i+1] == cc[1]
    fun repeatChar(c: Char): Boolean {
        repeat(3) { if (isChar(c)) ++i else return true }
        return false
    }

    if (!repeatChar('M')) return false
    if (isSubtract("CM") || isSubtract("CD")) i+=2
    else {
        if (isChar('D')) ++i
        if (!repeatChar('C')) return false
    }
    if (isSubtract("XC") || isSubtract("XL")) i+=2
    else {
        if (isChar('L')) ++i
        if (!repeatChar('X')) return false
    }
    if (isSubtract("IX") || isSubtract("IV")) i+=2
    else {
        if (isChar('V')) ++i
        if (!repeatChar('I')) return false
    }
    return i == roman.length
}
```

Listagem 8.4: Primeira versão da função `isValidRoman()` de `Roman.kt`

Apesar de evitar a repetição de código com as funções internas auxiliares, a validação do grupo das centenas (com os símbolos M, D e C), a validação do grupo das dezenas (com os símbolos C, L e X) e a validação do grupo das unidades (com os símbolos X, V e I), ainda faz alguma repetição da mesma estrutura de código.

Podemos evitar esta repetição estrutural fazendo uma lista com os símbolos de cada grupo, passando a ter a implementação da listagem 8.5

```
fun isValidRoman(roman: String): Boolean {
    var i = 0
    fun isChar(c: Char) =
        i < roman.length && roman[i] == c
    fun isNextChar(c: Char) =
        i < roman.length-1 && roman[i+1] == c
    fun repeatChar(c: Char): Boolean {
        repeat(3) { if (isChar(c)) ++i else return true }
        return false
    }

    if (!repeatChar('M')) return false
    listOf("MDC", "CLX", "XVI").forEach { g ->
        if (isChar(g[2]) && (isNextChar(g[0]) || isNextChar(g[1]))) i+=2
        else {
            if (isChar(g[1])) ++i
            if (!repeatChar(g[0])) return false
        }
    }
    return i == roman.length
}
```

Listagem 8.5: Segunda versão da função `isValidRoman()` de `Roman.kt`

Mas também é possível identificar as letras de cada grupo percorrendo a lista `romanLetters`, que já estava declarada na versão anterior, avançando de 2 em 2 no índice das letras (`il`)

```
val romanLetters = listOf('M', 'D', 'C', 'L', 'X', 'V', 'I')
```

Com `il==0` fica `romanLetters[il]=='M'` (milhares), com `il==2` `romanLetters[il]=='C'` (centenas), `il==4` dezenas e `il==6` unidades.

Neste caso, podemos generalizar também o grupo dos milhares verificando os casos que não ocorrem nesse grupo (`il>0`).

```
fun isValidRoman(roman: String): Boolean {
    var i = 0
    fun isChar(idx: Int) =
        i < roman.length && roman[i] == romanLetters[idx]
    fun isNextChar(idx: Int) =
        i < roman.length-1 && roman[i+1] == romanLetters[idx]

    for(il in romanLetters.indices step 2) {
        if (il>0 && isChar(il) && (isNextChar(il-1) || isNextChar(il-2))) i
            +=2
        else {
            if (il>0 && isChar(il-1)) ++i
            var reps = 0
            while(isChar(il)) {
                ++i;
                if (++reps > 3) return false
            }
        }
    }
    return i == roman.length
}
```

Listagem 8.6: Versão final da função `isValidRoman()` de `Roman.kt`

A função `romanToInt()` usa as listas `romanLetters` e `romanValues` que já estavam declaradas:

```
val romanLetters = listOf('M', 'D', 'C', 'L', 'X', 'V', 'I')
val romanValues = listOf(1000, 500, 100, 50, 10, 5, 1)
```

e retorna o valor representado pelo número romano, assumindo que é válido.

```
fun romanToInt(roman: String): Int {
    var lastValue = 0
    var number = 0
    roman.reversed()
        .map { romanValues[romanLetters.indexOf(it)] }
        .forEach { value ->
            number += if (value >= lastValue) value else -value
            lastValue = value
        }
    return number
}
```

Listagem 8.7: Versão final da função `romanToInt()` de `Roman.kt`

Esta função percorre as letras do número romano da direita para a esquerda e vai acumulando o somatório de cada símbolo na variável `number`, mas cada vez que o valor dos símbolo anterior (`lastValue`) é menor que o valor do símbolo atual, subtrai em vez de somar.

Por exemplo, para `roman=="MCMXV"` faz: `number = +5 +10 +1000 -100 +1000 ⇒ 1915`.