

Programação Funcional em OCaml

José Romildo Malaquias

BCC222: Programação Funcional

Universidade Federal de Ouro Preto
Departamento de Computação

2015–2

1	OCaml	1-1
1.1	A linguagem OCaml	1-1
1.1.1	História de OCaml	1-1
1.1.2	Características de OCaml	1-2
1.1.3	Por que OCaml?	1-2
1.1.4	Usos e Aplicações	1-2
1.2	O Sistema OCaml	1-3
1.2.1	Ferramentas da distribuição OCaml	1-3
1.2.2	O ambiente interativo	1-3
1.2.3	Pacotes e bibliotecas	1-3
1.2.4	Editores	1-4
1.3	Instalação das ferramentas de desenvolvimento em OCaml	1-4
1.3.1	Instalação do OCaml no Windows: Instalador oficial	1-4
1.3.2	Instalação do OCaml no Ubuntu	1-6
1.3.3	Instalação do ambiente interativo no Eclipse	1-6
1.4	Usando o ambiente interativo	1-9
2	Valores, Tipos e Funções	2-1
2.1	Considerações léxicas	2-1
2.1.1	Comentários	2-1
2.1.2	Palavras reservadas	2-1
2.1.3	Identificadores	2-2
2.2	Alguns tipos básicos	2-2
2.2.1	Números inteiros: tipo <code>int</code>	2-2
2.2.2	Números reais: tipo <code>float</code>	2-3
2.2.3	Caracteres: tipo <code>char</code>	2-4
2.2.4	Cadeias de caracteres: tipo <code>string</code>	2-5
2.2.5	Valores lógicos: tipo <code>bool</code>	2-5
2.2.6	Tupla vazia: tipo <code>unit</code>	2-5
2.3	Constantes	2-6
2.4	Aplicação de função	2-6
2.5	Definindo variáveis e funções globais	2-8
2.6	Tipos função	2-9
2.7	Checagem de tipos	2-9
2.8	Inferência de tipos	2-10
2.9	Exercícios	2-11
3	Expressão Condicional	3-1
3.1	Expressão condicional	3-1
3.2	Definição de função com expressão condicional	3-2
4	Definições Locais	4-1
4.1	Expressão <code>let</code>	4-1
4.2	Exercícios	4-4
5	Programas Interativos	5-1
5.1	Interação com o mundo	5-1
5.2	Expressão sequência	5-1
5.2.1	Ignorando o resultado de uma expressão	5-2
5.2.2	Blocos de programa	5-2
5.3	Estrutura de um programa em OCaml	5-2
5.4	Canais de entrada e saída	5-5
5.5	Funções de saída na saída padrão	5-6
5.6	Funções de saída na saída de erro padrão	5-6

5.7	Funções de entrada na entrada padrão	5-6
5.8	Exercícios	5-7
6	Funções Recursivas	6-1
6.1	Recursividade	6-1
6.2	Recursividade mútua	6-5
6.3	Recursividade de cauda	6-6
6.4	Vantagens da recursividade	6-8
6.5	Exercícios	6-9
7	Valores Aleatórios	7-1
7.1	Valores aleatórios	7-1
7.2	Exemplo: lançamento de dados	7-1
7.3	Jogo adivinha o número	7-2
7.4	Jogo craps	7-6
8	Tuplas, Listas, e Polimorfismo Paramétrico	8-1
8.1	Tuplas	8-1
8.2	Listas	8-2
8.3	Polimorfismo paramétrico	8-4
8.3.1	Operação sobre vários tipos de dados	8-4
8.3.2	Variáveis de tipo	8-4
8.3.3	Valor polimórfico	8-5
8.3.4	Instanciação de variáveis de tipo	8-5
8.4	Funções polimórficas predefinidas	8-5
9	Casamento de Padrão	9-1
9.1	Casamento de padrão	9-1
9.1.1	Casamento de padrão	9-1
9.1.2	Padrão constante	9-2
9.1.3	Padrão variável	9-2
9.1.4	Padrão curinga	9-2
9.1.5	Padrão tupla	9-2
9.1.6	Padrões lista	9-3
9.1.7	Padrões combinados	9-4
9.1.8	Padrões faixa	9-5
9.1.9	Padrões nomeados	9-5
9.2	Casamento de padrão em definições com <code>let</code>	9-6
9.3	Definição de função usando padrões	9-6
9.4	Expressão de Seleção Múltipla	9-8
9.4.1	Forma e regras de tipo da expressão <code>match</code>	9-8
9.4.2	Avaliação de expressões <code>match</code>	9-8
9.4.3	Exemplos de expressões <code>match</code>	9-9
9.4.4	Expressão <code>match</code> com guardas	9-11
9.5	Exercícios	9-12
10	Expressões Lambda	10-1
10.1	Valores de primeira classe	10-1
10.1.1	Valores de primeira classe	10-1
10.1.2	Valores de primeira classe: Literais	10-1
10.1.3	Valores de primeira classe: Variáveis	10-2
10.1.4	Valores de primeira classe: Argumentos	10-2
10.1.5	Valores de primeira classe: Resultado	10-2
10.1.6	Valores de primeira classe: Componentes	10-3
10.2	Expressão lambda	10-3
10.2.1	Expressões lambda with <code>fun</code>	10-3
10.2.2	Exemplos de expressões lambda	10-3
10.2.3	Uso de expressões lambda	10-4
10.2.4	Exercícios	10-4
10.2.5	Expressões lambda with <code>function</code>	10-5
10.3	Aplicação parcial de funções	10-6
10.3.1	Aplicação parcial de funções	10-6
10.3.2	Aplicação parcial de funções: exemplos	10-6

10.4	<i>Currying</i>	10-7
10.4.1	Funções <i>curried</i>	10-7
10.4.2	Por que <i>currying</i> é útil?	10-8
10.4.3	Convenções sobre <i>currying</i>	10-8
10.5	Utilidade de expressões lambda	10-8
10.5.1	Exercícios	10-10
11	Funções de Ordem Superior	11-1
11.1	Funções de Ordem Superior	11-1
11.2	Composição de funções	11-1
11.3	A função <code>List.filter</code>	11-2
11.4	A função <code>map</code>	11-3
11.5	<code>fold_left</code>	11-3
11.6	<code>fold_right</code>	11-4
11.7	Cupom fiscal do supermercado	11-5

Resumo

As atividades de programação serão desenvolvidas usando a linguagem OCaml (<http://www.ocaml.org/>).

Nesta aula o aluno será apresentado à linguagem OCaml, passando a conhecer suas principais características. Serão apresentados os passos necessários para instalar o sistema de programação em OCaml e as principais ferramentas utilizadas durante o processo de desenvolvimento em OCaml.

Haverá oportunidade de familiarização com o ambiente de programação em OCaml através da avaliação de expressões no ambiente interativo e compilação em *batch*.

Sumário

1.1	A linguagem OCaml	1-1
1.1.1	História de OCaml	1-1
1.1.2	Características de OCaml	1-2
1.1.3	Por que OCaml?	1-2
1.1.4	Usos e Aplicações	1-2
1.2	O Sistema OCaml	1-3
1.2.1	Ferramentas da distribuição OCaml	1-3
1.2.2	O ambiente interativo	1-3
1.2.3	Pacotes e bibliotecas	1-3
1.2.4	Editores	1-4
1.3	Instalação das ferramentas de desenvolvimento em OCaml	1-4
1.3.1	Instalação do OCaml no Windows: Instalador oficial	1-4
1.3.2	Instalação do OCaml no Ubuntu	1-6
1.3.3	Instalação do ambiente interativo no Eclipse	1-6
1.4	Usando o ambiente interativo	1-9

1.1 A linguagem OCaml

1.1.1 História de OCaml

ML é uma família muito grande de linguagens de programação que inclui Standard ML, OCaml, F#, SML#, Manticore, MetaOCaml, JoCaml, Alice ML, Dependent ML, Flow Caml, e muitas outras. Todas as linguagens ML compartilham várias características fundamentais, além de uma boa dose de sintaxe. Elas são de ordem superior, estritas, na maior parte puras, e tipadas, com tipos algébricos e outros tipos de dados. Seus sistemas de tipo são derivados de Hindley-Milner. O desenvolvimento dessas linguagens inspirou um conjunto significativo de pesquisas em Ciência da Computação e influenciou a concepção de muitas outras linguagens de programação, incluindo Haskell, Scala e Clojure, Rust, ATS e muitas outras.

OCaml¹ é uma linguagem de programação de uso geral, com ênfase na expressividade e segurança. Desenvolvida há mais de 20 anos no INRIA² por um grupo de pesquisadores líderes, OCaml beneficia-se de um dos sistemas de tipos mais avançados e suporta os estilos de programação funcional, imperativo e orientado a objetos, que facilitam o desenvolvimento de um software flexível e confiável. Usado em ambientes onde um único erro pode custar milhões e onde velocidade é importante, OCaml é apoiada por uma comunidade ativa que desenvolveu um rico conjunto de bibliotecas e irá ajudá-lo a tirar o máximo de possibilidades de OCaml.

OCaml é uma linguagem de programação moderna, de alto nível, com muitos recursos úteis. É um produto de código aberto de mais de vinte anos de pesquisa de ponta que permite o desenvolvimento rápido de software robusto, conciso e correto.

OCaml começou com a ML original criada na universidade de Edimburgo por Robin Milner, no começo da década de 1970. ML significa *Meta Language* pois ela foi feita para funcionar como a meta-linguagem do provador de teoremas LCF.

¹<http://www.ocaml.org>

²Institut National de Recherche en Informatique et en Automatique.

Os primeiros usuários de ML perceberam que ela funcionava bem para programar outras coisas, então ela foi separada do seu sistema de origem.

Várias variações de ML foram desenvolvidas em universidades e institutos da Europa. No INRIA³ Pierre-Louis Curien criou a *Categorical Abstract Machine* (CAM), uma máquina virtual e modelo de computação adequado para execução de linguagens funcionais.

Gérard Huet e outros pesquisadores criaram uma versão de ML usando a máquina CAM, e a chamaram de CAML, em meados da década de 1980.

Como o código executado por CAML era lento e usava muita memória, Xavier Leroy e Damien Doligez criaram *Caml Light* no início da década de 1990, muito mais rápida e compacta.

A linguagem foi estendida na década de 1990 para incluir suporte a orientação a objetos, resultando em *Objective Caml*.

Em 2011 o nome foi mudado para *OCaml*.

1.1.2 Características de OCaml

- Estaticamente tipada
- Inferência de tipos
- Orientada a expressões
- Compilação nativa ou para *bytecode*
- Uso interativo ou em *batch*
- Tipos algébricos e casamento de padrão
- Módulos
- Objetos e classes
- Avaliação estrita

1.1.3 Por que OCaml?

- Aprendizagem de um novo paradigma de programação.
 - Desempenho e expressividade
- Programas escritos em OCaml tendem a ter desempenho mais próximo de linguagens como C++, mas com código mais sucinto, geralmente mais próximo de linguagens como Python e Ruby.

1.1.4 Usos e Aplicações

OCaml não é somente experimental. Existem aplicações de complexidade e tamanho significativos escritos em OCaml. Estas aplicações tendem a ser de domínios relativamente complexos ou que usam pesquisas de ponta. Algumas delas funcionam em ambientes que requer alta confiabilidade.

- **Jane Street Capital** empresa financeira especializada na compra e venda de produtos financeiros em alta frequência (*Hight-Speed Trading*).
- **Coq** sistema assistente para provas que pode ser usado
 - para auxílio na prova de teoremas matemáticos (e verificar as provas criadas),
 - para verificação de outros sistemas de software (provar que um programa segue sua especificação), e
 - como sistema de programação, para criar os chamados *produtos certificados*.
- **Mirage OS** sistema operacional criado especificamente para a computação em nuvens.
- **Microsoft** criou o projeto SLAM que resultou na ferramenta SDV (*Static Driver Verifier*) para verificação de drivers.
- **Facebook** compilador da linguagem Hack e algumas ferramentas internas.

³O INRIA (*Institute national de recherche en informatique et en automatique*) é uma organização pública francesa de carácter científico e tecnológico.

1.2 O Sistema OCaml

Para o desenvolvimento de aplicações em OCaml é necessário o compilador de OCaml e um conjunto de bibliotecas. A distribuição de OCaml oferece dois compiladores:

- **ocamlc**: um compilador para bytecode
- **ocamlopt**: um compilador nativo

Existe também um ambiente interativo (**ocaml**) onde o programador pode carregar módulos e avaliar expressões, facilitando o desenvolvimento incremental.

Para preparar os programas é necessário um editor de texto e um terminal, ou um ambiente de desenvolvimento integrado (IDE).

O desenvolvimento baseado na linha de comando usa um terminal e um editor de textos. O programador digita os arquivos fontes em um editor de texto e compila a aplicação em um terminal executando um dos compiladores.

Emacs (juntamente com o plugin Tuareg) é um bom editor de texto com facilidades para desenvolvimento em OCaml. Porém o grau de dificuldade de sua aprendizagem é considerado elevado. Emacs está disponível para a maioria das plataformas, incluindo Linux e Windows.

Notepad++ é uma opção mais simples disponível no Windows. No Linux temos outras opções como o gedit (parte do ambiente GNOME) e o kate (parte do ambiente KDE).

Eclipse é um ambiente de desenvolvimento integrado muito usado com várias linguagens de programação como Java e C++. O plugin OcaIDE oferece suporte para desenvolvimento em OCaml no Eclipse.

1.2.1 Ferramentas da distribuição OCaml

- **ocamlc**: compilador para *bytecode*.
- **ocamlopt**: compilador para código nativo.
- **ocaml**: ambiente interativo.
- **ocamldoc**: gerador de documentação baseado em comentários.
- **ocamlbuild**: ferramenta para compilação de projetos OCaml.
- **ocamldep**: analisador de dependências entre módulos, geralmente usado para determinar a ordem de compilação dos arquivos.
- **ocamllex** e **ocamlyacc**: geradores de analisadores léxicos e sintáticos, usados principalmente na criação de compiladores e interpretadores para linguagens de programação.
- **ocamlmktop**: possibilita a criação de ambientes interativos customizados para OCaml.

1.2.2 O ambiente interativo

O REPL (*Read Eval Print Loop*) lê declarações e expressões digitadas pelo usuário, compila-as para **bytecode**, e executa-as. Ele é útil para testar e explorar novas idéias.

O ambiente interativo padrão (**ocaml**) tem algumas deficiências, como por exemplo a edição da entrada é bastante básica e não há suporte para um histórico de linhas digitadas. Sugere-se adicionar estas capacidades usando utilitários como **ledit** ou **rlwrap**. Pode-se também usar o ambiente interativo alternativo **utop**, que é bastante melhorado.

1.2.3 Pacotes e bibliotecas

- **findlib**: ferramenta para instalação, gerenciamento e uso de bibliotecas usando a ferramenta **ocamlfind**.
- **OPAM** (OCaml Package Manager): sistema de gerenciamento de pacotes avançado com resolução de dependência. Ainda não disponível para Windows.

1.2.4 Editores

Vários editores suportam a linguagem OCaml, como por exemplo:

- Emacs
- VIM
- Sublime Text
- Textmate
- Atom

1.3 Instalação das ferramentas de desenvolvimento em OCaml

O compilador de OCaml e suas bibliotecas podem ser instalados de várias maneiras. As mais usadas são:

- usando OPAM, um gerenciador de pacotes específicos para OCaml (ainda não disponível para Windows),
- usando um gerenciador de pacotes suportado para sua plataforma, ou
- a partir do código fonte.

A página [Install OCaml](http://ocaml.org/doc/install.html)⁴ apresenta maiores detalhes.

Alguns ambientes de desenvolvimento integrado (IDEs) como o OcaIDE discutido na seção 1.3.3 não funcionam bem com as últimas versões do OCaml. Por exemplo a versão 1.2.21 do OcaIDE (a versão mais recente no momento em que este documento foi escrito), lançada em junho de 2014, exige que a versão do OCaml seja no máximo 4.01.0.

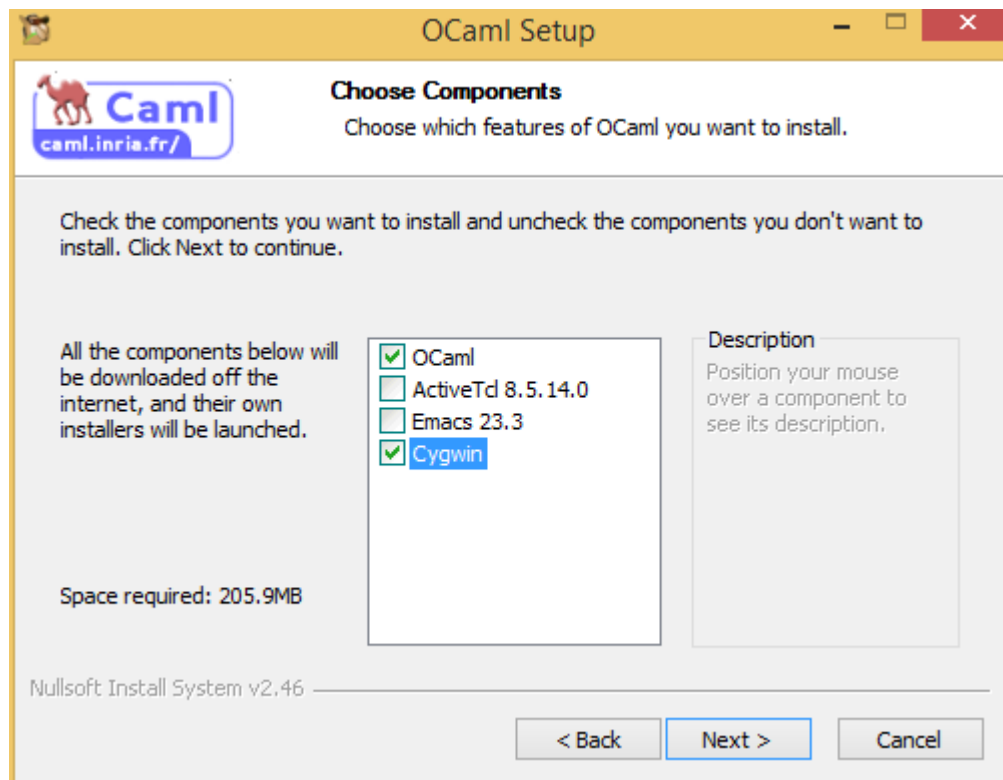
1.3.1 Instalação do OCaml no Windows: Instalador oficial

Há várias maneiras de se instalar OCaml no Windows. Vamos usar o instalador oficial, disponível na página <http://protz.github.com/ocaml-installer/>.

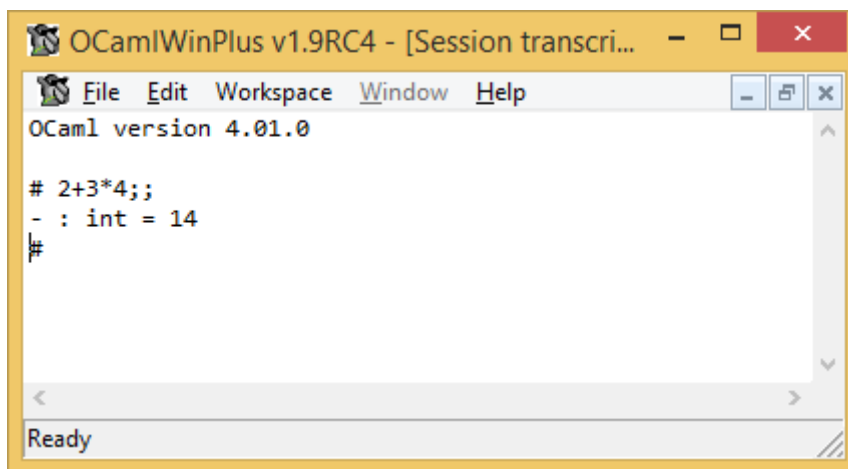
1. Faça o download da versão mais recente do programa de instalação disponível na página <http://protz.github.io/ocaml-installer/>. No momento em que estas instruções foram escritas a versão mais recente era OCaml-4.02.3, acessível pelo link <http://gallium.inria.fr/~protzenk/caml-installer/ocaml-4.02.3-i686-mingw64-installer4.exe>.
2. Execute o programa de instalação, que poderá instalar:
 - OCaml, findlib, camlp4 e flexdll.
 - Cygwin (opcional), uma coleção de ferramentas que permitem que o Windows possa de certa forma agir como um sistema Unix. É necessário para compilação nativa de programas em OCaml. O programa de instalação do Cygwin será executado automaticamente com os pacotes necessários ao OCaml previamente selecionados.

Certifique-se de habilitar a instalação do Cygwin.

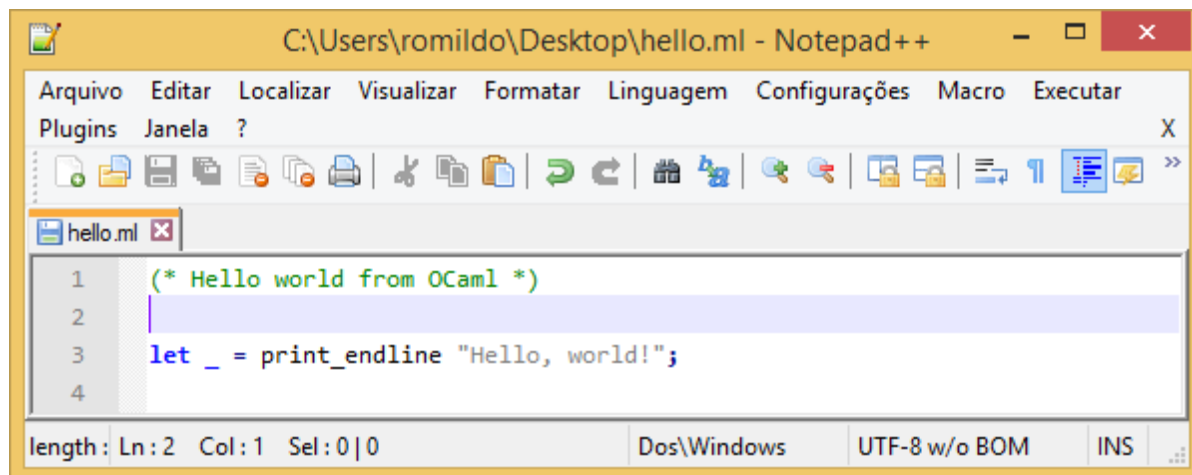
⁴<http://ocaml.org/docs/install.html>



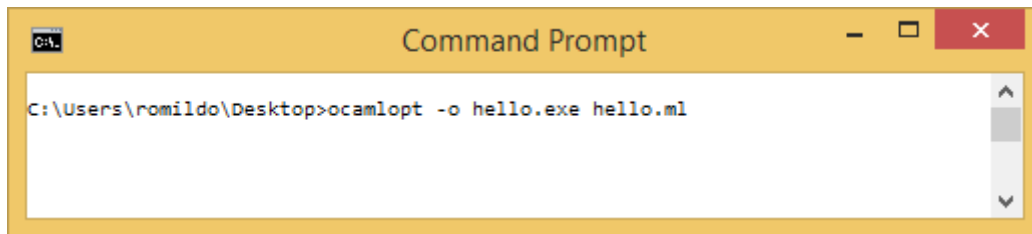
O ambiente interativo poderá ser executado usando o item OCamlWin no menu de programas. Será aberta uma janela onde você pode avaliar expressões e declarações.



Para desenvolver um programa, edite o texto do programa em um editor de texto, como o Notepad++:

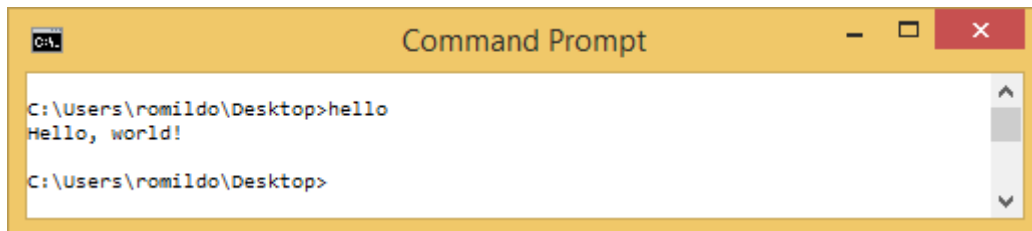


Em seguida compile o programa no terminal:



```
C:\Users\romildo\Desktop>ocamlpt -o hello.exe hello.ml
```

E execute o programa no terminal:



```
C:\Users\romildo\Desktop>hello
Hello, world!

C:\Users\romildo\Desktop>
```

1.3.2 Instalação do OCaml no Ubuntu

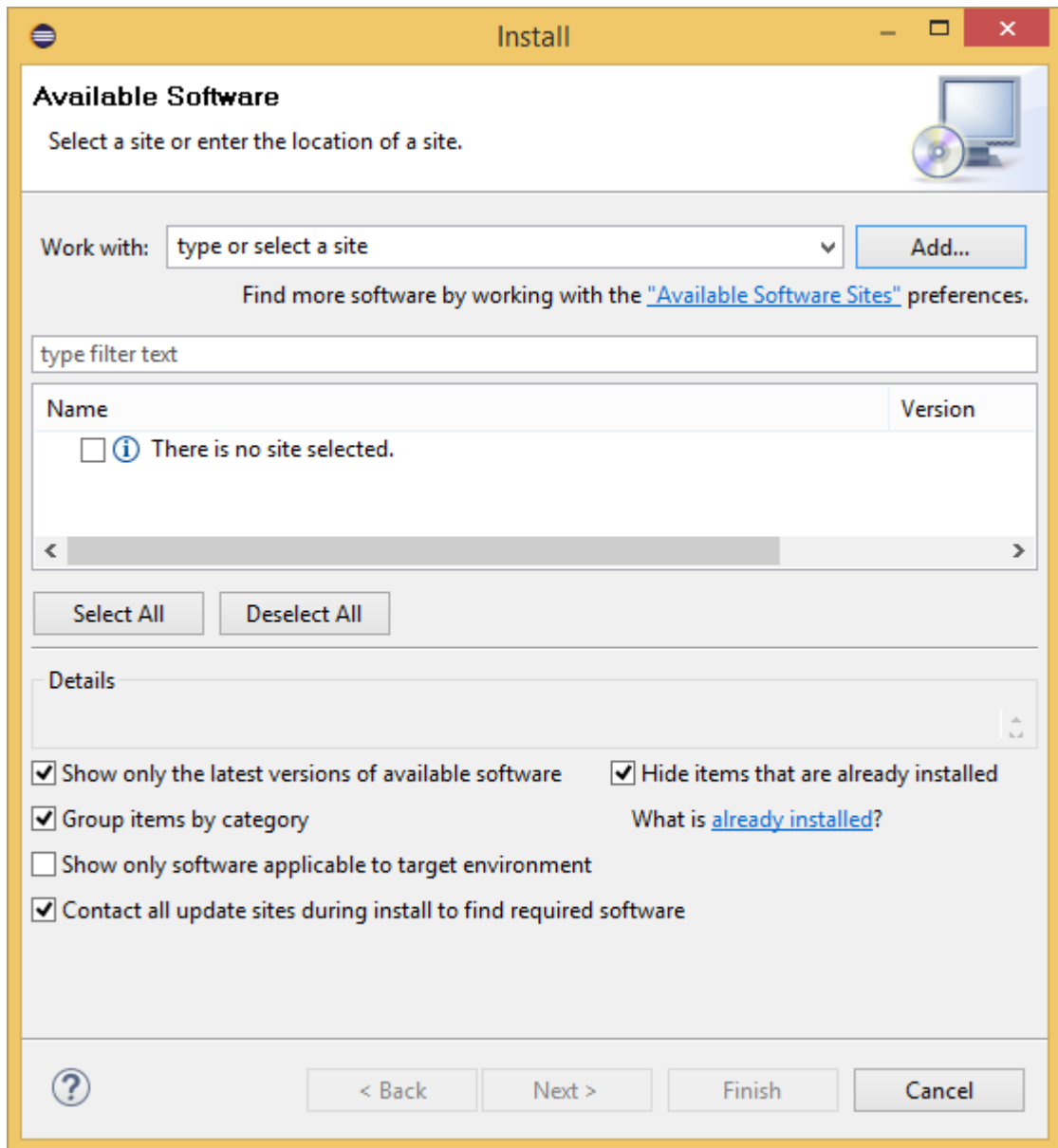
OCaml é muito fácil de ser instalado no Ubuntu. Use o gerenciador de pacotes para instalar os seguintes pacotes:

- `ocaml`, para desenvolvimento de aplicações
- `ocaml-native-compilers`, para compilação de programas para código nativo
- `ocaml-doc`, para ter acesso ao manual de referência
- `tuareg-mode`, um plugin para o editor Emacs
- `ocaml-findlib`, para instalar e usar bibliotecas e suas dependências facilmente
- `menhir`, um gerador de analisadores sintáticos
- `rlwrap`, para facilitar a edição da linha de entrada do ambiente interativo,

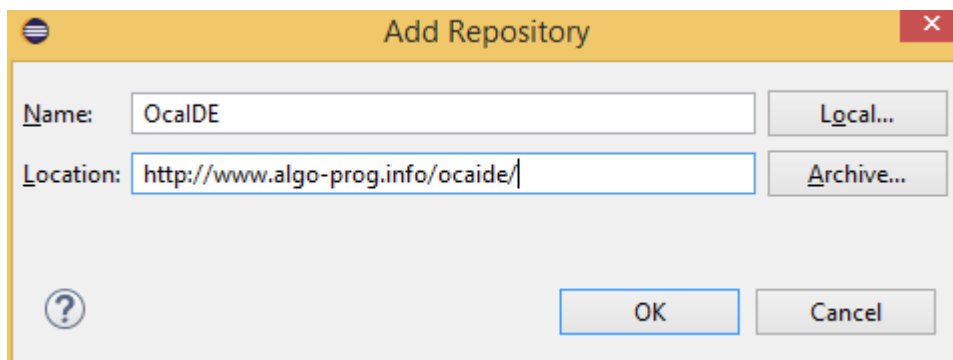
1.3.3 Instalação do ambiente interativo no Eclipse

Para instalar OcaIDE, um plugin para Eclipse:

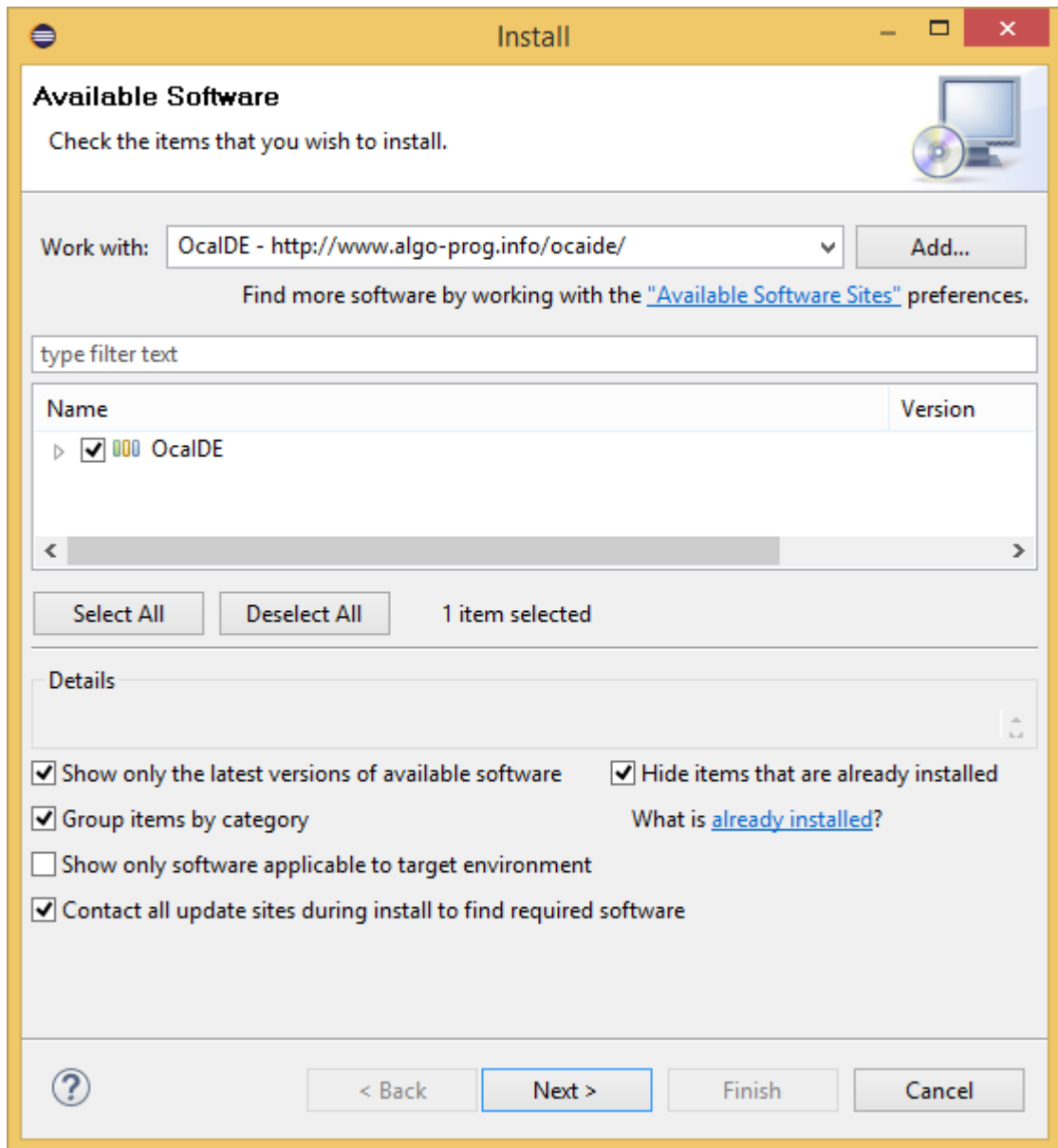
1. Instale a máquina virtual do Java versão 1.7 ou superior (<http://java.com/en/download/index.jsp>).
2. Instale uma versão recente do Eclipse (<http://www.eclipse.org/downloads/>).
3. Inicie o Eclipse.
4. No Eclipse acesse o menu `Help→Install New Software...`



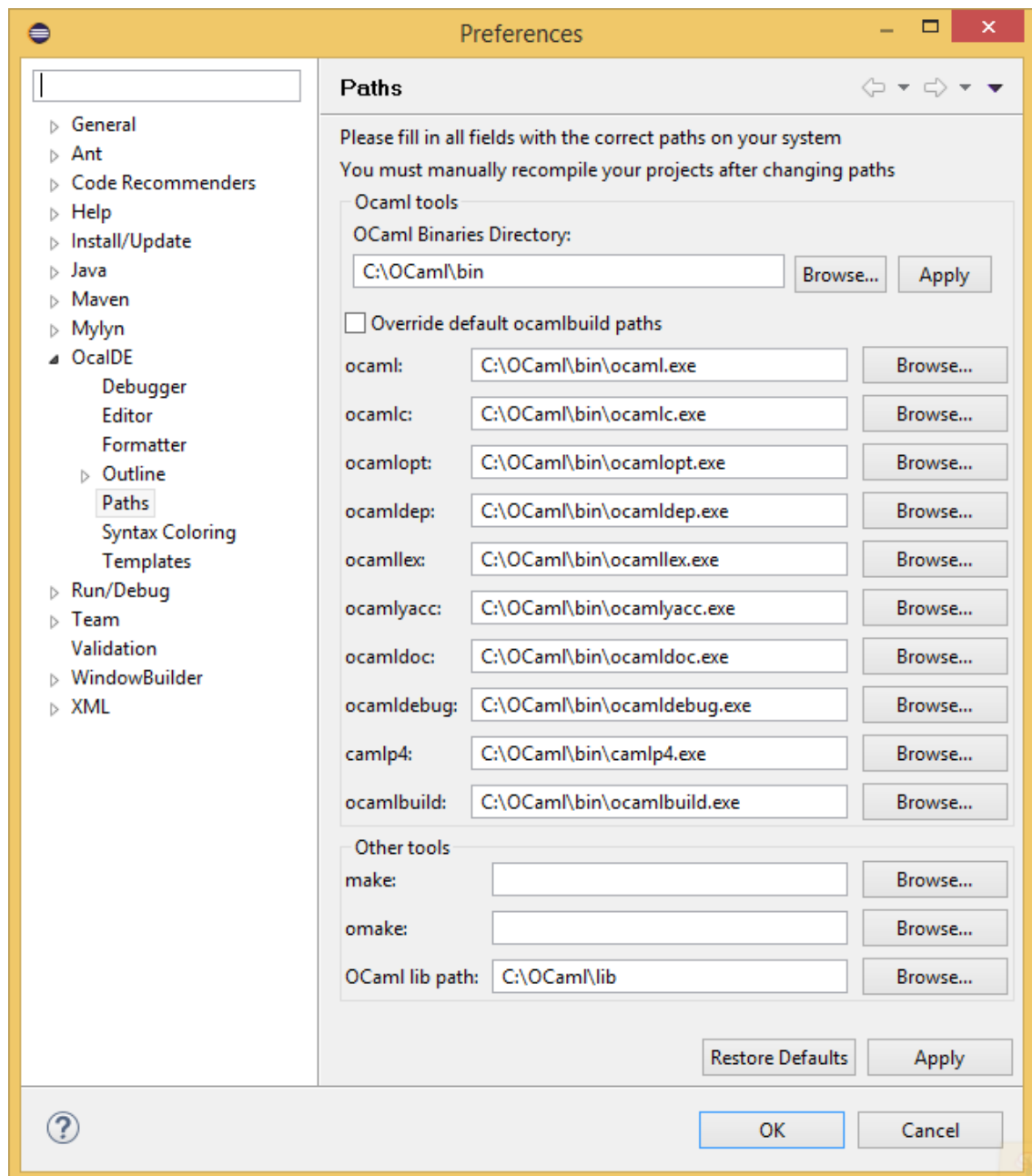
5. Clique Add para configurar um novo repositório.
6. No campo Name digite OcaIDE e no campo Location digite <http://www.algo-prog.info/ocaide/>.



7. Clique em Ok.
8. Selecione a categoria OcaIDE na lista de plugins.



9. Clique Next.
10. Clique Next.
11. Aceite os termos do acordo de licença.
12. Clique Finish. Será feito o download do plugin.
13. Aceite a instalação (o plugin não tem assinatura digital).
14. Reinicie o Eclipse quando solicitado.
15. Após o reinício do Eclipse, o ambiente já está pronto para uso.
16. Para acessar a ajuda online clique no menu Help→Help Contents e escolha *OCaml Development User Guide* na lista de tópicos disponíveis.
17. Verifique os caminhos onde o OCaml pode ser encontrado. Para tanto acesse o menu Window→Preferences e selecione a aba OcaIDE→Paths. Se necessário preencha o campo Ocaml Binaires Directory para indicar a localização correta dos programas executáveis do OCaml e em seguida clique em Apply. Pode ser necessário indicar também o caminho das bibliotecas do OCaml no campo Ocaml lib path. Conclua clicando em OK.



Alternativamente o plug-in pode ser instalado manualmente:

1. Faça o download do arquivo jar a partir do link <http://www.algo-prog.info/ocaide/plugins>. No momento a versão mais recente do plug-in é http://www.algo-prog.info/ocaide/plugins/Ocaml_1.2.21.201406141244.jar.
2. Coloque o plug-in no diretório dropins da instalação do Eclipse.
3. Reinicie o Eclipse.

1.4 Usando o ambiente interativo

No ambiente interativo do OCaml podemos digitar expressões e declarações da linguagem OCaml. Elas são compiladas e avaliadas, e o tipo e o valor obtido são exibidos.

O ambiente interativo padrão pode ser iniciado a partir de um terminal usando o comando **ocaml**.

Em um sistema Unix:

```
$ ocaml
      OCaml version 4.01.0

#
```

Para adicionar facilidades de edição da linha de entrada e histórico, use os utilitários `ledit` ou `rlwrap`.
Em um sistema Unix:

```
$ rlwrap ocaml
      OCaml version 4.01.0

#
```

Expressões da linguagem OCaml podem ser digitadas no prompt. A entrada deve ser encerrada com dois sinais de ponto-e-vírgula: `;;`.

Por exemplo:

```
# 2 + 3*4;;
- : int = 14
```

```
# (2 + 3)*4;;
- : int = 20
```

```
# print_string "Hello, world!\n";;
Hello, world!
- : unit = ()
```

Observe que o ambiente interativo responde informando o tipo e o valor calculados.
Algumas diretivas interessantes que podem ser usadas no ambiente interativo:

#quit

Para encerrar o ambiente interativo use a diretiva `#quit` ou digite `Control+D` (no Linux) ou `Control+Z` (no Windows).

#cd

Para mudar o diretório de trabalho atual use a diretiva `#cd` "*nome-do-diretório*".

#use

Para ler, compilar e executar expressões e declarações de um dado arquivo use a diretiva `#use` "*nome-do-arquivo*".

2 VALORES, TIPOS E FUNÇÕES

Resumo

As linguagens funcionais modernas apresentam um sistema de tipos robusto que permite ao compilador verificar se os operandos usados nas operações estão corretos. Com a inferência de tipos isto pode ser feito sem o programador ter que necessariamente anotar os tipos das variáveis e funções usadas nos programas.

Nesta aula vamos conhecer alguns tipos básicos de OCaml.

Sumário

2.1	Considerações léxicas	2-1
2.1.1	Comentários	2-1
2.1.2	Palavras reservadas	2-1
2.1.3	Identificadores	2-2
2.2	Alguns tipos básicos	2-2
2.2.1	Números inteiros: tipo <code>int</code>	2-2
2.2.2	Números reais: tipo <code>float</code>	2-3
2.2.3	Caracteres: tipo <code>char</code>	2-4
2.2.4	Cadeias de caracteres: tipo <code>string</code>	2-5
2.2.5	Valores lógicos: tipo <code>bool</code>	2-5
2.2.6	Tupla vazia: tipo <code>unit</code>	2-5
2.3	Constantes	2-6
2.4	Aplicação de função	2-6
2.5	Definindo variáveis e funções globais	2-8
2.6	Tipos função	2-9
2.7	Checagem de tipos	2-9
2.8	Inferência de tipos	2-10
2.9	Exercícios	2-11

2.1 Considerações léxicas

2.1.1 Comentários

Comentários são usados para fazer anotações no programa que podem ajudar a entender o funcionamento do mesmo. Os comentários são ignorados pelo compilador.

Comentários são delimitados por `(*` e `*)`. Comentários podem ser aninhados. Comentários não podem ocorrer em literais caracter e literais string.

2.1.2 Palavras reservadas

As seguintes palavras são reservadas e não podem ser usadas como identificadores:

<code>and</code>	<code>as</code>	<code>assert</code>	<code>asr</code>	<code>begin</code>	<code>class</code>
<code>constraint</code>	<code>do</code>	<code>done</code>	<code>downto</code>	<code>else</code>	<code>end</code>
<code>exception</code>	<code>external</code>	<code>false</code>	<code>for</code>	<code>fun</code>	<code>function</code>
<code>functor</code>	<code>if</code>	<code>in</code>	<code>include</code>	<code>inherit</code>	<code>initializer</code>
<code>land</code>	<code>lazy</code>	<code>let</code>	<code>lor</code>	<code>lsl</code>	<code>lsr</code>
<code>lxor</code>	<code>match</code>	<code>method</code>	<code>mod</code>	<code>module</code>	<code>mutable</code>
<code>new</code>	<code>object</code>	<code>of</code>	<code>open</code>	<code>or</code>	<code>private</code>
<code>rec</code>	<code>sig</code>	<code>struct</code>	<code>then</code>	<code>to</code>	<code>true</code>
<code>try</code>	<code>type</code>	<code>val</code>	<code>virtual</code>	<code>when</code>	<code>while</code>
<code>with</code>					

As seguintes sequências de caracteres também são reservadas:

!=	#	&	&&	'	(	)	*	+	,	-
-.>	->	.	..	:	::	::=	::>	;	;;	<
<-	=	>	>]	>}	?	[	[<	[>	[	]
_	'	{	{<		]		}	~		

2.1.3 Identificadores

Identificadores são usados para nomear elementos em um programa. São formados por uma sequência de letras, dígitos decimais, sublinhado (`_`) e apóstrofo (`'`) começando com uma letra ou sublinhado, e não pode ser uma palavra reservada.

São exemplos de identificadores:

```
peso
altura_aluno
Estrutura
nome3
x'
__conceito
```

Não são identificadores:

```
7anos
nome-do-professor
let
```

2.2 Alguns tipos básicos

Um **tipo** é uma *coleção de valores* relacionados. Tipos servem para *classificar os valores* de acordo com as suas características.

Em OCaml **nomes de tipo** são sequências de letras, dígitos decimais, sublinhados e apóstrofo, *começando com uma letra minúscula ou sublinhado*.

Por exemplo, o tipo **bool** contém os dois valores lógicos **false** e **true**, comumente usados nas operações lógicas.

Alguns tipos básicos de OCaml são apresentados a seguir.

2.2.1 Números inteiros: tipo **int**

Caracterização do tipo **int**

- inteiros de precisão fixa
- limitado, sendo a faixa de valores determinada pelo tamanho da palavra da plataforma:
 - 32 bits: de -2^{30} até $2^{30} - 1$
 - 64 bits: de -2^{62} até $2^{62} - 1$

Exemplos de valores do tipo **int**

```
7703876
7_703_876
-18605
0b100101
0o2012
0x2FA3
```

Algumas operações relacionadas com o tipo **int**

operação	descrição
-	negação unária de inteiros
~-	negação unária de inteiros
+	adição de inteiros
-	subtração de inteiros
*	multiplicação de inteiros
/	quociente da divisão de inteiros
mod	resto da divisão de inteiros; é um operador binário
succ	sucessor
pred	antecessor
abs	valor absoluto
min_int	o maior inteiro representável
max_int	o menor inteiro representável

2.2.2 Números reais: tipo **float**

Caracterização do tipo **float**

- aproximação de números reais em ponto flutuante
- precisão dupla de acordo com o padrão IEEE 754, com 53 bits de mantissa e expoente de -1022 até 1023

Exemplos de valores do tipo **float**

```
786.43
-78643.0
-7.893_001_999
418.
.2015
987.3201E-60
34e8
```

Algumas operações relacionadas com o tipo **float**

operação	descrição
-.	negação unária de reais
~.	negação unária de reais
+	adição de reais
-	subtração de reais
*	multiplicação de reais
/	divisão de reais
**	potenciação de reais
abs_float	valor absoluto
sqrt	raiz quadrada
exp	exponencial
log	logaritmo natural
log10	logaritmo decimal
sin	seno
cos	cosseno
tan	tangente
asin	arco seno
acos	arco cosseno
atan	arco tangente
float_of_int	conversão de inteiro para real
float	conversão de inteiro para real
int_of_float	conversão de real para inteiro
truncate	conversão de real para inteiro
ceil	arredonda para cima
floor	arredonda para baixo
infinity	infinito positivo
neg_infinity	infinito negativo
nan	<i>not a number</i> : valor especial denotando resultado de operação indefinida
min_float	o menor valor positivo não nulo e não denormalizado do tipo float
max_float	o maior valor positivo finito do tipo float

Observação

Diferentemente da maioria das linguagens de programação, os literais, as funções e os operadores para os tipos numéricos **não são sobrecarregados**. Assim o literal 2342 é somente do tipo **int** e nunca do tipo **float**. O operador + pode ser usado apenas com inteiros, e nunca com números em ponto flutuante. Para números em ponto flutuante existe o operador +..

2.2.3 Caracteres: tipo **char**

Caracterização do tipo **char**

- enumeração cujos valores representam caracteres ISO 8859-1
- representados internamente como inteiros de 8 bits entre 0 e 255

Exemplos de valores do tipo **char**

'B'
'!'
'\n'
'\t'
'\066'
'\x42'

Algumas operações relacionadas com o tipo **char**

operação	descrição
char_of_int	obtém o caracter com o código dado
int_of_char	obtém o código do caracter dado

2.2.4 Cadeias de caracteres: tipo `string`

Caracterização do tipo `string`

- sequências finitas de caracteres

Exemplos de valores do tipo `string`

```
"Brasil"  
""  
"bom\ndia"
```

Algumas operações relacionadas com o tipo `string`

operação	descrição
<code>^</code>	concatenação
<code>. [indice]</code>	seleção
<code>string_of_bool</code>	converte de bool para string
<code>bool_of_string</code>	converte de string para bool
<code>string_of_int</code>	converte de int para string
<code>int_of_string</code>	converte de string para int
<code>string_of_float</code>	converte de float para string
<code>float_of_string</code>	converte de string para float

2.2.5 Valores lógicos: tipo `bool`

Caracterização do tipo `bool`

- valores lógicos

Exemplos de valores do tipo `bool`

```
false  
true
```

Algumas operações relacionadas com o tipo `bool`

operação	descrição
<code>not</code>	negação lógica
<code>&&</code>	e lógico
<code> </code>	ou lógico
<code>=</code>	igualdade estrutural
<code><></code>	desigualdade estrutural
<code><</code>	menor que
<code><=</code>	menor ou igual a
<code>></code>	maior que
<code>>=</code>	maior ou igual a

2.2.6 Tupla vazia: tipo `unit`

Caracterização do tipo `unit`

- possui um único valor: a tupla vazia
- útil para expressões que são avaliadas apenas pelo efeito que produzem, não tendo um valor interessante para ser o resultado

Exemplos de valores do tipo `unit`

```
()
```

Algumas operações relacionadas com o tipo `unit`

operação	descrição
<code>ignore</code>	descarta o valor do argumento e retorna <code>()</code> , preservando o efeito

2.3 Constantes

As formas mais simples de expressões são as constantes. Elas representam valores em sua forma mais simples, ou seja, já estão reduzidos à sua forma canônica.

São constantes:

- os literais inteiros, fracionários, caracteres e strings,
- os construtores constantes dos tipos algébricos,
- as constantes especiais **false**, **true**, **()** (tupla vazia), **[]** (lista vazia), e **[] []** (array vazio), e
- **begin end**, que é equivalente a **()**.

Algumas destas constantes somente serão apresentadas posteriormente.

2.4 Aplicação de função

Aplicação de função é uma das formas de expressão mais comuns na programação funcional, uma vez que os programas são organizados em funções.

Sintaticamente uma aplicação de função em **notação prefixa** consiste em escrever a função seguida dos argumentos, se necessário *separados por caracteres brancos* (espaços, tabuladores, mudança de linha, etc.).

Exemplos:

```
# sqrt 25.0;;
- : float = 5.
# cos 0.0;;
- : float = 1.
# exp 1.0;;
- : float = 2.71828182845904509
# compare "ana" "paula";;
- : int = -1
# max 18 501;;
- : int = 501
```

Observe que, diferentemente de várias outras linguagens de programação, **os argumentos não são escritos entre parênteses e nem separados por vírgula**.

Parênteses podem ser usados para **agrupar subexpressões**. Por exemplo:

```
# sqrt (max 25.0 0.36);;
- : float = 5.
# min (sqrt 9.0) 0.81;;
- : float = 0.81
```

Aplicações de função também podem ser escritas em **notação infixa**, onde a função é escrita entre os seus argumentos. Neste caso dizemos que as funções são **operadores infixos**. Os nomes dos operadores infixos são sequências de caracteres simbólicos

! \$ % & * + - . / : < = > ? @ ^ | ~

começando com um dos caracteres

= < > @ ^ | & + - * / \$ %

tais como **<=>** e **!!**. Exemplos:

```
# 2 + 3;;
- : int = 5
# 10 / 4;;
- : int = 2
# (12 - 7) * 6;;
- : int = 30
# 6 <= 17;;
```

```

- : bool = true
# 'A' = 'B';;
- : bool = false
# 'A' <> 'B';;
- : bool = true
# true || false;;
- : bool = true
# true && false;;
- : bool = false

```

Assim como na Matemática e em outras linguagens de programação, os operadores possuem um **nível de precedência** (ou prioridade) e uma **associatividade**. Parênteses podem ser usados para agrupar subexpressões dentro de expressões maiores *quebrando* a precedência ou associatividade dos operadores.

A tabela 2.1 lista a precedência relativa e a associatividade dos operadores e outras construções de OCaml. O nível de precedência e associatividade de um operador infixo é determinado pelo seu primeiro caracter. Na tabela 2.1 *... significa qualquer operador infixo começando com *.

Exemplos:

```

# 2 + 3 * 4;;          (* * tem maior precedência que + *)
- : int = 14
# 5.0 ** 2.0 -. 10.0;;  (* ** tem maior precedência que -. *)
- : float = 15.
# 2.0 ** 3.0 ** 2.0;;   (* ** associa-se à direita *)
- : float = 512.

```

Aplicações de função em notação prefixa tem precedência maior do que todos os operadores. Exemplos:

```

# abs 10 - 20;;        (* abs tem precedência maior que - *)
- : int = -10
# abs (10 - 20);;
- : int = 10
# succ 9 + max 5 4 * 3;; (* succ e max tem precedência maior que + e * *)
- : int = 25

```

Um operador pode ser associativo à esquerda, associativo à direita, ou não-associativo. Quando dois operadores com a mesma precedência disputam um operando,

- se eles forem associativos à esquerda, o operador da esquerda é escolhido,
- se eles forem associativos à direita, o operador da direita é escolhido,
- se eles forem não associativos, a expressão é mal formada e contém um erro de sintaxe,

Exemplos:

```

# 15 - 4 - 6;;        (* - associa-se à esquerda *)
- : int = 5
# 15 - (4 - 6);;
- : int = 17
# 10 - 2 + 5;;         (* + e - tem a mesma precedência e associam-se à esquerda *)
- : int = 13
# 10 - (2 + 5);;
- : int = 3
# 2.0 ** 3.0 ** 4.0;;  (* ** associa-se à direita *)
- : float = 2.41785163922925835e+24

```

Qualquer operador infixo pode ser usado em notação prefixa, bastando escrevê-lo entre parênteses. Exemplos:

```

# (+) 4 5;;
- : int = 9
# (/.) 18.2 2.0;;
- : float = 9.1
# (>=) 10 20;;
- : bool = false
# sqrt ((+.) 4.1 4.9);;
- : float = 3.

```

operador ou construção	associativade
símbolo prefixo	-
. .(C .[.{'	-
#	-
aplicação de função aplicação de construtor aplicação de tag assert lazy	esquerda
- (prefixo) - . (prefixo)	-
**... lsl lsr asr	direita
*... /... %... mod land lor lxor	esquerda
+... -...	esquerda
::	direita
@... ^...	direita
=... <... >... ... &... \$... !=	esquerda
& &&	direita
or 	direita
,	-
<- :=	direita
if	-
;	direita
let match fun function try	-

Tabela 2.1: Precedências e associatividades dos operadores e construções de OCaml.

2.5 Definindo variáveis e funções globais

Além de poder usar as funções das bibliotecas, o programador também pode *definir* e *usar* suas próprias variáveis e funções. *Variáveis* e *funções* são definidas usando a palavra reservada **let**.

Nomes de variáveis e funções são identificadores que começam com letra minúscula ou sublinhado.

Uma declaração global de variável é da forma:

let nome = expressão

sendo *nome* o nome da variável, e *expressão* uma expressão qualquer. Quando elaborada, esta declaração avalia a expressão associando o seu valor ao nome indicado.

Exemplos:

```
# let idade = 21;;  
val idade : int = 21  
# let x = 2*idade - 1;;  
val x : int = 41  
# idade <= 30;;  
- : bool = true
```

Uma declaração global de função é da forma:

let nome par₁ ... par_n = expressão

sendo *nome* o nome da função, *par₁, ..., par_n* ($n > 0$) os nomes dos parâmetros formais da função, e *expressão* o corpo da função. Quando elaborada, esta declaração cria uma função associando-a ao nome indicado.

Exemplos:

```
# let dobro x = x + x;;  
val dobro : int -> int = <fun>  
# dobro 5;;  
- : int = 10  
# let quadruplo x = dobro (dobro x);;  
val quadruplo : int -> int = <fun>  
# quadruplo 5;;  
- : int = 20
```

2.6 Tipos função

Nas linguagens funcionais uma **função** é um valor de primeira classe e, assim como os demais valores, tem um tipo. Este tipo é caracterizado pelos tipos dos argumentos e pelo tipo do resultado da função.

Em OCaml um **tipo função** é escrito usando o operador de tipo $\rightarrow$:

$t_1 \rightarrow \dots \rightarrow t_n$

onde

- $t_1, \dots, t_{n-1}$ são os tipos dos argumentos
- t_n é o tipo do resultado

Exemplos:

bool $\rightarrow$ bool

tipo das funções com um argumento do tipo **bool**, e resultado do tipo **bool**, como por exemplo a função `not`

bool $\rightarrow$ bool $\rightarrow$ bool

tipo das funções com dois argumentos do tipo **bool**, e resultado do tipo **bool**, como por exemplo os operadores `&&` e `||`

int $\rightarrow$ float $\rightarrow$ float $\rightarrow$ bool

tipo das funções com três argumentos, sendo o primeiro do tipo **int**, o segundo e o terceiro do tipo **float**, e o resultado do tipo **bool**

2.7 Checagem de tipos

Toda expressão sintaticamente correta tem o seu tipo calculado em *tempo de compilação*. Se não for possível determinar o tipo de uma expressão ocorre um **erro de tipo**.

A aplicação de uma função a um ou mais argumentos de *tipo inadequado* constitui um **erro de tipo**. Por exemplo:

```
# not 'A';;  
Error: This expression has type char but an expression was expected of type  
      bool
```

Explicação:

A função `not` requer um valor booleano, porém foi aplicada ao argumento `'A'`, que é um caracter.

OCaml é uma linguagem **fortemente tipada**, com um **sistema de tipos** muito avançado. Todos os possíveis erros de tipo são encontrados em tempo de compilação (**tipagem estática**). Isto torna os programas *mais seguros e mais rápidos*, eliminando a necessidade de verificações de tipo em tempo de execução.

2.8 Inferência de tipos

Toda expressão bem formada tem um **tipo mais geral**, que pode ser calculado *automaticamente em tempo de compilação* usando um processo chamado **inferência de tipos**.

A capacidade de inferir tipos automaticamente *facilita a programação*, deixando o programador livre para omitir anotações de tipo ao mesmo tempo que permite a verificação de tipos.

A inferência de tipo é feita usando as **regras de tipagem** de cada forma de expressão.

Literais inteiros Os literais inteiros são do tipo **int**.

Literais fracionários Os literais fracionários são do tipo **float**.

Literais caracteres Os literais caracteres são do tipo **char**.

Literais strings Os literais strings são do tipo **string**.

Construtores constantes Os construtores constantes de um tipo são do tipo associado. Assim:

- os construtores constantes booleanos **true** e **false** são do tipo **bool**, e

Aplicação de função

$$\frac{\begin{array}{c} x_1 :: a_1 \\ \vdots \\ x_n :: a_n \\ f :: a_1 \rightarrow \dots \rightarrow a_n \rightarrow b \end{array}}{f \ x_1 \ \dots \ x_n :: b}$$

Em uma aplicação de função:

- o tipo dos argumentos deve ser compatível com os domínios da função
- o tipo do resultado deve ser compatível com o contra-domínio da função

2.9 Exercícios

Tarefa 2.1: Movimento Retilíneo Uniformemente Variado

A posição s de um corpo em movimento retilíneo uniformemente variado, em função do tempo t , é dado pela equação

$$s = s_0 + v_0 t + \frac{1}{2} a t^2$$

onde s_0 é a posição inicial do corpo, v_0 é a sua velocidade inicial, e a é a sua aceleração.



Utilize o ambiente interativo do OCaml para calcular a posição de uma bola em queda livre no instante $t = 8$ s, considerando que a posição inicial é $s_0 = 100$ m, a velocidade inicial é $v_0 = 15$ m/s e a aceleração da gravidade é $a = -9.81$ m/s².

Dica:

Use a declaração `let` para criar variáveis correspondentes aos dados e em seguida avalie a expressão correspondente à função horária do movimento usando estas variáveis.

Tarefa 2.2: Expressões matemáticas

Utilize o ambiente interativo para avaliar as expressões aritméticas seguintes, considerando que $x = 3$ e $y = 4$.

- $\frac{4}{3} \pi \sin x^2 - 1$
- $\frac{x^2 y^3}{(x - y)^2}$
- $\frac{1}{x^2 - y} - e^{-4x} + \sqrt[3]{\frac{35}{y}} \sqrt{xy}$
- $\frac{24 + 4.5^3}{e^{4.4} - \log_{10} 12560}$
- $\cos \frac{5\pi}{6} \sin^2 \frac{7\pi}{8} + \frac{\tan(\frac{\pi}{6} \ln 8)}{\sqrt{7} + 2}$

Tarefa 2.3

Defina uma função para calcular o quadrado do dobro do seu argumento.

Tarefa 2.4

Defina uma função para calcular o dobro do quadrado do seu argumento.

Tarefa 2.5: Lados de um triângulo

Os lados de qualquer triângulo respeitam a seguinte restrição:

A soma dos comprimentos de quaisquer dois lados de um triângulo é superior ao comprimento do terceiro lado.

Escreva uma função que receba o comprimento de três segmentos de reta e resulte em um valor lógico indicando se satisfazem esta restrição.

Tarefa 2.6: Energia armazenada em uma mola

A força requerida para comprimir uma mola linear é dada pela equação

$$F = kx$$

onde F é a força em N (Newton), x é a compressão da mola em m (metro), e k é a constante da mola em N/m .

A energia potencial armazenada na mola comprimida é dada pela equação

$$E = \frac{1}{2}kx^2$$

onde E é a energia em J (joule).

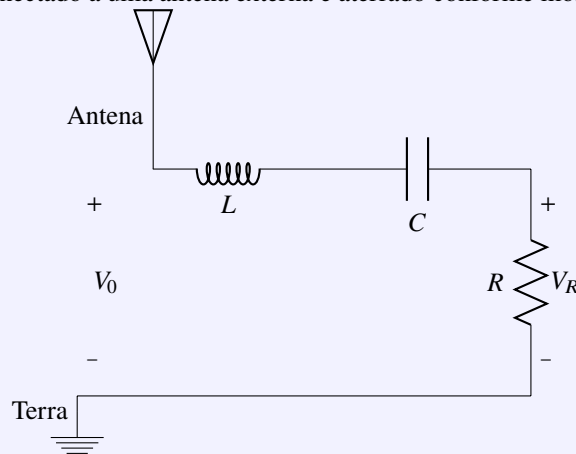
Defina funções para calcular a compressão e a energia potencial armazenada em uma mola, dadas a constante elástica da mola e a força usada para comprimi-la.

Tarefa 2.7: Custo da energia elétrica

Sabe-se que o quilowatt de energia elétrica custa um quinto do salário mínimo. Defina uma função que receba o valor do salário mínimo e a quantidade de quilowatts consumida por uma residência, e resulta no valor a ser pago com desconto de 15%.

Tarefa 2.8: Receptor de rádio

Uma versão simplificada da parte frontal de um receptor de rádio AM é apresentada na figura abaixo. Esse receptor é composto por um circuito que contém um resistor R , um capacitor C e um indutor L conectados em série. O circuito é conectado a uma antena externa e aterrado conforme mostra a figura.



O circuito permite que o rádio selecione uma estação específica dentre as que transmitem na faixa AM. Na frequência de ressonância do circuito, essencialmente todo o sinal V_0 da antena vai até o resistor, que representa o resto do rádio. Em outras palavras, o rádio recebe seu sinal mais forte na frequência de ressonância. A frequência de ressonância do circuito indutor-capacitor é dada pela equação

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

onde L é a indutância em H (henry) e C é a capacitância em F (farad).

Defina uma função que receba a indutância L e a capacitância C , e resulta na frequência de ressonância desse aparelho de rádio

Teste sua função pelo cálculo da frequência do rádio quando $L = 0,25mH$ e $C = 0,10nF$.

Tarefa 2.9: Força gravitacional

A lei da gravitação universal, proposta por Newton a partir das observações de Kepler sobre os movimentos dos corpos celestes, diz que:

Dois corpos quaisquer se atraem com uma força diretamente proporcional ao produto de suas massas e inversamente proporcional ao quadrado da distância entre eles.

Essa lei é formalizada pela seguinte equação:

$$F = G \frac{m_1 m_2}{d^2}$$

onde:

- F é força de atração em Newtons (N),
- G é a constante de gravitação universal ($6.67 \times 10^{-11} \text{ N m}^2/\text{kg}^2$),
- m_1 e m_2 são as massas dos corpos envolvidos, em quilos (kg), e
- d é a distância entre os corpos em metros (m).

1. Defina uma variável para denotar a constante de gravitação universal.
2. Defina uma função que recebe as massas dos dois corpos e a distância entre eles, e resulta na força de atração entre esses dois corpos. Use a variável definida em 1.
3. Teste suas definições no ambiente interativo calculando a força de atração entre a terra e a lua sabendo que a massa da terra é 6×10^{24} kg, a massa da lua é 1×10^{23} kg, e a distância entre eles é 4×10^5 km.

Tarefa 2.10: Salário líquido

Defina uma função que recebe o salário base de um funcionário e resulta no salário líquido a receber, sabendo-se que o funcionário tem gratificação de 10% sobre o salário base e paga imposto de 7% sobre o salário base.

Tarefa 2.11

Defina uma função que verifica se uma equação do segundo grau

$$ax^2 + bx + c = 0$$

possui raízes reais. Para tanto é necessário que o discriminante $\Delta = b^2 - 4ac$ seja não negativo.

Tarefa 2.12: Área do círculo

Defina uma função que recebe a medida do raio r de um círculo e resulta na área A do círculo, dada por:

$$A = \pi \times r^2$$

Tarefa 2.13: Número de degraus

Defina uma função que recebe a altura dos degraus de uma escada e a altura que o usuário deseja alcançar subindo a escada, e resulta na quantidade mínima de degraus que ele deverá subir para atingir seu objetivo, sem se preocupar com a altura do usuário.

Tarefa 2.14

Determine o tipo de cada função definida a seguir.

- 1) **let** dobro $x = x * 2$
- 2) **let** aprovado $\text{nota} = \text{nota} \geq 6$
- 3) **let** myLog $x \text{ b} = \log x / \log b$

3 EXPRESSÃO CONDICIONAL

Resumo

Expressões condicionais permitem a escolha entre duas alternativas na obtenção do valor da expressão, com base em uma condição (expressão lógica).

Nesta aula vamos nos familiarizar com o uso de expressões condicionais. Vamos também aprender a fazer declarações locais a uma equação.

Sumário

- 3.1 Expressão condicional 3-1
- 3.2 Definição de função com expressão condicional 3-2

3.1 Expressão condicional

Uma **expressão condicional** tem a forma

```
if condição then exp1 else exp2
```

onde *condição* é uma *expressão booleana* (chamada **predicado**) e *exp₁* (chamada **consequência**) e *exp₂* (chamada **alternativa**) são expressões de um *mesmo tipo*. O valor da expressão condicional é o valor de *exp₁* se a condição é verdadeira, ou o valor de *exp₂* se a condição é falsa.

Seguem alguns exemplos de expressões condicionais e seus valores.

```
if true then 1 else 2           ~> 1
if false then 1 else 2          ~> 2
if 2>1 then "OK" else "FAIL"    ~> "OK"
if 5 mod 2 = 0 then 3+2 else 3-2 ~> 1
```

A expressão condicional é uma **expressão**, e portanto *sempre tem um valor*. Assim uma expressão condicional pode ser usada *dentro de outra expressão*. Veja os exemplos seguintes.

```
5 * (if true then 10 else 20)    ~> 50
5 * if true then 10 else 20       ~> 50
String.length (if 2<=1 then "OK" else "FAIL") ~> 4
```

Veja na tabela 2.1 a precedência relativa da expressão condicional. Ela mostra que a precedência da expressão condicional só não é menor do que a precedência do operador ; e também das expressões **let**, **match**, **fun**, **function** e **try**.

```
(if 12 mod 2 = 0 then 10 else 20) + 1 ~> 11
if 12 mod 2 = 0 then 10 else 20 + 1    ~> 10
```

Regra de inferência de tipo

$$\frac{\begin{array}{l} test :: \text{bool} \\ e_1 :: a \\ e_2 :: a \end{array}}{if\ test\ then\ e_1\ else\ e_2 :: a}$$

Observe que a *consequência* e a *alternativa* devem ser do mesmo tipo, que também é o tipo do resultado.

Exemplos no ambiente interativo:

```
# if 4>5 then 'S' else 'N';;
- : char = 'N'
# if 8 mod 2 <> 0 then 10 else 20;;
- : int = 20
# if 17 mod 2 = 0 then 12.0 else 5.1;;
- : float = 5.1
```

```
# if 'A' then "ok" else "bad";;
Error: This expression has type char but an expression was expected of type
      bool
# if 17 mod 2 <> 0 then not true else 'H';;
Error: This expression has type char but an expression was expected of type
      bool
```

else opcional

A cláusula **else** de uma expressão condicional é *opcional*. Quando ela é omitida e o valor do teste é falso, o resultado é `()` (ou seja, a tupla vazia), do tipo **unit**. Como de forma geral o compilador não pode determinar se o valor do teste é verdadeiro ou falso, impõe-se a restrição de que o tipo da primeira alternativa também seja **unit**, contemplando assim a possibilidade de o valor do teste ser verdadeiro ou falso, com resultados do mesmo tipo para a expressão condicional.

```
# if true then print_endline "ok";;
ok
- : unit = ()
# if true then 10;;
Error: This expression has type int but an expression was expected of type
      unit
```

Tarefa 3.1

Determine o valor e o tipo das expressões seguintes caso a expressão esteja correta. Se a expressão estiver incorreta, indique qual é o problema encontrado.

- `if sqrt (abs_float (10.0 -. 35.0) *. 100.0) < 5.0 then "aceito" else "negado"`
- `if pred (int_of_char 'B') then 10 else 20`
- `if 1 mod 2 <> 0 then sqrt 0.09 else Char.lowercase 'B'`
- `4 * if 'B' < 'A' then 2 + 3 else 2 - 3`
- `compare (if 'B' < 'A' then 2 + 3 else 2) 3`
- `if 5 > 0 then "minas gerais"`
- `if 5 > 0 then print_string "minas gerais"`

3.2 Definição de função com expressão condicional

Como na maioria das linguagens de programação, *funções* podem ser *definidas usando expressões condicionais*. Por exemplo, a função para calcular o valor absoluto de um número inteiro pode ser definida como segue:

```
let valorAbsoluto n =
  if n >= 0 then n else -n
```

`valorAbsoluto` recebe um inteiro `n` e resulta em `n` se ele é não-negativo, e `-n` caso contrário.

Expressões condicionais podem ser *aninhadas*, como mostra o exemplo a seguir onde é definida uma função para determinar o sinal de um número inteiro.

```

let sinal n =
  if n < 0 then
    -1
  else
    if n = 0 then
      0
    else
      1

```

Veja outro exemplo onde é definida uma função para análise do índice de massa corporal de uma pessoa:

```

let analisaIMC imc =
  if imc <= 18.5 then
    "Você está abaixo do peso, seu emo!"
  else if imc <= 25.0 then
    "Você parece normal. Deve ser feio!"
  else if imc <= 30.0 then
    "Você está gordo! Perca algum peso!"
  else
    "Você está uma baleia. Parabéns!"

```

Tarefa 3.2: Maior de três valores

Defina uma função `max3` que recebe três valores e resulta no maior deles. Use expressões condicionais aninhadas.

Teste sua função no ambiente interativo.

Tarefa 3.3: Conceito obtido por estudante

O conceito obtido por um estudante a partir da média ponderada de suas notas (no intervalo $[0 - 10]$) é dada pela tabela seguinte:

média ponderada	conceito
$[8, 10]$	A
$[7, 8[$	B
$[6, 7[$	C
$[5, 6[$	D
$[0, 5[$	E

Defina uma função `conceito_obtido` do tipo `float -> char` que recebe a média ponderada obtida por um estudante e resulta no conceito.

Teste sua função no ambiente interativo do OCaml.

4 DEFINIÇÕES LOCAIS

Resumo

Nesta aula vamos aprender a fazer definições locais a uma expressão usando a expressão **let**.

Sumário

4.1	Expressão let	4-1
4.2	Exercícios	4-4

4.1 Expressão let

Em muitas situações é desejável poder definir valores e funções auxiliares na avaliação de uma expressão principal. Isto pode ser feito escrevendo-se **expressões let**.

Uma **expressão let** é formada por uma definição de variável ou função, e por um corpo (que é uma expressão), introduzidos pelas palavras chave **let** e **in**:

let definição in expressão

O **escopo** dos nomes definidos em uma expressão let restringe-se ao corpo da própria expressão let. O **tipo** de uma expressão let é o tipo do seu corpo. O **valor** de uma expressão let é o valor do seu corpo, calculado em um contexto que inclui os nomes introduzidos na definição local.

Veja alguns exemplos:

```
# let x = 3+2 in (x+1)*(x-1) + 2*x - 4;;  
- : int = 30  
  
# let quadrado x = x*x in quadrado 5 + quadrado 3;;  
- : int = 34
```

A expressão let é uma **expressão**, e portanto *sempre tem um valor*. Assim uma expressão let pode ser usada *dentro de outra expressão*. Por exemplo:

```
# 5 * (let x = 3+2 in (x+1)*(x-1) + 2*x - 4) + 1;;  
- : int = 151
```

Expressões let podem ser aninhadas, como no exemplo a seguir:

```
# let x = 3+2 in let y = 5-1 in x*x + 2*x - y;;  
- : int = 31
```

Na tabela 2.1 vemos que a expressão **let**, juntamente com as expressões **match**, **fun**, **function** e **try** tem a menor precedência. Portanto o corpo de uma expressão let avança para a direita tanto quanto possível.

Lembre-se que os nomes definidos em uma expressão local são válidos apenas no corpo da expressão let, como ilustra o exemplo a seguir:

```
# (let x = 3+2 in x*x + 2*x - 1) + 3*x;;  
Error: Unbound value x
```

Se a variável sendo definida na expressão let já existir no escopo em que a expressão let é usada, a variável local criada esconde a variável já existente. Por exemplo:

```
# let x = 1001;;  
val x : int = 1001  
  
# (let x = 2 in x * x * 4) + x;;
```

```
- : int = 1017  
  
# x;;  
- : int = 1001
```

Observe que em OCaml a mesma palavra chave **let** é usada para fazer definições de variáveis e funções tanto locais quanto globais. A palavra chave **in** é usada somente no caso de definições locais.

Expressões **let** são interessantes para dividir uma expressão maior em subexpressões menores que são nomeadas. Os nomes são então usados na expressão principal. Isto pode aumentar a legibilidade do código.

Expressões **let** também são interessantes quando uma determinada subexpressão é usada várias vezes no código. Ao dar um nome à subexpressão em uma expressão **let** evitamos que a subexpressão seja escrita e avaliada várias vezes.

Na prática expressões **let** são usadas para criar variáveis ou funções locais dentro de outras funções. Digamos que precisamos de uma função para calcular a raiz *positiva* de uma equação do segundo grau, se houver (se não houver, retorna o valor *infinito*):

```
let raiz_positiva a b c =  
  if (b *. b -. 4.0 *. a *. c) >= 0.0 then  
    (-.b +. sqrt (b *. b -. 4.0 *. a *. c)) /. (2.0 *. a)  
  else  
    infinity
```

Além da repetição do cálculo do discriminante na fórmula da equação do segundo grau, o cálculo da raiz com uma expressão muito longa torna o código mais difícil de entender. Usando uma variável local para o discriminante (normalmente chamado de *delta*), a mesma função fica mais legível:

```
let raiz_positiva a b c =  
  let delta = b *. b -. 4.0 *. a *. c in  
  if delta >= 0.0 then  
    (-.b +. sqrt delta) /. (2.0 *. a)  
  else  
    infinity
```

Tipicamente a formatação da expressão **let** é feita colocando a definição em uma linha começando com **let** e terminando com **in**, e o resto do código continuando na linha seguinte, com a mesma indentação.

O exemplo seguinte usa uma expressão **let** na definição de uma função que calcula a área da superfície de um cilindro, dados o raio da base e a altura do cilindro:

```
let areaSuperfCilindro r h =  
  let pi = acos (-1.0) in  
  let areaLado = 2.0 *. pi *. r *. h in  
  let areaBase = pi *. r**2.0 in  
  areaLado +. 2.0*.areaBase
```

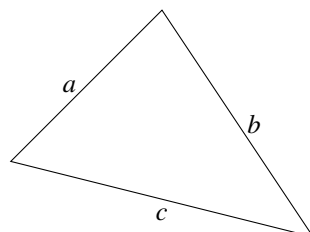
Como outro exemplo, considere a fórmula de Heron

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

para calcular a área de um triângulo com lados *a*, *b* e *c*, sendo

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

o semiperímetro do triângulo. Como *s* aparece várias vezes na fórmula, podemos defini-lo localmente uma única vez e usá-lo quantas vezes forem necessárias.



```
let areaTriangulo a b c =
  let s = (a +. b +. c) /. 2.0 in
  sqrt (s *. (s -. a) *. (s -. b) *. (s -. c))
```

Esta definição assume que os argumentos da função são valores válidos para os lados de um triângulo.

```
# areaTriangulo 5. 6. 8.;;
- : float = 14.9812382665786341
```

Uma definição mais elaborada para esta função somente calcula a área se os argumentos forem lados de um triângulo (um lado deve ser positivo e menor do que a soma dos outros dois lados):

```
let areaTriangulo a b c =
  if a > 0.0 && b > 0.0 && c > 0.0 &&
    a < b +. c &&
    b < a +. c &&
    c < a +. b then
    let s = (a +. b +. c) /. 2.0 in
    sqrt (s *. (s -. a) *. (s -. b) *. (s -. c))
  else
    0.0
```

Veja outro exemplo de definição local:

```
let g x y =
  let square x = x*x in
  let a = square (y+1) in
  if x <= 10 then
    x + a
  else if x > 10 then
    x - a
  else
    x
```

O próximo exemplo mostra uma função para análise do índice de massa corporal.

```
let analisaIMC peso altura =
  let imc = peso /. altura ** 2.0 in
  if imc <= 18.5 then "Você está abaixo do peso, seu emo!"
  else if imc <= 25.0 then "Você parece normal. Deve ser feio!"
  else if imc <= 30.0 then "Você está gordo! Perca algum peso!"
  else "Você está uma baleia. Parabéns!"
```

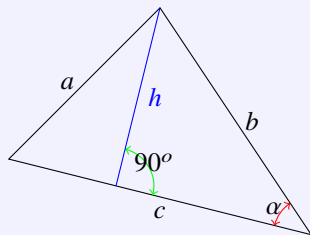
Ou ainda:

```
let analisaIMC peso altura =
  let imc = peso /. altura ** 2.0 in
  let magro = 18.5 in
  let normal = 25.0 in
  let gordo = 30.0 in
  if imc <= magro then "Você está abaixo do peso, seu emo!"
  else if imc <= normal then "Você parece normal. Deve ser feio!"
  else if imc <= gordo then "Você está gordo! Perca algum peso!"
  else "Você está uma baleia. Parabéns!"
```

4.2 Exercícios

Tarefa 4.1: Área de um triângulo usando relações métricas

A área de um triângulo de lados a , b e c pode ser calculada usando relações métricas em um triângulo qualquer.



Pela lei dos cossenos temos:

$$a^2 = b^2 + c^2 - 2bc \cos \alpha \implies \cos \alpha = \frac{b^2 + c^2 - a^2}{2bc}$$

Pela relação fundamental da trigonometria temos:

$$\sin^2 \alpha + \cos^2 \alpha = 1 \implies \sin \alpha = \sqrt{1 - \cos^2 \alpha}$$

Pela definição de seno temos:

$$\sin \alpha = \frac{h}{b} \implies h = b \sin \alpha$$

Pela definição da área de um triângulo temos:

$$A = \frac{ch}{2}$$

Defina uma função para calcular a área de um triângulo de lados a , b e c usando as equações apresentadas. Caso a , b e c não possam ser lados de um triângulo a função deve resultar em zero.

Dica Faça definições locais para os valores $\cos \alpha$, $\sin \alpha$ e h .

Tarefa 4.2: Número de raízes reais da equação do segundo grau

Defina uma função chamada `numRaizes` que recebe os três coeficientes de uma equação do segundo grau

$$ax^2 + bx + c = 0$$

e calcula a quantidade de raízes reais distintas da equação. Assuma que a equação é não degenerada (isto é, o coeficiente do termo de grau dois não é zero).

Use uma definição local para calcular o discriminante da equação.

$$\Delta = b^2 - 4ac$$

Se Δ for positivo a equação tem duas raízes reais e distintas, se for nulo, a equação tem uma raiz real, e se for negativo, a equação não tem raízes reais.

Tarefa 4.3: Notas de um estudante

A nota final de um estudante é calculada a partir de três notas atribuídas respectivamente a um trabalho de laboratório, a uma avaliação semestral e a um exame final. A média ponderada das três notas mencionadas obedece aos pesos a seguir:

nota	peso
trabalho de laboratório	2
avaliação semestral	3
exame final	5

O programa a seguir, que está incompleto, recebe as três notas e determina e exibe o conceito obtido pelo aluno usando a tabela:

média ponderada	conceito
[8.0 – 10.0]	A
[7.0 – 8.0[	B
[6.0 – 7.0[	C
[5.0 – 6.0[	D
[0.0 – 5.0[	E

Considera-se que as notas digitadas são válidas.

```
let conceito n1 n2 n3 =
  (* complete a definição da função *)

let main =
  print_string "Digite a nota do trabalho de laboratório .... ";
  let laboratorio = read_float () in
  print_string "Digite a nota da avaliação semestral ..... ";
  let semestral = read_float () in
  print_string "Digite a nota do exame final ..... ";
  let final = read_float () in
  print_newline ();
  print_string "Conceito obtido: ";
  print_char (conceito laboratorio semestral final);
  print_newline ()
```

Exemplo de execução da aplicação

```
Digite a nota do trabalho de laboratório ....: 7.8
Digite a nota da avaliação semestral .....: 8.0
Digite a nota do exame final .....: 4.9
```

Conceito obtido: C

Você deve completar a definição do função `conceito`. Use uma definição local para calcular a média.

5 PROGRAMAS INTERATIVOS

Resumo

Programas interativos se comunicam com o usuário recebendo dados e exibindo resultados. Nesta aula vamos aprender como desenvolver programs funcionais que interagem com o usuário.

Sumário

5.1	Interação com o mundo	5-1
5.2	Expressão sequência	5-1
5.2.1	Ignorando o resultado de uma expressão	5-2
5.2.2	Blocos de programa	5-2
5.3	Estrutura de um programa em OCaml	5-2
5.4	Canais de entrada e saída	5-5
5.5	Funções de saída na saída padrão	5-6
5.6	Funções de saída na saída de erro padrão	5-6
5.7	Funções de entrada na entrada padrão	5-6
5.8	Exercícios	5-7

5.1 Interação com o mundo

Programas interativos podem exibir mensagens para o usuário e obter valores informados pelo usuário. De forma geral um programa poderá *trocar informações* com o *restante do sistema computacional* para obter dados do sistema computacional e gravar dados no sistema computacional.

Em *linguagens imperativas* (como **C**) as operações de entrada e saída produzem **efeitos colaterais**, refletidos na *atualização de variáveis globais* que representam o *estado* do sistema de computação.

Já em *linguagens declarativas* as operações de entrada e saída são puras e não produzem efeito colateral visível diretamente no programa. Estas operações são integradas na linguagem usando mecanismos como ações de entrada e saída que podem ser combinadas sequencialmente para formar novas ações, mas que somente são executadas quando passadas para o sistema operacional. Este é o modelo adotado pela linguagem **Haskell**. A linguagem **Clean** já usa um outro modelo que integra ao sistema de tipos da linguagem o conceito de tipos únicos para acomodar a possibilidade de atualização de variáveis como uma otimização do compilador.

Sendo uma linguagem multi-paradigma, **OCaml** suporta a programação imperativa. Em OCaml operações de entrada e saída são implementadas através de produção de efeito colateral. Uma expressão em OCaml, quando avaliada, tem um **valor** e pode ter um **efeito**.

5.2 Expressão sequência

Uma **expressão sequência** é formada por uma sequência de expressões separadas por ponto-e-vírgula (;):

$$exp_1 ; \dots ; exp_n$$

A sua avaliação é feita pela avaliação de cada uma das expressões exp_i na ordem em que foram escritas. O tipo e o valor da expressão sequência são dados respectivamente pelo tipo e pelo valor da última expressão da sequência.

Os valores das expressões da sequência, exceto da última, são descartados. Logo estas expressões serão interessantes somente se elas produzirem algum efeito. Para lembrar que seus valores são descartados, o compilador emite um aviso caso o seu tipo não seja **unit**.

Exemplos:

```
# 2 * 3 ; 2 + 3 ;;
Warning 10: this expression should have type unit.
- : int = 5
```

```
# print_string "resultado: " ; print_int 10 ; print_newline () ; true ;;
resultado: 10
- : bool = true

# print_int 1; 2 ; 3;;
Warning 10: this expression should have type unit.
1- : int = 3
```

5.2.1 Ignorando o resultado de uma expressão

val ignore : 'a -> unit

Descarta o valor de seu argumento e retorna (). Por exemplo, ignore (f x) descarta o resultado da função f, que espera-se produzir algum efeito colateral. É equivalente a f x; (), exceto que este pode gerar um aviso do compilador; por outro lado escrevendo-se ignore (f x) evita-se o aviso.

Exemplos:

```
# ignore (2 * 3);;
- : unit = ()

# ignore (2 * 3); 2 + 3;;
- : int = 5
```

5.2.2 Blocos de programa

Para delimitar um grupo de expressões podemos utilizar parênteses:

(expr)

ou as palavras reservadas begin e end:

begin expr end

Exemplos:

```
# ( print_int (2+3); print_newline (); 2-3 );;
5
- : int = -1

# begin print_int (2+3); print_newline (); 2-3 end;;
5
- : int = -1
```

5.3 Estrutura de um programa em OCaml

Código em OCaml pode ser compilado separadamente e executado de forma não interativa utilizando os compiladores ocamlc e ocamlpt.

O código-fonte deve ser colocado em um arquivo com extensão .ml. Ele consiste de uma sequência de frases, que serão avaliadas em tempo de execução em sua ordem de aparição no arquivo de origem. Ao contrário do modo interativo, tipos e valores não são impressos automaticamente; o programa deve chamar as funções de impressão explicitamente para produzir alguma saída.

Aqui está um exemplo de programa autônomo para imprimir números de Fibonacci:

```
(* File fib.ml *)

let rec fib n =
  if n < 2 then
    1
  else
    fib (n-1) + fib (n-2)

let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int (fib arg);
  print_newline ();

let _ = main ()
```

`Sys.argv` é um array de strings contendo os parâmetros da linha de comando. `Sys.argv.(1)` é, assim, o primeiro parâmetro da linha de comando.

O programa acima é compilado e executado com os seguintes comandos no shell:

```
$ ocamlc -o fib fib.ml

$ ./fib 10
89

$ ./fib 20
10946
```

Programas OCaml autônomos mais complexos são geralmente compostos de vários arquivos fonte, e podem se ligar a bibliotecas pré-compiladas. A recompilação de projetos OCaml multi-arquivos pode ser automatizada usando o gerenciador de compilação `ocamlbuild`.

Tarefa 5.1: Preparando e executando um programa no terminal

1. Grave o código fonte do programa apresentado em um arquivo texto chamado `fib.ml`.

```
(* File fib.ml *)

let rec fib n =
  if n < 2 then
    1
  else
    fib (n-1) + fib (n-2)

let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int (fib arg);
  print_newline ()

let _ = main ()
```

2. Compile o programa usando o compilador `ocamlc` em um terminal:

```
$ ocamlc -o fib fib.ml
```

3. Execute o programa já compilado:

```
$ ./fib 10
89
```

4. Compile o programa usando o compilador `ocamlopt` em um terminal e execute o programa:

```
$ ocamlopt -o fib fib.ml

$ ./fib 10
89
```

5. Compile o programa usando o gerenciador de compilação `ocamlbuild` em um terminal gerando bytecodes e execute o programa:

```
$ ocamlbuild fib.byte

$ ./fib.byte 10
89
```

6. Compile o programa usando o gerenciador de compilação `ocamlbuild` em um terminal gerando código nativo e execute o programa:

```
$ ocamlbuild fib.native

$ ./fib.native 10
89
```

Tarefa 5.2: Preparando e executando um programa em OCaml no Eclipse

Use o ambiente de desenvolvimento integrado OcaIDE para preparar e executar o seguinte programa:

```
let _ =  
  print_endline "Somando dois números:";  
  print_endline "=====";  
  print_string "  digite um número inteiro.....: ";  
  let x = read_int () in  
  print_string "  digite outro número inteiro....: ";  
  let y = read_int () in  
  let z = x + y in  
  print_string "  soma calculada.....: ";  
  print_int z;  
  print_newline ()
```

1. Crie um novo projeto: File→New→Project....
2. Selecione o tipo de projeto OCaml Project (ocamlbuild).
3. Clique Next.
4. Digite o nome do projeto Somando no campo Project name.
5. Clique Finish.
6. Crie um novo módulo: File→New→Module.
7. Digite o nome do módulo soma no campo File name.
8. Clique Finish.
9. No editor digite o código do programa.
10. Configure o projeto para compilar o módulo. Para isto acesse o menu de contexto do projeto: Properties ou então no menu principal Project→Properties.
11. Selecione o item Project.
12. No campo Target (.native, .byte, .d.byte, .cma, ...) indique o módulo principal para gerar o programa executável: soma.byte (para compilar para bytecodes) ou soma.native (para compilar para código nativo).
13. Sempre que o código fonte for alterado e salvo, ele será automaticamente recompilado.
14. Para executar o programa acione no menu principal: Run→Run. A aba Console será usada para interação com o usuário.

5.4 Canais de entrada e saída

Os dispositivos de entrada e saída em OCaml são representados por **canais de entrada e saída**:

type in_channel

Tipo dos canais de entrada.

type out_channel

Tipo dos canais de saída.

Os dispositivos de entrada e saída padrão são representados pelas seguintes variáveis:

val stdin : in_channel

Dispositivo de entrada padrão.

val stdout : out_channel

Dispositivo de saída padrão.

val stderr : out_channel
Dispositivo de saída de erro padrão.

5.5 Funções de saída na saída padrão

val print_char : char -> unit
Imprime um caracter na saída padrão.

val print_string : string -> unit
Imprime uma string na saída padrão.

val print_bytes : bytes -> unit
Imprime uma seqüência de bytes na saída padrão.

val print_int : int -> unit
Imprime um inteiro, em notação decimal, na saída padrão.

val print_float : float -> unit
Imprime um número em ponto flutuante, em notação decimal, na saída padrão.

val print_endline : string -> unit
Imprime uma string, seguido de um caractere de nova linha, na saída padrão e descarrega¹ a saída padrão.

val print_newline : unit -> unit
Imprime um caractere de nova linha na saída padrão, e descarrega a saída padrão. Isso pode ser usado para simular bufferização de linha da saída padrão.

5.6 Funções de saída na saída de erro padrão

val prerr_char : char -> unit
Imprime um caracter na saída de erro padrão.

val prerr_char : char -> unit
Imprime um caracter na saída de erro padrão.

val prerr_string : string -> unit
Imprime uma string na saída de erro padrão.

val prerr_bytes : bytes -> unit
Imprime uma seqüência de bytes na saída de erro padrão.

val prerr_int : int -> unit
Imprime um inteiro, em decimal, na saída de erro padrão.

val prerr_float : float -> unit
Imprime um número em ponto flutuante na saída de erro padrão.

val prerr_endline : string -> unit
Imprime uma string, seguido de um caractere de nova linha na saída de erro padrão e descarrega a saída de erro padrão.

val prerr_newline : unit -> unit
Imprime um caractere de nova linha na saída de erro padrão, e descarrega a saída de erro padrão.

5.7 Funções de entrada na entrada padrão

val read_line : unit -> string
Descarrega a saída padrão, e então lê os caracteres da entrada padrão até que um caractere de nova linha seja encontrado. Retorna a string de todos os caracteres lidos, sem o caractere de nova linha no final.

val read_int : unit -> int
Descarrega a saída padrão, e em seguida lê uma linha da entrada padrão e converte-a para um número inteiro. Lança a exceção **Failure "int_of_string"** se a linha lida não é uma representação válida de um inteiro.

¹Basicamente descarregar um canal de saída consiste em completar as operações de saída pendentes associadas ao canal.

```
val read_float : unit -> float
```

Descarrega a saída padrão, e em seguida lê uma linha da entrada padrão e converte-a para um número em ponto flutuante. O resultado não é especificado se a linha lida não é uma representação válida de um número de ponto flutuante.

5.8 Exercícios

Tarefa 5.3: Produto de três números

Escreva um programa que solicita ao usuário três números em ponto flutuante, lê os números, e calcula e exibe o produto dos números.

Exemplo de execução da aplicação

```
Digite um número: 10.0
Digite outro número: 2.3
Digite outro número: 5.0
Produto dos números digitados: 115.0
```

Tarefa 5.4: Linha de crédito

A prefeitura de Contagem abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto.

Fazer um programa que permita entrar com o salário bruto e o valor da prestação, e informar se o empréstimo pode ou não ser concedido.

Exemplo de execução da aplicação

```
Análise de crédito
-----
Salário bruto: 1000
Valor da prestação: 20
O empréstimo pode ser concedido
```

Exemplo de execução da aplicação

```
Análise de crédito
-----
Salário bruto: 1000
Valor da prestação: 430.23
O empréstimo não pode ser concedido
```

Tarefa 5.5: Classe eleitoral

Crie um programa que leia a idade de uma pessoa e informe a sua classe eleitoral:

não eleitor abaixo de 16 anos;

eleitor obrigatório entre 18 (inclusive) e 65 anos;

eleitor facultativo de 16 até 18 anos e acima de 65 anos (inclusive).

Exemplo de execução da aplicação

Classe eleitoral

Digite a idade da pessoa: 11
não eleitor

Exemplo de execução da aplicação

Classe eleitoral

Digite a idade da pessoa: 17
eleitor facultativo

Exemplo de execução da aplicação

Classe eleitoral

Digite a idade da pessoa: 20
eleitor obrigatório

Exemplo de execução da aplicação

Classe eleitoral

Digite a idade da pessoa: 73
eleitor facultativo

Tarefa 5.6: Situação de um aluno

Faça um programa que receba três notas de um aluno, e calcule e mostre a média aritmética das notas e a situação do aluno, dada pela tabela a seguir.

média das notas	situação
menor que 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

Exemplo de execução da aplicação

Digite as três notas do aluno:
9.1
2.9
7.4
Média: 6.466666666667

Tarefa 5.7: Peso ideal

Escreva um programa que solicita ao usuário a altura (em metros) e o sexo de uma pessoa, lê estes dados, calcula o peso ideal da pessoa segundo a tabela a seguir (onde h é a altura da pessoa), e exibe o resultado.

sexo	peso ideal
masculino	$72.7 \times h - 58$
feminino	$62.1 \times h - 44.7$

Exemplo de execução da aplicação

Altura (em metros).....: 1.72
Sexo (f/m).....: f
Peso ideal.....: 62.112

Exemplo de execução da aplicação

Altura (em metros).....: 1.72
Sexo (f/m).....: m
Peso ideal.....: 67.044

Exemplo de execução da aplicação

Altura (em metros).....: 1.72
Sexo (f/m).....: ?
Sexo inválido

Exemplo de execução da aplicação

Altura (em metros).....: 1.72
Sexo (f/m).....:
Sexo inválido

Tarefa 5.8: Impostos

Faça um programa que apresente o menu a seguir, permita ao usuário escolher a opção desejada, receba os dados necessários para executar a operação, e mostre o resultado.

Opções:

- 1. Imposto
2. Novo salário
3. Classificação

Digite a opção desejada:

Verifique a possibilidade de opção inválida.

Na **opção 1** receba o salário de um funcionário, calcule e mostre o valor do imposto sobre o salário usando as regras a seguir:

salário	taxa de imposto
Abaixo de R\$500,00	5%
De R\$500,00 a R\$850,00	10%
Acima de R\$850,00	15%

Na **opção 2** receba o salário de um funcionário, calcule e mostre o valor do novo salário, usando as regras a seguir:

salário	aumento
Acima de R\$1.500,00	R\$25,00
De R\$750,00 (inclusive) a R\$1.500,00 (inclusive)	R\$50,00
De R\$450,00 (inclusive) a R\$750,00	R\$75,00
Abaixo de R\$450,00	R\$100,00

Na **opção 3** receba o salário de um funcionário e mostre sua classificação usando a tabela a seguir:

salário	classificação
Até R\$750,00 (inclusive)	mal remunerado
Acima de R\$750,00	bem remunerado

Exemplo de execução da aplicação

Análise de salário

Opções:

1. Imposto
2. Novo salário
3. Classificação

Digite a opção desejada: 1

Cálculo do imposto

Digite o salário: 700

Imposto calculado: 70.0

Exemplo de execução da aplicação

Análise de salário

Opções:

1. Imposto
2. Novo salário
3. Classificação

Digite a opção desejada: 2

Cálculo do novo salário

Digite o salário: 700

Novo salário: 775

Exemplo de execução da aplicação

Análise de salário

Opções:

1. Imposto
2. Novo salário
3. Classificação

Digite a opção desejada: 3

Classificação do salário

Digite o salário: 700

Classificação obtida: mal remunerado

Exemplo de execução da aplicação

Análise de salário

Opções:

- 1. Imposto
2. Novo salário
3. Classificação

Digite a opção desejada: 4

Opção inválida!

Tarefa 5.9: Terno pitagórico

Em Matemática um **terno pitagórico** (ou trio pitagórico, ou ainda tripla pitagórica) é formado por três números a , b e c tais que $a^2 + b^2 = c^2$. O nome vem do *teorema de Pitágoras* que afirma que se as medidas dos lados de um triângulo retângulo são números inteiros, então elas formam um terno pitagórico.

Codifique um programa que leia três números positivos e verifique se eles formam um terno pitagórico.

Exemplo de execução da aplicação

Verificação de ternos pitagóricos

Digite o primeiro número positivo: 3
Digite o segundo número positivo: 4
Digite o terceiro número positivo: 5
Os números formam um terno pitagórico

Exemplo de execução da aplicação

Verificação de ternos pitagóricos

Digite o primeiro número positivo: 6
Digite o segundo número positivo: 5
Digite o terceiro número positivo: 4
Os números não formam um terno pitagórico

Exemplo de execução da aplicação

Verificação de ternos pitagóricos

Digite o primeiro número positivo: 3
Digite o segundo número positivo: -4
Digite o terceiro número positivo: 0
Números inválidos

6 FUNÇÕES RECURSIVAS

Resumo

Definições recursivas são comuns na programação funcional. Nesta aula vamos aprender a definir funções recursivas.

Sumário

6.1	Recursividade	6-1
6.2	Recursividade mútua	6-5
6.3	Recursividade de cauda	6-6
6.4	Vantagens da recursividade	6-8
6.5	Exercícios	6-9

6.1 Recursividade

Recursividade é uma idéia *inteligente* que desempenha um papel central na *programação funcional* e na *ciência da computação* em geral. **Recursividade** é o mecanismo de programação no qual *uma definição* de função ou de outro objeto *refere-se ao próprio objeto* sendo definido. Assim **função recursiva** é uma função que é definida em termos de si mesma. São sinônimos: recursividade, recursão, recorrência.

Recursividade é o mecanismo básico para *repetições* nas linguagens funcionais.

Definições recursivas em OCaml são introduzidas pela declaração **let rec**.

Estratégia para a definição recursiva de uma função:

1. *dividir* o problema em problemas menores do mesmo tipo
2. *resolver* os problemas menores (dividindo-os em problemas ainda menores, se necessário)
3. *combinar* as soluções dos problemas menores para formar a solução final

Ao dividir o problema sucessivamente em problemas menores eventualmente os casos simples são alcançados:

- não podem ser mais divididos
- suas soluções são definidas explicitamente

De modo geral, uma **definição de função recursiva** é dividida em duas partes:

- Há um ou mais **casos base** que dizem o que fazer em situações simples, onde não é necessária nenhuma recursão.
 - Nestes casos *a resposta pode ser dada de imediato*, sem chamar recursivamente a função sendo definida.
 - Isso garante que a recursão eventualmente pode *parar*.
- Há um ou mais **casos recursivos** que são mais gerais, e definem a função em termos de uma *chamada* mais simples *a si mesma*.

Como exemplo de função recursiva, considere o cálculo do fatorial de um número natural. A função que calcula o fatorial de um número natural pode ser definida recursivamente como segue:

```
let rec fatorial n =  
  if n = 0 then  
    1  
  else  
    if n > 0 then  
      fatorial (n-1) * n  
    else  
      raise (Invalid_argument "fatorial")
```

Nesta definição:

- A primeira alternativa estabelece que o fatorial de 0 é 1. Este é o *caso base*.
- A segunda alternativa estabelece que o fatorial de um número positivo é o produto deste número e do fatorial do seu antecessor. Este é o *caso recursivo*.

Observe que no caso recursivo o subproblema `fatorial (n-1)` é *mais simples* que o problema original `fatorial n` e está mais próximo do caso base `fatorial 0`.

Aplicando a função fatorial:

```
fatorial 4
  ~> fatorial 3 * 4
  ~> fatorial 2 * 3 * 4
  ~> fatorial 1 * 2 * 3 * 4
  ~> fatorial 0 * 1 * 2 * 3 * 4
  ~> 1 * 1 * 2 * 3 * 4
  ~> 1 * 2 * 3 * 4
  ~> 2 * 3 * 4
  ~> 6 * 4
  ~> 24
```

Em muitas implementações de linguagens de programação uma *chamada de função* usa um espaço de memória (chamado de **quadro**, *frame* ou **registro de ativação**) em uma área da memória (chamada **pilha** ou *stack*) onde são armazenadas informações importantes, tais como:

- argumentos da função
- variáveis locais
- variáveis temporárias
- endereço de retorno da função

Normalmente para cada nova chamada de função que é realizada, um novo quadro é alocado na pilha. Quando a função retorna, o quadro é desalocado.

De maneira bastante simplificada a sequência de alocação de quadros na pilha de execução na chamada da função `fatorial 6` do exemplo anterior é indicada nas figuras seguintes.

n = 4 ~>4 * fatorial 3

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * fatorial 2

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * fatorial 2
n = 2 ~>2 * fatorial 1

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * fatorial 2
n = 2 ~>2 * fatorial 1
n = 1 ~>1 * fatorial 2

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * fatorial 2
n = 2 ~>2 * fatorial 1
n = 1 ~>1 * fatorial 2
n = 0 ~>1

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * fatorial 2
n = 2 ~>2 * fatorial 1
n = 1 ~>1 * 1 = 1

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * fatorial 2
n = 2 ~>2 * 1 = 2

n = 4 ~>4 * fatorial 3
n = 3 ~>3 * 2

n = 4 ~>4 * 6 = 24

Tarefa 6.1: Aplicando fatorial

Digite a função `fatorial` em um arquivo fonte e carregue-o no ambiente interativo.

- Mostre que `fatorial 7 = 5040` usando uma sequência de passos de simplificação.
- Determine o valor da expressão `fatorial 7` usando o ambiente interativo.
- Determine o valor da expressão `fatorial 31` usando o ambiente interativo. Se você tiver uma calculadora científica, verifique o resultado na calculadora. Explique porque os resultados diferem.
- O que acontece ao se calcular o valor da expressão `fatorial (-2)`?

Vejamos outro exemplo. A função que calcula a potência de dois (isto é, a base é dois) para números naturais pode ser definida recursivamente como segue:

```
let rec pot_dois n =  
  if n = 0 then  
    1  
  else  
    if n > 0 then  
      2 * pot_dois (n-1)  
    else  
      raise (Invalid_argument "pot_dois")
```

Nesta definição:

- A primeira alternativa estabelece que $2^0 = 1$. Este é o *caso base*.
- A segunda alternativa estabelece que $2^n = 2 \times 2^{n-1}$, sendo $n > 0$. Este é o *caso recursivo*.

Observe que no caso recursivo o subproblema `pot_dois (n-1)` é *mais simples* que o problema original `pot_dois n` e está mais próximo do caso base `pot_dois 0`.

Aplicando a função potência de dois:

```
pot_dois 4  
↪ 2 * pot_dois 3  
↪ 2 * (2 * pot_dois 2)  
↪ 2 * (2 * (2 * pot_dois 1))  
↪ 2 * (2 * (2 * (2 * pot_dois 0)))  
↪ 2 * (2 * (2 * (2 * 1)))  
↪ 2 * (2 * (2 * 2))  
↪ 2 * (2 * 4)  
↪ 2 * 8  
↪ 16
```

Tarefa 6.2

Considere a seguinte definição para a função potência de dois:

```
let rec pot_dois' n =  
  if n = 0 then  
    1  
  else  
    2 * pot_dois' (n-1)
```

O que acontece ao calcular o valor da expressão `pot_dois' (-3)`?

Vejamos mais um exemplo de definição recursiva. A multiplicação de inteiros está disponível na biblioteca como uma operação primitiva por questões de eficiência. Porém ela pode ser definida usando adição e recursividade em um de seus argumentos:

```

let rec mul m n =
  if n = 0 then
    0
  else if n > 0 then
    m + mul m (n-1)
  else
    - (mul m (- n))

```

Nesta definição:

- A primeira alternativa estabelece que quando o multiplicador é zero, o produto também é zero. Este é o *caso base*.
- A segunda alternativa estabelece que $m \times n = m + m \times (n - 1)$, sendo $n > 0$. Este é um *caso recursivo*.
- A terceira alternativa estabelece que $m \times n = -(m \times (-n))$, sendo $n < 0$. Este é outro *caso recursivo*.

Aplicando a função de multiplicação:

```

% mul 7 (-3)
~> - (mul 7 (- (-3)))
~> - (mul 7 3)
~> - (7 + mul 7 2)
~> - (7 + (7 + mul 7 1))
~> - (7 + (7 + (7 + mul 7 0)))
~> - (7 + (7 + (7 + 0)))
~> - (7 + (7 + 7))
~> - (7 + 14)
~> - 21
~> -21

```

A definição recursiva da multiplicação formaliza a idéia de que a multiplicação pode ser reduzida a adições repetidas.

Tarefa 6.3

Mostre que $\text{mul } 5 \ 6 = 30$.

Vamos analisar outro exemplo de função recursiva: a sequência de Fibonacci. Na sequência de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, ...

os dois primeiros elementos são 0 e 1, e cada elemento subsequente é dado pela soma dos dois elementos que o precedem na sequência. A função a seguir calcula o n -ésimo número de Fibonacci, para $n \geq 0$:

```

let rec fib n =
  if n = 0 then
    0
  else if n = 1 then
    1
  else if n > 1 then
    fib (n-2) + fib (n-1)
  else
    raise (Invalid_argument "fib")

```

Nesta definição:

- A primeira e a segunda alternativa são os *casos base*.
- A terceira alternativa é o *caso recursivo*.

Neste caso temos **recursão múltipla**, pois a função sendo definida é usada mais de uma vez em sua própria definição.

Aplicando a função de fibonacci:

```

fib 5
  ~> fib 3 + fib 4
  ~> (fib 1 + fib 2) + (fib 2 + fib 3)
  ~> (1 + (fib 0 + fib 1)) + ((fib 0 + fib 1) + (fib 1 + fib 2))
  ~> (1 + (0 + 1)) + ((0 + 1) + (1 + (fib 0 + fib 1)))
  ~> (1 + 1) + (1 + (1 + (0 + 1)))
  ~> 2 + (1 + (1 + 1))
  ~> 2 + (1 + 2)
  ~> 2 + 3
  ~> 5

```

Tarefa 6.4

Mostre que $\text{fib } 6 = 8$.

6.2 Recursividade mútua

Recursividade mútua ocorre quando duas ou mais funções são definidas em termos uma da outra. Em OCaml funções mutuamente recursivas devem ser definidas simultaneamente.

Em OCaml **declarações simultâneas** são introduzidas por **let** ou **let rec** e são separadas por **and**. Veja alguns exemplos:

```

# let x = 2 and y = 3 in x + y - 1;;
- : int = 4

# let x = 2 and y = 2*x in x + y - 1;;
Error: Unbound value x

# let x = 2 in let y = 2*x in x + y - 1;;
- : int = 5

```

Para exemplificar funções mutuamente recursivas vamos definir as funções par e ímpar que determinam se um número é par ou ímpar, respectivamente. Uma definição eficiente destas funções usa o resto da divisão por dois:

```

let par n =
  n mod 2 = 0

let impar n =
  n mod 2 <> 0

```

No entanto elas também podem ser definidas usando recursividade mútua:

```

let rec par n =
  if n = 0 then
    true
  else if n > 0 then
    impar (n-1)
  else
    par (-n)

and impar n =
  if n = 0 then
    false
  else if n > 0 then
    par (n-1)
  else
    impar (-n)

```

Nestas definições observamos que:

- Zero é par, mas não é ímpar.
- Um número positivo é par se seu antecessor é ímpar.
- Um número positivo é ímpar se seu antecessor é par.
- Um número negativo é par (ou ímpar) se o seu oposto for par (ou ímpar).

Aplicando as função par e ímpar:

```
par (-5)
  ~> par 5
  ~> impar 4
  ~> par 3
  ~> impar 2
  ~> par 1
  ~> impar 0
  ~> false
```

6.3 Recursividade de cauda

Uma função recursiva apresenta **recursividade de cauda** se o *resultado final* da chamada recursiva é o resultado final da própria função. Se o resultado da chamada recursiva deve ser processado de alguma maneira para produzir o resultado final, então a função não apresenta recursividade de cauda.

Por exemplo, a função recursiva a seguir não apresenta recursividade de cauda:

```
let rec fatorial n =
  if n = 0 then
    1
  else
    if n > 0 then
      fatorial (n-1) * n
    else
      raise (Invalid_argument "fatorial")
```

No caso recursivo, o resultado da chamada recursiva `fatorial (n-1)` é multiplicado por `n` para produzir o resultado final.

A função recursiva a seguir também não apresenta recursividade de cauda:

```
let rec par n =
  if n = 0 then
    true
  else if n > 0 then
    not (par (n-1))
  else
    raise (Invalid_argument "par")
```

No caso recursivo, a função `not` é aplicada ao resultado da chamada recursiva `par (n-1)` para produzir o resultado final.

Já a função recursiva `pdois` a seguir apresenta recursividade de cauda:

```
let pdois n =
  let rec pdois' x y =
    if x = 0 then
      y
    else if x > 0 then
      pdois' (x-1) (2*y)
    else
      raise (Invalid_argument "pdois'")
  in
  pdois' n 1
```

No caso recursivo, o resultado da chamada recursiva `pdois (n-1) (2*y)` é o resultado final.

Tarefa 6.5

Mostre que `potencia_dois 5 = 32`.

Tarefa 6.6

Faça uma definição recursiva da função par usando recursividade de cauda.

Otimização de chamada de cauda

Uma **chamada de cauda** acontece quando uma função chama outra função como sua *última ação*, não tendo mais nada a fazer além desta última ação. O *resultado final* da função é dado pelo resultado da chamada de cauda. Em tais situações o programa não precisa voltar para a função que chama quando a função chamada termina. Portanto, após a chamada de cauda, o programa não precisa manter qualquer informação sobre a função chamadora na pilha.

Algumas implementações de linguagem tiram proveito desse fato e na verdade não utilizam qualquer espaço extra de pilha quando fazem uma chamada de cauda, pois o espaço na pilha da própria função chamadora é reutilizado, já que ele não será mais necessário para a função chamadora. Esta técnica é chamada de **eliminação da cauda**, **otimização de chamada de cauda** ou ainda **otimização de chamada recursiva**. A **otimização de chamada de cauda** permite que funções com recursividade de cauda recorram indefinidamente *sem estourar a pilha*.

A figura a seguir ilustra a situação da pilha de execução na avaliação da expressão `pdois 3`, sem otimização de chamada de cauda.

pdois 3 n = 3 ~~pdois' n 1	pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 1 ~~pdois' (x-1) (2*y)	pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 1 ~~pdois' (x-1) (2*y) pdois' 2 2 x = 2 y = 2 ~~pdois' (x-1) (2*y)	pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 5 ~~pdois' (x-1) (2*y) pdois' 2 2 x = 2 y = 2 ~~pdois' (x-1) (2*y) pdois' 1 4 x = 1 y = 4 ~~pdois' (x-1) (2*y)	pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 5 ~~pdois' (x-1) (2*y) pdois' 2 2 x = 2 y = 2 ~~pdois' (x-1) (2*y) pdois' 1 4 x = 1 y = 4 ~~pdois' (x-1) (2*y) pdois' 0 8 x = 0 y = 8 ~~y ~~8
----------------------------------	--	--	--	---

pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 5 ~~pdois' (x-1) (2*y) pdois' 2 2 x = 2 y = 2 ~~pdois' (x-1) (2*y) pdois' 1 4 x = 1 y = 4 ~~pdois' (x-1) (2*y) ~~8	pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 5 ~~pdois' (x-1) (2*y) pdois' 2 2 x = 2 y = 2 ~~pdois' (x-1) (2*y) ~~8	pdois 3 n = 3 ~~pdois' n 1 pdois' 3 1 x = 3 y = 5 ~~pdois' (x-1) (2*y) ~~8	pdois 3 n = 3 ~~pdois' n 1 ~~8
---	---	---	--------------------------------------

Já com otimização de chamada de cauda, as chamadas recursivas reaproveitam o quadro da função chamadora, não havendo necessidade de crescimento da pilha.

pdois 3 n = 3 ~~pdois' n 1	pdois' 3 1 x = 3 y = 1 ~~pdois' (x-1) (2*y)	pdois' 2 2 x = 2 y = 2 ~~pdois' (x-1) (2*y)	pdois' 1 4 x = 1 y = 4 ~~pdois' (x-1) (2*y)	pdois' 0 8 x = 0 y = 8 ~~y ~~8
----------------------------------	--	--	--	---

Estruturas de repetição

Muitas linguagens funcionais não possuem *estruturas de repetição* e usam funções recursivas para fazer repetições. Nestes casos a otimização de chamada de cauda é fundamental para uma boa eficiência dos programas.

Em particular o núcleo funcional da linguagem OCaml não usa estruturas de repetição.

Por exemplo, considere uma aplicação para calcular e exibir a soma de uma sequência de números reais não negativos. Os números da sequência serão obtidos da entrada padrão. A sequência será encerrada com um valor negativo. Este valor negativo é chamado de **sentinela** e é um valor de entrada inválido, usado para indicar o final da entrada, e não faz parte da sequência.

Exemplo de execução da aplicação

```

Digite uma sequência de números (um por linha).
Para terminar digite um número negativo.
1
3
5
0
17
-3
Soma: 26.000000

```

Um programa imperativo para resolver este problema tipicamente é estruturado usando algum comando de repetição. Porém um programa funcional tipicamente faz uso de funções recursivas.

```

let rec ler_e_somar soma =
  let x = read_float () in
  if x < 0.0 then
    soma
  else
    ler_e_somar (soma +. x)

let _ =
  print_endline "Digite uma sequência de números (um por linha).";
  print_endline "Para terminar digite um número negativo.";
  let total = ler_e_somar 0.0 in
  Printf.printf "Soma: %f\n" total

```

6.4 Vantagens da recursividade

A recursividade permite que:

- muitas funções possam ser naturalmente definidas em termos de si mesmas,
- propriedades de funções definidas usando recursão podem ser provadas usando **indução**, uma técnica matemática simples, mas poderosa.

6.5 Exercícios

Tarefa 6.7: Fatorial duplo

O fatorial duplo de um número natural n é o produto de todos os números de 1 (ou 2) até n , contados de 2 em 2. Por exemplo, o fatorial duplo de 8 é $8 \times 6 \times 4 \times 2 = 384$, e o fatorial duplo de 7 é $7 \times 5 \times 3 \times 1 = 105$. Defina uma função para calcular o fatorial duplo usando recursividade.

Tarefa 6.8: Multiplicação em um intervalo

Defina uma função recursiva que recebe dois números naturais m e n , e retorna o produto de todos os números no intervalo $[m, n]$:

$$m \times (m + 1) \times \cdots \times (n - 1) \times n$$

Tarefa 6.9: Fatorial

Usando a função definida no exercício 6.8, escreva uma definição não recursiva para calcular o fatorial de um número natural.

Tarefa 6.10: Adição

Defina uma função recursiva para calcular a soma de dois números naturais, sem usar os operadores $+$ e $-$. Utilize as funções `succ` e `pred` da biblioteca, que calculam respectivamente o sucessor e o antecessor de um valor.

Tarefa 6.11: Potenciação

Defina uma função recursiva para calcular a potência de um número, considerando que o expoente é um número natural. Utilize o método das multiplicações sucessivas:

$$x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ vezes}}$$

Tarefa 6.12: Raiz quadrada inteira

A raiz quadrada inteira de um número inteiro positivo n é o maior número inteiro cujo quadrado é menor ou igual a n . Por exemplo, a raiz quadrada inteira de 15 é 3, e a raiz quadrada inteira de 16 é 4.

Defina uma função recursiva para calcular a raiz quadrada inteira.

Tarefa 6.13: Quociente e resto da divisão inteira

Defina duas funções recursivas que calculam o quociente e o resto da divisão inteira de dois números naturais usando subtrações sucessivas.

Tarefa 6.14: Máximo divisor comum

Defina uma função recursiva para calcular o máximo divisor comum de dois números inteiros não negativos a e b , usando o algoritmo de Euclides:

$$\text{mdc}(a, b) = \begin{cases} a & \text{se } b = 0, \\ \text{mdc}(b, a \bmod b) & \text{se } b > 0, \\ \text{mdc}(a, -b) & \text{se } b < 0 \end{cases}$$

Tarefa 6.15: Fatorial

Considere a seguinte função para calcular o fatorial de um número:

```
let fat n =  
  let rec fat_aux x f =  
    if x = 0 then  
      f  
    else if x > 0 then  
      fat_aux (x-1) (x*f)  
    else  
      raise (Invalid_argument "fat_aux")  
  in  
    fat_aux n 1
```

- Mostre que `fat 6 = 720`.
- Compare o cálculo de `fat 6` com o cálculo de `fatorial 6` apresentado anteriormente. Qual versão da função fatorial é mais eficiente: `fatorial` ou `fat`? Explique.

Tarefa 6.16: Sequência de Fibonacci

Defina uma função com recursividade de cauda para calcular o n -ésimo ($n \geq 0$) número de Fibonacci.

Tarefa 6.17: Soma de uma sequência

Faça um programa que leia um número natural n , e então leia outros n números e calcule e exiba a soma destes números.

Exemplo de execução da aplicação

```
Quantidade de números: 4  
Digite um número: 10  
Digite um número: -5  
Digite um número: 1  
Digite um número: 20  
Soma dos números digitados: 26.000000
```

Exemplo de execução da aplicação

```
Quantidade de números: -6  
Soma dos números digitados: 0.000000
```

Tarefa 6.18: Média aritmética de uma sequência de números

Faça um programa que leia uma sequência de números não negativos e determine a média aritmética destes números. A entrada dos números deve ser encerrada com um número inválido (negativo).

Exemplo de execução da aplicação

```
Cálculo da média aritmética
-----
Digite uma sequência de números (um por linha)
Para terminar digite um valor negativo
10
9
8
9.2
-1
A média dos números digitados é 9.171428571428573
```

Exemplo de execução da aplicação

```
Cálculo da média aritmética
-----
Digite uma sequência de números (um por linha)
Para terminar digite um valor negativo
-5
Sequência vazia
```

Tarefa 6.19: Perda de massa por radioatividade

Um elemento químico radioativo perde sua massa de acordo com a função

$$m(t) = m_0 e^{-kt}$$

onde, t é o tempo (em segundos), m_0 é a massa inicial (em gramas) e k é a constante 5×10^{-2} .

Faça uma aplicação que, dada a massa inicial desse elemento, calcule a perda de massa durante um minuto, exibindo as massas resultantes em intervalos de 10 segundos.

Tarefa 6.20: Cálculo aproximado de π

A série abaixo converge para o número π quando $n \rightarrow \infty$.

$$4 \sum_{i=0}^n \frac{(-1)^i}{2i+1}$$

Codifique um programa que solicita ao usuário o número de parcelas da série e calcula e exibe o valor aproximado de π usando o número solicitado de parcelas.

Tarefa 6.21: Aumento salarial

Um funcionário de uma empresa recebe aumento salarial anualmente. O primeiro aumento é de 1,2% sobre seu salário inicial. Os aumentos subsequentes sempre correspondem ao dobro do percentual de aumento do ano anterior. Faça uma aplicação onde o usuário deve informar o salário inicial do funcionário, o ano de contratação e o ano atual, e calcula e exibe o seu salário atual.

Tarefa 6.22: Fechamento de notas de uma disciplina

Faça uma aplicação para fechamento das notas de uma disciplina. Cada aluno recebe uma nota para cada uma das três atividades desenvolvidas. O usuário deverá informar a quantidade de alunos na turma, e em seguida as notas de cada aluno. Calcule e exiba:

- a média aritmética das três notas de cada aluno,
- a situação do aluno, dada pela tabela seguinte

média aritmética	situação
até 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

- a média da turma
- o percentual de alunos aprovados
- o percentual de alunos em exame especial
- o percentual de alunos reprovados

7 VALORES ALEATÓRIOS

Resumo

A geração de valores pseudo-aleatórios em aplicações em OCaml pode ser feita através da biblioteca `Random`. Nesta aula vamos aprender a desenvolver aplicações que usam valores aleatórios. Estes tópicos serão usados na implementação do jogo *adivinha o número*.

Sumário

7.1	Valores aleatórios	7-1
7.2	Exemplo: lançamento de dados	7-1
7.3	Jogo adivinha o número	7-2
7.4	Jogo craps	7-6

7.1 Valores aleatórios

O módulo `Random` da biblioteca padrão do OCaml lida com a tarefa comum de geração de valores pseudo-aleatórios em OCaml.

As funções básicas para geração de números pseudo-aleatórios são:

val init : int -> unit

Inicializa o gerador, utilizando o argumento como uma semente. A mesma semente sempre produzirá a mesma sequência de números.

val self_init : unit -> unit

Inicializa o gerador com uma semente aleatória escolhida de uma forma dependente do sistema. Se `/dev/urandom` está disponível no computador, ele é utilizada para fornecer uma semente inicial altamente aleatória. Caso contrário, uma semente menos aleatória é calculada a partir de parâmetros do sistema (tempo atual, processo de IDs).

val int : int -> int

`Random.int bound` retorna um inteiro aleatório entre 0 (inclusive) e bound (exclusive). bound deve ser maior do que 0 e menor do que 2^{30} .

val float : float -> float

`Random.float bound` retorna um número aleatório de ponto flutuante entre 0 e bound (inclusive). Se bound é negativo, o resultado é negativo ou zero. Se bound é 0, o resultado é 0.

val bool : unit -> bool

`Random.bool ()` retorna verdadeiro ou falso com probabilidade 0,5 cada.

7.2 Exemplo: lançamento de dados

lancadados.ml

```
let _ =
  print_endline "Lançamento de dois dados";
  Random.self_init ();
  let x = Random.int 6 + 1 in
  let y = Random.int 6 + 1 in
  Printf.printf "Faces obtidas: %d e %d\n" x y
```

Exemplo de execução da aplicação

```
$ ./lancadados  
Lancamento de dois dados  
Faces obtidas: 3 e 5
```

Exemplo de execução da aplicação

```
$ ./lancadados  
Lancamento de dois dados  
Faces obtidas: 4 e 1
```

7.3 Jogo adivinha o número

Tarefa 7.1: Adivinha o número

Escreva uma aplicação para jogar o jogo *adivinha o número*, como explicado a seguir.

- O programa escolhe um número a ser adivinhado pelo jogador (usuário) selecionando um número inteiro aleatório no intervalo de 1 a 1000.
- O programa exibe a mensagem *Adivinhe um número entre 1 e 1000*.
- O jogador informa o seu palpite.
- Se o palpite do jogador estiver incorreto:
 - o programa exibe a mensagem *Muito alto* ou *Muito baixo* convenientemente para ajudar o jogador a acertar o número nas próximas jogadas.
 - o jogo continua com o programa solicitando o próximo palpite e analisando a resposta do usuário.
- Quando o jogador insere a resposta correta:
 - o programa exibe a mensagem *Parabéns, você adivinhou o número*, e
 - permite que o usuário escolha se quer jogar novamente, e joga novamente em caso afirmativo.

Exemplo de execução da aplicação

```
$ ./advinha
Adivinha o número v1.0
=====
Digite um número entre 1 e 1000: 444
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 200
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 111
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 157
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 138
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 123
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 130
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 125
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 128
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 129
Parabéns, você acertou

Deseja jogar novamente? n
```

1. Defina uma função `sim_ou_nao` que recebe uma string e interage com o usuário da seguinte forma:

- exibe a string na saída padrão (com o objetivo de fazer uma pergunta do tipo *sim ou não* ao usuário)
- lê a resposta do usuário
- verifica se a resposta é
 - *s* ou *S*, retornando verdadeiro
 - *n* ou *N*, retornando falso
 - qualquer outra coisa, chamando `sim_ou_nao` novamente para que o usuário responda corretamente.

Use uma expressão **match**.

Exemplo de execução da aplicação

```
# sim_ou_nao "Quer jogar novamente?";;
Quer jogar novamente? talvez
Quer jogar novamente? k
Quer jogar novamente? s
- : bool = true

# sim_ou_nao "Você é inteligente?";;
Você é inteligente? com certeza
Você é inteligente?
Você é inteligente? acho que sim
Você é inteligente? n
- : bool = false
```

Esta função deve ser usada em `jogar` (item 3) para verificar se o usuário deseja continuar jogando ou não.

2. Defina uma função `acertar` que recebe um número a ser adivinhado e interage com o usuário da seguinte forma:

- exibe uma mensagem solicitando um número entre 1 e 1000
- lê o número informado pelo usuário
- compara o número informado com o número a ser adivinhado:
 - se forem iguais, exibe uma mensagem parabenizando o usuário por ter adivinhado o número
 - caso contrário
 - * exibe uma mensagem informando que o número é muito pequeno ou muito grande, adequadamente
 - * exibe uma mensagem solicitando ao usuário uma nova tentativa
 - * faz uma nova tentativa através de uma chamada recursiva de `acertar`

Exemplo de execução da aplicação

```
# acertar 119;;
Digite um número entre 1 e 1000: 600
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 23
Muito pequeno.
Tente novamente.

Digite um número entre 1 e 1000: 119
Parabéns, você acertou!
- : unit = ()
```

A função `acertar` deverá ser usada na definição de `jogar` (item 3).

3. O programa deve permitir ao usuário jogar várias vezes, o que nos leva à necessidade do uso de recursão.

Defina uma função `jogar` que recebe a tupla vazia como argumento e interage com o usuário da seguinte forma:

- gera um número inteiro aleatório entre 1 e 1000, inclusive
- interage com o usuário até que o usuário acerte o número (veja o item 2)
- verifica se o usuário deseja jogar novamente (veja o item 1)
 - se sim, chama `jogar` recursivamente
 - se não, resulta na tupla vazia.

Exemplo de execução da aplicação

```
# jogar ();;
Digite um número entre 1 e 1000: 509
Muito pequeno.
Tente novamente.

Digite um número entre 1 e 1000: 780
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 640
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 590
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 544
Muito pequeno.
Tente novamente.

Digite um número entre 1 e 1000: 577
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 550
Muito pequeno.
Tente novamente.

Digite um número entre 1 e 1000: 560
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 555
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 553
Muito grande.
Tente novamente.

Digite um número entre 1 e 1000: 551
Muito pequeno.
Tente novamente.

Digite um número entre 1 e 1000: 552
Parabéns, você acertou!

Deseja jogar novamente? n
- : unit = ()
```

A função `jogar` deve ser usada para que o usuário possa jogar o jogo (item 4).

4. Defina uma função `main` do tipo **unit** -> **unit** que interage com o usuário da seguinte forma:

- exibe uma mensagem identificando o programa e sua versão,
- inicializa o gerador de números aleatórios, e
- usa a função `jogar` (veja o item 3) para jogar o jogo *adivinha o número*.

Chame a função `main` como última ação no seu programa.

Tarefa 7.2: Adivinha o número com argumentos na linha de comando

Modifique o programa `adivinha.ml` (tarefa 7.1) de forma que o usuário possa especificar o intervalo a ser utilizado para adivinhar o número através de dois argumentos na linha de comando.

Tarefa 7.3: Adivinha o número com contagem de tentativas

Modifique o programa `adivinha.ml` (tarefa 7.2) para que seja exibida o número de tentativas feitas pelo usuário toda vez que lhe é solicitado uma nova tentativa de adivinhar um número.

7.4 Jogo craps

Tarefa 7.4: Craps

Craps é um jogo de azar popular que é jogado em cassinos e nas ruas de todo o mundo. As regras do jogo são simples e diretas.

O jogador lança dois dados. Cada dado tem seis faces numeradas de um a seis. Depois que os dados param de rolar, a soma dos pontos nas faces viradas para cima é calculada. Se a soma for 7 ou 11 no primeiro lance, o jogador ganha. Se a soma for 2, 3 ou 12 no primeiro lance (chamado *craps*), o jogador perde (isto é, a *casa* ganha). Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se a *pontuação alvo* do jogador. Neste caso, para ganhar, o jogador deve continuar a lançar os dados até *fazer sua pontuação alvo* (isto é, obter um valor igual à sua pontuação alvo, determinada no primeiro lance). O jogador perde se obtiver um 7 antes de fazer sua pontuação alvo.

Escreva uma aplicação em OCaml para jogar o jogo *craps* com o usuário.

Dica 1 As únicas interações com o usuário (jogador) deverão ser para:

- solicitar que ele tecle ENTER para lançar os dados,
- exibir os pontos obtidos nas faces dos dados em cada lançamento juntamente com a pontuação (soma dos pontos nas faces dos dados), e
- exibir o resultado final (ganhou ou perdeu).

Dica 2 Exemplos de possíveis execuções do programa:

Craps v1.0

Tecle ENTER para lançar os dados
Faces dos dados: 1 e 6
Pontuação: 7
Você ganhou. Parabéns!

Craps v1.0

Tecle ENTER para lançar os dados
Faces dos dados: 6 e 6
Pontuação: 12
CRAPS!
Você perdeu. Sinto muito!

Craps v1.0

Tecle ENTER para lançar os dados
Faces dos dados: 3 e 3
Pontuação: 6

Tecle ENTER para lançar os dados novamente
Faces dos dados: 1 e 6

Pontuação: 7
Você perdeu. Sinto muito!

Craps v1.0

Tecle ENTER para lançar os dados
Fases dos dados: 4 e 4
Pontuação: 8

Tecle ENTER para lançar os dados novamente
Fases dos dados: 3 e 1
Pontuação: 4

Tecle ENTER para lançar os dados novamente
Fases dos dados: 6 e 2
Pontuação: 8
Você ganhou. Parabéns!

8 TUPLAS, LISTAS, E POLIMORFISMO PARAMÉTRICO

Resumo

Nesta aula vamos conhecer os tipos tuplas e listas, que são tipos polimórficos pré-definidos em OCaml. Vamos aprender também sobre funções polimórficas.

Sumário

8.1	Tuplas	8-1
8.2	Listas	8-2
8.3	Polimorfismo paramétrico	8-4
8.3.1	Operação sobre vários tipos de dados	8-4
8.3.2	Variáveis de tipo	8-4
8.3.3	Valor polimórfico	8-5
8.3.4	Instanciação de variáveis de tipo	8-5
8.4	Funções polimórficas predefinidas	8-5

8.1 Tuplas

Tupla é uma estrutura de dados formada por uma sequência de valores **possivelmente de tipos diferentes**. Os componentes de uma tupla são identificados pela **posição** em que ocorrem na tupla.

Em OCaml uma **expressão tupla** é formada por uma sequência de expressões separadas por vírgula e comumente delimitada por parênteses:

$$(exp_1, \dots, exp_n)$$

onde $n \geq 0$ e $n \neq 1$, e $exp_1, \dots, exp_n$ são expressões cujos valores são os componentes da tupla. Os parênteses não são obrigatórios, mas são comumente usados por lembrar a notação matemática para tuplas.

O tipo de uma tupla é o **produto cartesiano** dos tipos dos seus componentes. Ele é expresso usando o operador binário de tipos $*$:

$$t_1 * \dots * t_n$$

onde $n \geq 0$ e $n \neq 1$, e $t_1, \dots, t_n$ são os tipos dos respectivos componentes da tupla. Observe que o tamanho de uma tupla (quantidade de componentes) é codificado no seu tipo.

Observe que `()` é a **tupla vazia**, do tipo **unit**.

Não existe tupla de um único componente.

A tabela a seguir mostra alguns exemplos de tuplas:

tupla	tipo
<code>('A', 't')</code>	<code>char * char</code>
<code>('A', 't', 'o')</code>	<code>char * char * char</code>
<code>('A', true)</code>	<code>char * bool</code>
<code>("Joel", 'M', true, "COM")</code>	<code>string * char * bool * string</code>
<code>(true, ("Ana", 'f'), 43)</code>	<code>bool * (string * char) * int</code>
<code>()</code>	<code>unit</code>
<code>("nao eh tupla")</code>	<code>string</code>

Vejamos algumas **operações com tuplas** definidas na biblioteca padrão:

- `fst`: seleciona o primeiro componente de um *par*:

```
# fst ("pedro", 19);;
- : string = "pedro"

# fst (10, 20, 30);;
Error: This expression has type 'a * 'b * 'c
      but an expression was expected of type 'd * 'e
```

- **snd**: seleciona o segundo componente de um *par*:

```
# snd ("pedro", 19);;
- : int = 19
```

8.2 Listas

Lista é uma estrutura de dados formada por uma sequência de valores (elementos) **do mesmo tipo**. Os elementos de uma lista são identificados pela **posição** em que ocorrem na lista.

Em OCaml uma **expressão lista** é formada por uma sequência de expressões separadas por ponto e vírgula e delimitada por colchetes:

$$[\text{exp}_1 ; \dots ; \text{exp}_n]$$

onde $n \geq 0$, e $\text{exp}_1, \dots, \text{exp}_n$ são expressões cujos valores são os elementos da lista.

Um **tipo lista** é formado usando o construtor de tipo **list** precedido pelo tipo dos seus elementos:

$$t \text{ list}$$

onde t é o tipo dos elementos da lista. Observe que o tamanho de uma lista (quantidade de elementos) não é codificado no seu tipo. Normalmente construtores de tipos usam a notação posfixa: primeiro escreve-se os argumentos de tipo, seguidos do construtor de tipo. No exemplo **int list**, **int** é um argumento de tipo e **list** é um construtor de tipo.

A tabela a seguir mostra alguns exemplos de listas:

lista	tipo
<code>['O'; 'B'; 'A']</code>	char list
<code>['B'; 'A'; 'N'; 'A'; 'N'; 'A']</code>	char list
<code>[false; true; true]</code>	bool list
<code>[[false]; []; [true; false; true]]</code>	bool list list
<code>[1.; 8.; 6.; 10.48; -5.]</code>	float list

Estruturalmente uma lista pode ser de duas formas:

- **lista vazia**

- não contém nenhum elemento
- é denotada pelo construtor constante `[]`

- **lista não vazia**

- contém pelo menos um elemento
- é formada por uma **cabeça** (o primeiro elemento da sequência) e por uma **cauda** (uma lista dos demais elementos da sequência)
- é construída pelo construtor `::`, um operador binário infix com associatividade à direita e precedência imediatamente inferior à precedência dos operadores aditivos (+) e (–)
 - * o operando da esquerda é a cabeça da lista
 - * o operando da direita é a cauda da lista

Por exemplo, a lista formada pela sequência dos valores 1, 8, e 6 pode ser escrita como

lista
<code>[1; 8; 6]</code>
<code>1 :: [8; 6]</code>
<code>1 :: (8 :: [6])</code>
<code>1 :: (8 :: (6 :: []))</code>
<code>1 :: 8 :: 6 :: []</code>

Observe que os parênteses neste exemplo são desnecessários, já que o operador `::` associa-se à direita.

Os exemplos anteriores podem ser reescritos com estes construtores de lista:

notação estendida	notação básica
<code>['O'; 'B'; 'A']</code>	<code>'O' :: 'B' :: 'A' :: []</code>
<code>['B'; 'A'; 'N'; 'A'; 'N'; 'A']</code>	<code>'B' :: 'A' :: 'N' :: 'A' :: 'N' :: 'A' :: []</code>
<code>[false; true; true]</code>	<code>false :: true :: true :: []</code>
<code>[[false]; []; [true; false; true]]</code>	<code>(false :: []) :: [] :: (true :: false :: true :: []) :: []</code>
<code>[1.; 8.; 6.; 10.48; -5.]</code>	<code>1. :: 8. :: 6. :: 10.48 :: -5. :: []</code>

Vejamos algumas **operações com listas** definidas na biblioteca padrão:

- **List.hd**: seleciona a **cabeça** (primeiro elemento) de uma lista:

```
# List.hd [1; 2; 3; 4; 5];;  
- : int = 1  
  
# List.hd [];;  
Exception: Failure "hd".  
  
# List.hd [10, 20, 30];;  
- : int * int * int = (10, 20, 30)
```

- **List.tl**: seleciona a **cauda** da lista, ou seja, a lista formada por todos os elementos exceto o primeiro:

```
# List.tl [1; 2; 3; 4; 5];;  
- : int list = [2; 3; 4; 5]  
  
# List.tl [5*4, 6*5];;  
- : (int * int) list = []  
  
# List.tl [5*4; 6*5];;  
- : int list = [30]  
  
# List.tl [8-1];;  
- : int list = []  
  
# List.tl [];;  
Exception: Failure "tl".
```

- **List.length**: calcula o tamanho (quantidade de elementos) de uma lista:

```
# List.length [1; 2; 3; 4; 5];;  
- : int = 5  
  
# List.length [];;  
- : int = 0
```

- **List.nth**: seleciona o i -ésimo elemento de uma lista ($0 \leq i < n$, onde n é o comprimento da lista):

```
# List.nth [10; 20; 30; 40; 50] 2;;  
- : int = 30  
  
# List.nth [10; 20; 30; 40; 50] 0;;  
- : int = 10  
  
# List.nth [10; 20; 30; 40; 50] 10;;  
Exception: Failure "nth".  
  
# List.nth [10; 20; 30; 40; 50] (-4);;  
Exception: Invalid_argument "List.nth".
```

- **List.mem**: verifica se um valor é elemento de uma lista

```
# List.mem 30 [10; 20; 30; 40];;  
- : bool = true  
  
# List.mem 25 [10; 20; 30; 40];;  
- : bool = false
```

- **List.append** ou **(@)**: concatena duas listas.

```
# List.append [1; 2; 3] [40; 50];;
- : int list = [1; 2; 3; 40; 50]

# [1; 2; 3] @ [40; 50];;
- : int list = [1; 2; 3; 40; 50]
```

- **List.rev**: inverte uma lista:

```
# List.rev [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
```

- **List.rev_append**: inverte a primeira lista e concatena o resultado com a segunda lista:

```
# List.rev_append [1; 2; 3] [40; 50];;
- : int list = [3; 2; 1; 40; 50]
```

- **List.combine**: junta duas listas em uma única lista formada pelos pares dos elementos correspondentes:

```
# List.combine ["pedro"; "ana"; "carlos"] [19; 17; 22];;
- : (string * int) list = [("pedro", 19); ("ana", 17); ("carlos", 22)]
```

- **List.split**: separa uma lista de pares em duas listas formadas pelos componentes dos pares:

```
# List.split [("pedro", 19); ("ana", 17); ("carlos", 22)];;
- : string list * int list = (["pedro"; "ana"; "carlos"], [19; 17; 22])
```

8.3 Polimorfismo paramétrico

8.3.1 Operação sobre vários tipos de dados

Algumas funções podem operar sobre vários tipos de dados. Por exemplo: a função **List.hd** recebe uma lista e retorna o primeiro elemento da lista:

```
List.hd ['b'; 'a'; 'n'; 'a'; 'n'; 'a'] ~> 'b'
List.hd ["maria"; "paula"; "peixoto"] ~> "maria"
List.hd [true; false; true; true] ~> true
List.hd [("ana", 2.8); ("pedro", 4.3)] ~> ("ana", 2.8)
```

Não importa qual é o tipo dos elementos da lista. Qual deve ser o tipo de **List.hd**?

```
char list -> char
string list -> string
bool list -> bool
(string * float) list -> string * float
```

Observamos que na verdade **List.hd** pode ter vários tipos.

8.3.2 Variáveis de tipo

Quando um tipo pode ser *qualquer* tipo da linguagem, ele é representado por uma **variável de tipo**.

No exemplo dado, sendo 'a o tipo dos elementos da lista que é passada como argumento para a função **List.hd**, então o tipo de **List.hd** é 'a list -> 'a. 'a é uma *variável de tipo* e pode ser substituída por qualquer tipo. O tipo de **List.hd** estabelece que **List.hd** recebe uma lista com elementos de um tipo qualquer, e retorna um valor deste mesmo tipo.

Em OCaml variáveis de tipo devem começar com um apóstrofo, e são geralmente denominadas 'a, 'b, 'c, etc.

8.3.3 Valor polimórfico

Um valor é chamado **polimórfico** (*de muitas formas*) se o seu tipo contém uma ou mais *variáveis de tipo*.

Por exemplo, o tipo da função `List.hd` pode ser escrito como `'a list -> 'a`. Ou seja, *para qualquer tipo* `'a`, `List.hd` recebe uma lista de valores do tipo `'a` e retorna um valor do tipo `'a`.

Já o tipo da função `List.length`, que recebe uma lista e resulta no tamanho da lista, é `'a list -> int`. Ou seja, *para qualquer tipo* `'a`, `List.length` recebe uma lista de valores do tipo `'a` e retorna um inteiro.

A função `fst` é do tipo `'a * 'b -> 'a`. Ou seja, *para quaisquer tipos* `'a` e `'b`, `fst` recebe um par de valores do tipo `'a * 'b` e retorna um valor do tipo `'a`.

8.3.4 Instanciação de variáveis de tipo

As variáveis de tipo podem ser *instanciadas* para diferentes tipos em diferentes circunstâncias.

Por exemplo, a função `List.length` do tipo `'a list -> int`, pode ser aplicada em diferentes tipos listas, como mostra a tabela:

expressão	valor	instanciação da variável de tipo
<code>List.length [false; true]</code>	2	<code>'a = bool</code>
<code>List.length [5; 4; 3; 2; 1]</code>	5	<code>'a = char</code>
<code>List.length ["ana"; "joel"; "mara"]</code>	3	<code>'a = string</code>
<code>List.length [("ana", true)]</code>	1	<code>'a = string * bool</code>
<code>List.length [(& &), ()]</code>	2	<code>'a = bool -> bool -> bool</code>

8.4 Funções polimórficas predefinidas

Muitas das funções definidas na biblioteca padrão são polimórficas. Algumas delas são mencionadas na tabela seguinte:

função	tipo	descrição
<code>fst</code>	<code>'a * 'b -> 'a</code>	seleciona o primeiro elemento de um par
<code>snd</code>	<code>'a * 'b -> 'b</code>	seleciona o segundo elemento de um par
<code>List.hd</code>	<code>'a list -> 'a</code>	seleciona o primeiro elemento de uma lista
<code>List.tl</code>	<code>'a list -> 'a list</code>	seleciona a cauda de uma lista
<code>List.combine</code>	<code>'a list -> 'b list -> ('a * 'b) list</code>	combina duas listas, elemento a elemento
<code>List.split</code>	<code>('a * 'b) list -> 'a list * 'b list</code>	transforma uma lista de pares em um par de listas

Observe que a lista vazia é polimórfica. O seu tipo é `'a list`.

Tarefa 8.1: Tipo de expressões

Verifique se as seguintes expressões são válidas e determine o seu tipo em caso afirmativo.

- `['a'; 'b'; 'c']`
- `('a', 'b', 'c')`
- `[(false, '0'); (true, '1')]`
- `[(false; true), ['0'; '1']]`
- `[List.tl; List.rev]`
- `[[]]`
- `[[10; 20; 30]; []; [5; 6]; [24]]`
- `(10e-2, 20e-2, 30e-3)`
- `[(2, 3); (4, 5.6); (6, 4.55)]`
- `[(["bom"; "dia"; "brasil"]; fst; "Velho mundo")]`
- `[List.length]`

Tarefa 8.2: Tipo de funções

Determine o tipo de cada uma das funções definidas a seguir, e explique o que elas calculam.

- a) `let second xs = List.hd (List.tl xs)`
- b) `let const x y = x`
- c) `let swap (x, y) = (y, x)`
- d) `let apply f x = f x`
- e) `let flip f x y = f y x`
- f) `let pair x y = (x, y)`
- g) `let palindrome xs = List.rev xs = xs`
- h) `let twice f x = f (f x)`
- i) `let mostra (nome, idade) = "Nome: " ^ nome ^ ", idade: " ^ string_of_int idade`

Tarefa 8.3: Último

Defina uma função chamada `ultimo` que seleciona o último elemento de uma lista não vazia, usando as funções da biblioteca padrão.

Tarefa 8.4: Primeiros

Defina uma função chamada `primeiros` que seleciona todos os elementos de uma lista não vazia, exceto o último., usando as funções da biblioteca padrão.

Tarefa 8.5: Equação do segundo grau

Defina uma função para calcular as raízes reais do polinômio

$$ax^2 + bx + c$$

O resultado da função deve ser a lista das raízes reais.

9 CASAMENTO DE PADRÃO

Resumo

Linguagens funcionais modernas usam casamento de padrão em várias situações, como por exemplo na seleção de componentes de estruturas de dados, na escolha de alternativas em expressões de seleção múltipla, e na aplicação de funções.

Nesta aula vamos aprender as principais formas de padrão e como funciona a operação casamento de padrão. Vamos ainda conhecer algumas construções de OCaml que usam casamento de padrão, como definições de valores e funções e expressões de seleção múltipla.

Sumário

9.1	Casamento de padrão	9-1
9.1.1	Casamento de padrão	9-1
9.1.2	Padrão constante	9-2
9.1.3	Padrão variável	9-2
9.1.4	Padrão curinga	9-2
9.1.5	Padrão tupla	9-2
9.1.6	Padrões lista	9-3
9.1.7	Padrões combinados	9-4
9.1.8	Padrões faixa	9-5
9.1.9	Padrões nomeados	9-5
9.2	Casamento de padrão em definições com let	9-6
9.3	Definição de função usando padrões	9-6
9.4	Expressão de Seleção Múltipla	9-8
9.4.1	Forma e regras de tipo da expressão match	9-8
9.4.2	Avaliação de expressões match	9-8
9.4.3	Exemplos de expressões match	9-9
9.4.4	Expressão match com guardas	9-11
9.5	Exercícios	9-12

9.1 Casamento de padrão

9.1.1 Casamento de padrão

Em ciência da computação, **casamento de padrão**¹ é o ato de verificação da presença de um **padrão** em um dado ou em um conjunto de dados. O padrão é rigidamente especificado. O casamento de padrão é usado para testar se o objeto de estudo possui a estrutura desejada.

Algumas linguagens de programação funcionais como Haskell, ML e Mathematica possuem uma sintaxe especial para expressar padrões e uma construção na linguagem para execução condicional baseada no casamento de padrões.

Em OCaml um **padrão** é uma construção da linguagem de programação que permite *analisar* a estrutura de um valor e *associar variáveis* aos seus componentes.

Casamento de padrão é uma operação envolvendo *um padrão* e *uma expressão* que faz a correspondência (*casamento*) entre o padrão e o valor da expressão. Um casamento de padrão pode *suced*er ou *falhar*, dependendo da forma do padrão e da expressão envolvidos. Quando o casamento de padrão sucede as **variáveis** que ocorrem no padrão são *associadas* aos componentes correspondentes do valor.

Por exemplo o padrão ("ana", peso, _) especifica valores que são triplas onde o primeiro componente é a string "ana", o segundo componente é chamado de peso, e o terceiro componente é irrelevante. O valor ("ana", 58.7, 'F') casa com este padrão e em consequência a variável peso é associada ao valor 58.7. Já o valor ("pedro", 75.3, 'M') não casa com esse padrão.

Em um casamento de padrão, o padrão e a expressão devem ser do *mesmo tipo*.

Existem várias **formas de padrão**. Na sequência algumas delas são apresentadas.

¹Pattern matching em inglês.

9.1.2 Padrão constante

O **padrão constante** é simplesmente uma *constante*. O **casamento** *sucede* se e somente se o padrão for *idêntico* ao valor. Nenhuma associação de **variável** é produzida.

Veja os exemplos seguintes, onde ✓ indica sucesso e ✗ indica falha:

padrão	valor	casamento
10	10	✓
10	28	✗
10	'P'	erro de tipo
'P'	'P'	✓
'P'	'q'	✗
'P'	true	erro de tipo
true	true	✓
true	false	✗
true	65	erro de tipo

9.1.3 Padrão variável

O **padrão variável** é simplesmente um identificador de *variável* de valor (e como tal deve começar com letra minúscula ou sublinhado). O **casamento** *sucede sempre*. A **variável** é associada ao *valor*.

Exemplos:

padrão	valor	casamento
x	10	✓ x ↦ 10
alfa	563.1223	✓ alfa ↦ 563.1223
letra	'K'	✓ letra ↦ 'K'
nomeCliente	"Ana Maria"	✓ nomeCliente ↦ "Ana Maria"
pessoa	("Ana", 'F', 16)	✓ pessoa ↦ ("Ana", 'F', 16)
notas	[5.6; 7.1; 9.0]	✓ notas ↦ [5.6; 7.1; 9.0]

9.1.4 Padrão curinga

O **padrão curinga** é escrito como um sublinhado (). O **casamento** *sucede sempre*. *Nenhuma* associação de **variável** é produzida. é também chamado de **variável anônima**, pois, assim como a variável, casa com qualquer valor, porém não nomeia o valor.

Exemplos:

padrão	valor	casamento
—	10	✓
—	28	✓
—	'P'	✓
—	()	✓
—	(18, 3, 2012)	✓
—	"Ana Maria"	✓
—	[5.6; 7.1; 9.0]	✓

9.1.5 Padrão tupla

Uma **tupla** de padrões também é um padrão:

$$(padrão_1, \dots, padrão_n)$$

O **casamento** *sucede* se e somente se cada um dos padrões casar com o componente correspondente do valor. Se as *aridades* (tamanhos) do padrão tupla e do valor tupla forem *diferentes*, então ocorre um *erro de tipo*.

Exemplos de padrão tupla:

padrão	valor	casamento
(18, true)	(18, true)	✓
(97, true)	(18, true)	×
(18, false)	(18, true)	×
(18, 'M')	(18, true)	erro de tipo
(18, true, 'M')	(18, true)	erro de tipo
()	()	✓
(x, y)	(5, 9)	✓ x ↦ 5 y ↦ 9
(d, _, a)	(5, 9, 2012)	✓ d ↦ 5 a ↦ 2012
(x, y, z)	(5, 9)	erro de tipo
(18, m, a)	(18, 3, 2012)	✓ m ↦ 3 a ↦ 2012
(d, 5, a)	(18, 3, 2012)	×
(nome, sexo, _)	("Ana", 'F', 18)	✓ nome ↦ "Ana" sexo ↦ 'F'
(_, _, idade)	("Ana", 'F', 18)	✓ idade ↦ 18
(_, (_, fam), 9)	('F', ("Ana", "Dias"), 9)	✓ fam ↦ "Dias"
(_, (_, fam), 5)	('F', ("Ana", "Dias"), 9)	×

9.1.6 Padrões lista

Estruturalmente uma lista pode ser vazia ou não vazia:

- padrão lista vazia

[]

- é um padrão constante
- o casamento sucede se e somente se o valor for a lista vazia

- padrão lista não vazia

$pad_1 :: pad_2$

- é formado por dois padrões pad_1 e pad_2
- o casamento sucede se e somente se o valor for uma lista não vazia cuja cabeça casa com pad_1 e cuja cauda casa com pad_2
- $::$ é o construtor de lista não vazia, um operador binário infix associativo à direita com precedência logo abaixo dos operadores aritméticos aditivos (+) e (-).

Exemplos de padrões lista:

- O padrão [] casa somente com a lista vazia.

padrão	valor	casamento
[]	[]	✓
[]	[1; 2; 3]	×

- O padrão $x::xs$ casa com qualquer lista não vazia, associando as variáveis x e xs com a cabeça e com a cauda da lista, respectivamente.

padrão	valor	casamento
$x::xs$	[]	×
$x::xs$	[1; 2; 3; 4]	✓ x ↦ 1 xs ↦ [2; 3; 4]
$x::xs$	['A']	✓ x ↦ 'A' xs ↦ []

- O padrão $x::y::_$ casa com qualquer lista que tenha pelo menos dois elementos, associando as variáveis x e y ao primeiro e segundo elementos da lista, respectivamente.

padrão	valor	casamento
$x::y::_$	$[]$	×
$x::y::_$	$["ana"]$	×
$x::y::_$	$[1; 2]$	✓ $x \mapsto 1$ $y \mapsto 2$
$x::y::_$	$[1; 2; 3; 4]$	✓ $x \mapsto 1$ $y \mapsto 2$

- O padrão $x::_:z::[]$ casa com qualquer lista que tenha exatamente três elementos, associando as variáveis x e z ao primeiro e ao terceiro elementos da lista, respectivamente.

padrão	valor	casamento
$x::_:z::[]$	$[]$	×
$x::_:z::[]$	$["ana"]$	×
$x::_:z::[]$	$[1; 2; 3]$	✓ $x \mapsto 1$ $z \mapsto 3$
$x::_:z::[]$	$[1; 2; 3; 4; 5]$	×

- O padrão $0::a::_$ casa com qualquer lista de números inteiros que tenha pelo menos dois elementos, sendo o primeiro igual a zero, associando a variável a ao segundo elemento da lista.

padrão	valor	casamento
$0::a::_$	$[]$	×
$0::a::_$	$[0]$	×
$0::a::_$	$[0; 2; 3]$	✓ $a \mapsto 2$
$0::a::_$	$[0; 10; 6; 3]$	✓ $a \mapsto 10$
$0::a::_$	$[7; 0; 8]$	×

- O padrão $(m,_)::_$ casa com qualquer lista não vazia de pares, associando a variável m ao primeiro componente do primeiro elemento da lista.

padrão	valor	casamento
$(m,_)::_$	$[]$	×
$(m,_)::_$	$[("fim", true)]$	✓ $m \mapsto "fim"$
$(m,_)::_$	$[(10, 'M'); (20, 'F')]$	✓ $m \mapsto 10$

Padrão lista na notação especial

O padrão

$[\text{padrão}_1 ; \dots ; \text{padrão}_n]$

é uma *abreviação sintática* para

$\text{padrão}_1 :: \dots :: \text{padrão}_n :: []$

Observe que o casamento sucede somente se o valor for uma lista com exatamente n elementos.

Exemplos: o padrão $[1; \text{alfa}]$ casa com qualquer lista de dois números que começa com 1, associando a variável alfa ao segundo elemento da lista.

padrão	valor	casamento
$[1; \text{alfa}]$	$[]$	×
$[1; \text{alfa}]$	$[1]$	×
$[1; \text{alfa}]$	$[1; 5]$	✓ $\text{alfa} \mapsto 5$
$[1; \text{alfa}]$	$[9; 5]$	×
$[1; \text{alfa}]$	$[1; 2; 3]$	×

9.1.7 Padrões combinados

Um padrão combinado é escrito como

$\text{pad}_1 \mid \dots \mid \text{pad}_n$

onde $pad_1, \dots, pad_n$ são padrões. O casamento de um padrão combinado *sucede* se e somente se o casamento de um dos padrões pad_i sucede.

Qualquer variável que seja usada em um padrão combinado deve ocorrer em todos os padrões da combinação. Exemplos de padrões combinados:

- O padrão $'a' | 'e' | 'i' | 'o' | 'u'$ casa com qualquer caracter que seja uma das vogais a, e, i, o ou u :

padrão	valor	casamento
$'a' 'e' 'i' 'o' 'u'$	$'a'$	✓
$'a' 'e' 'i' 'o' 'u'$	$'e'$	✓
$'a' 'e' 'i' 'o' 'u'$	$'H'$	×

- O padrão $[x] | [x; _] | [x; _; _]$ casa com qualquer lista de até três elementos e seleciona o primeiro elemento da lista:

padrão	valor	casamento
$[x] [x; _] [x; _; _]$	$[]$	×
$[x] [x; _] [x; _; _]$	$[10]$	✓ $x \mapsto 10$
$[x] [x; _] [x; _; _]$	$[10; 20]$	✓ $x \mapsto 10$
$[x] [x; _] [x; _; _]$	$[10; 20; 30]$	✓ $x \mapsto 10$
$[x] [x; _] [x; _; _]$	$[10; 20; 30; 40]$	×

- O padrão $(0, k) | (k, 1)$ casa com pares onde o primeiro componente é 0, selecionando o segundo componente, ou onde o segundo elemento é 1, selecionando o primeiro componente:

padrão	valor	casamento
$(0, k) (k, 1)$	$(0, 3)$	✓ $k \mapsto 3$
$(0, k) (k, 1)$	$(7, 1)$	✓ $k \mapsto 7$
$(0, k) (k, 1)$	$(0, 1)$	✓ $k \mapsto 1$
$(0, k) (k, 1)$	$(4, 5)$	×

9.1.8 Padrões faixa

O padrão

$$c_1 \dots c_2$$

onde c_1 e c_2 são literais caracter, casa com qualquer caracter na faixa de valores entre c_1 e c_2 , inclusive.

Exemplos:

padrão	valor	casamento
$'a' \dots 'z'$	$'a'$	✓
$'a' \dots 'z'$	$'h'$	✓
$'a' \dots 'z'$	$'M'$	×
$'a' \dots 'z'$	$'\#'$	×
$'0' \dots '9'$	$'8'$	✓
$'0' \dots '9'$	$'h'$	×

9.1.9 Padrões nomeados

O padrão

$$padrão_1 \text{ as } nome$$

onde $padrão_1$ é um padrão e $nome$ é um identificador de valor, casa com os mesmos valores que $padrão_1$. Se o casamento do padrão $padrão_1$ sucede, o identificador $nome$ é associado ao valor casado, além das associações de variáveis realizadas pelo casamento do padrão $padrão_1$.

Exemplos:

padrão	valor	casamento
<code>(_, 0) as par</code>	<code>('M', 0)</code>	✓ $\text{par} \mapsto ('M', 0)$
<code>(_, 0) as par</code>	<code>(5, 1)</code>	✗
<code>_::_ as lista</code>	<code>[2; 4; 6]</code>	✓ $\text{lista} \mapsto [2; 4; 6]$
<code>_::_ as lista</code>	<code>[]</code>	✗
<code>_::k::_ as xs</code>	<code>[2; 4; 6]</code>	✓ $\text{xs} \mapsto [2; 4; 6]$ $k \mapsto 4$
<code>(("ana", _) as nome, _, id)</code>	<code>(("ana", "pera"), 'f', 19)</code>	✓ $\text{nome} \mapsto ("ana", "pera")$ $\text{id} \mapsto 19$

9.2 Casamento de padrão em definições com let

Definições de variáveis com `let` podem ocorrer em um módulo (globais ao módulo), ou em uma expressão `let` (locais a uma expressão).

O lado esquerdo da definição pode ser um padrão e o lado direito deve ser uma expressão. O valor da expressão e o padrão devem casar. Caso o casamento de padrão falhe uma exceção `Match_failure` é lançada.

Por exemplo:

```
# let (prefixo, sufixo) = List.split [(10,"ana"); (20,"pedro"); (30,"carlos")];;
val prefixo : int list = [10; 20; 30]
val sufixo : string list = ["ana"; "pedro"; "carlos"]

# let tupla = 1.2, 3.4, 0.75, 0.11;;
val tupla : float * float * float * float = (1.2, 3.4, 0.75, 0.11)

# let x, y, z, w = tupla;;
val x : float = 1.2
val y : float = 3.4
val z : float = 0.75
val w : float = 0.11
```

Portanto o lado esquerdo na definição com `let` pode ser um padrão. Em geral é recomendável usar no `let` apenas **padrões irrefutáveis**, ou seja, padrões cujo casamento não podem falhar, para evitar a ocorrência de uma exceção caso o casamento falhe. Veja os exemplos:

```
# let _ :: segundo :: _ = [10; 20; 30; 40; 50];;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val segundo : int = 20

# let _ :: segundo :: _ = [10];;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Exception: Match_failure ("//toplevel//", 1, 4).
```

9.3 Definição de função usando padrões

Em uma **definição de função** em OCaml os **parâmetros** que representam os argumentos são *padrões*. Isto significa que eles não precisam ser necessariamente variáveis, como na maioria das linguagens convencionais, mas podem ser padrões de qualquer uma das formas existentes na linguagem.

Quando a função é aplicada a seus argumentos, é feito o casamento de padrão entre os argumentos e os respectivos parâmetros da função. Se algum dos casamentos de padrão *falhar* ocorre uma *exceção* que, se não for tratada no programa, culmina em um *erro de execução*.

Geralmente o uso de padrões para especificar os argumentos torna a definição da função mais *clara* e *concisa*.

Exemplo: a função `maximo` resulta no maior componente de um par de inteiros:

```

let maximo par =
  let x = fst par in
  let y = snd par in
  if x > y then
    x
  else
    y

```

Nesta definição par é um padrão variável e casa com qualquer argumento.

```

maximo (20, 30)  ~> 30
maximo (2+3, 2-3) ~> 5

```

Alternativamente a função maximo pode ser definida usando um padrão mais específico para pares que permita acessar diretamente os seus componentes:

```

let maximo (x, y) =
  if x > y then
    x
  else
    y

```

Nesta versão o parâmetro formal é o padrão (x, y) que casa com qualquer par associando x ao primeiro componente, e y ao segundo componente do par.

Outros exemplos:

```

let fst (x, _) = x

```

```

let snd (_, y) = y

```

```

fst (1+2, 1-2) ~> 3
snd (5/2, 5>2) ~> true

```

Tarefa 9.1: Distância entre dois pontos

A distância entre dois pontos (x_1, y_1, z_1) e (x_2, y_2, z_2) no espaço é dada por

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Defina uma função que recebe dois pontos no espaço e retorna a distância entre eles. Considere que um ponto no espaço é representado por uma tripla de números que são as coordenadas do ponto. Use casamento de padrão.

Tarefa 9.2

Estude a seguinte definição e apresente uma definição alternativa mais simples desta função, usando padrões.

```

let opp z =
  if fst z = 1 then
    fst (snd z) + snd (snd z)
  else
    if fst z = 2 then
      fst (snd z) - snd (snd z)
    else
      0

```

Tarefa 9.3: Comparação de frações

Considere uma fração é representada por um par de números inteiros. Defina uma função `compara_fracao` que recebe duas frações como argumentos e retorna -1 , 0 , ou 1 caso a primeira seja menor, igual, ou maior do que a segunda, respectivamente.

9.4 Expressão de Seleção Múltipla

A **expressão match** é uma forma de expressão que permite *selecionar* um entre vários resultados alternativos baseando-se em casamento de padrões. Uma expressão **match** é formada por:

- uma *expressão de controle*, cujo valor é usado para escolher uma das alternativas
- uma sequência de *alternativas*, onde cada alternativa é formada por:
 - um *padrão*, usado para decidir se a alternativa será escolhida
 - uma *expressão*, usada para dar o resultado caso a alternativa seja escolhida

Exemplo:

```
match calculo x (y / 2) with
| 0 -> x*x + 5*x + 6;
| 1 -> 4*y - 8;
| n -> (x*x + y*y) / n
```

9.4.1 Forma e regras de tipo da expressão match

Uma expressão **match** é da forma:

```
match exp with
| padrão1 -> res1
| :
| padrãon -> resn
```

onde:

- a *expressão de controle* é exp
- os *resultados alternativos* são dados pelas expressões $res_1, \dots, res_n$, selecionados pelos respectivos *padrões* $padrão_1, \dots, padrão_n$
- a barra vertical $|$ que precede o primeiro padrão é opcional.

Regras de tipo:

- a expressão de controle exp e os padrões $padrão_1, \dots, padrão_n$ devem ser todos de um *mesmo tipo*
- os resultados $res_1, \dots, res_n$ devem ser todos de um *mesmo tipo*, e este será o *tipo* da expressão **match** (ou seja, o tipo do resultado final)

9.4.2 Avaliação de expressões match

- É feito o *casamento de padrão* do valor de exp com os padrões, na sequência em que foram escritos, até que se obtenha sucesso ou se esgotem os padrões
- O *primeiro* padrão cujo casamento suceder é escolhido
- O *resultado* final da expressão **match** é dado pela expressão associada ao padrão escolhido
- O resultado é avaliado em um *ambiente estendido* com as associações de variáveis construídas no casamento de padrão
- Se a expressão *não casar* com nenhum dos padrões, a avaliação da expressão **match** falha é a *exceção* **Match_failure** é lançada.

9.4.3 Exemplos de expressões match

A expressão

```
match 3 - 2 + 1 with
| 0 -> "zero"
| 1 -> "um"
| 2 -> "dois"
| 3 -> "tres"
```

resulta em **"dois"**, pois o valor da expressão `3-2+1` é `2`, que casa com o terceiro padrão `2`, selecionando **"dois"** como resultado.

A expressão

```
match 23 > 10 with
| true  -> "beleza!"
| false -> "oops!"
```

resulta em **"beleza!"**, pois o valor da expressão `23 > 10` é `true`, que casa com o primeiro padrão `true`, selecionando **"beleza!"** como resultado.

A expressão

```
match Char.uppercase (String.get "masculino" 0) with
| 'F' -> 10.2
| 'M' -> 20.0
```

resulta em `20.0`, pois o valor da expressão `Char.uppercase (String.get "masculino" 0)` é `'M'`, que casa com o segundo padrão `'M'`, selecionando `20.0` como resultado.

A avaliação da expressão

```
match String.get "masculino" 0 with
| 'F' -> 10.2
| 'M' -> 20.0
```

falha, pois o valor da expressão `String.get "masculino" 0` não casa com nenhum dos padrões, fazendo com que uma exceção `Match_failure` seja lançada.

A expressão

```
match Char.uppercase (String.get "masculino" 0) with
| 'F' -> "mulher"
| 'M' -> 20.0
```

está incorreta, pois os resultados **"mulher"** e `20.0` não são do mesmo tipo.

A expressão

```
match String.get "Masculino" 0 = 'F' with
| true  -> "mulher"
| 1      -> "homem"
```

está incorreta, pois os padrões `true` e `1` não são do mesmo tipo.

A expressão

```
match String.get "Masculino" 0 with
| true  -> "mulher"
| false -> "homem"
```

está incorreta, pois a expressão `String.get "Masculino" 0` e os padrões `true` e `false` não são do mesmo tipo.

A expressão

```
match 3 - 2 + 1 with
| x -> 11 * x
```

resulta em 22, pois o valor da expressão $3 - 2 + 1$ é 2, que casa com o primeiro padrão x , associando a variável x ao valor 2, e selecionando $11 * x$ como resultado

A expressão

```
match 256 mod 10 with
| 7 -> 0
| n -> n * 1000
```

resulta em 6000, pois o valor da expressão $256 \bmod 10$ é 6, que casa com o segundo padrão n , associando a variável n ao valor 6, e selecionando $n * 1000$ como resultado

A expressão

```
match 257 mod 10 with
| 7 -> 0
| n -> n * 1000
```

resulta em 0, pois 7 é o primeiro padrão que casa com o valor da expressão $257 \bmod 10$.

Já a expressão

```
match 257 mod 10 with
| n -> n * 1000
| 7 -> 0
```

resulta em 7000, pois n é o primeiro padrão que casa com o valor da expressão $257 \bmod 10$.

Observe que quando os padrões não são mutuamente exclusivos a **ordem** em que as alternativas são escritas pode influenciar o resultado.

A expressão

```
match 46 - 2*20 with
| 0 -> "zero"
| 1 -> "um"
| 2 -> "dois"
| 3 -> "tres"
| 4 -> "quatro"
| _ -> "maior que quatro"
```

resulta em "maior que quatro", pois $_$ é o primeiro padrão que casa com o valor da expressão $46 - 2*20$.

A expressão

```
match (3+2, 3-2) with
| (0, 0) -> 10
| (_, 1) -> 20
| (x, 2) -> x*x
| (x, y) -> x*y - 1
```

resulta em 20, pois $(_, 1)$ é o primeiro padrão que casa com o valor da expressão $(3+2, 3-2)$.

A expressão

```
match List.tl [10] with
| [] -> "vazia"
| _ -> "nao vazia"
```

resulta em "vazia", pois o valor da expressão `List.tl [10]` casa com o padrão para lista vazia `[]`.

A expressão

```
match [10; 20; 30; 40] with
| [] -> "lista vazia"
| x::_ -> "cabeca: " ^ string_of_int x
```

resulta em **"cabeca: 10"**, pois a lista [10; 20; 30; 40] casa com o padrão para lista não vazia $x::_$, associando x a 10.

A expressão

```
match [10; 11; 12; 13; 14; 15] with
| x::y::z::_ -> x + y + z
| _          -> 0
```

resulta em 33, pois a lista [10; 11; 12; 13; 14; 15] casa com o padrão $x::y::z::_$, associando x a 10, y com 11 e z com 12.

A expressão

```
match [10; 20] with
| x::y::z::_ -> x + y + z
| _          -> 0
```

resulta em 0, pois a lista [10; 20] não casa com o primeiro padrão $x::y::z::_$, mas casa com o segundo $_$. Observe que o primeiro padrão casa somente com listas que tenham *pelo menos três elementos*.

A expressão

```
match [10; 20; 30] with
| [x1; _, x3] -> x1 + x3
| _           -> 0
```

resulta em 40, pois a lista [10; 20; 30] casa com o primeiro padrão $[x1; _, x3]$. Observe que este padrão casa somente com listas que tenham *exatamente três elementos*.

9.4.4 Expressão match com guardas

Em uma expressão match cada padrão pode ser acompanhado de uma expressão condicional chamada **guarda**, introduzida pela palavra reservada **when**. Neste caso a alternativa somente é escolhida quando o casamento de padrão sucede e a expressão é verdadeira.

A expressão

```
match [100; 20; 3] with
| a::b::xs when a > b -> b::a::xs
| ys          -> ys
```

resulta em [20; 100; 3], pois a lista [100; 20; 3] casa com o primeiro padrão $a::b::xs$ e o primeiro elemento é maior do que o segundo.

No exemplo a seguir, a expressão

```
match ("Paulo Roberto", 'M', 28, 69.3) with
| (_, _, idade, peso) when idade < 18 -> 2. *. peso
| (_, _, idade, peso) when idade < 21 -> 3. *. peso
| (_, 'F', _, peso)                    -> peso
| (_, 'M', idade, peso) when idade < 40 -> peso +. 10.
| (_, _, _, peso)                      -> 0.9 *. peso
```

resulta em 79.3, pois a tupla ("Paulo Roberto", 'M', 28, 69.3)

- casa com o primeiro padrão, porém a guarda não é satisfeita
- casa com o segundo padrão, porém a guarda não é satisfeita
- não casa com o terceiro padrão
- casa com o quarto padrão, e a guarda é satisfeita, logo o resultado é dado por $\text{peso} +. 10.$

9.5 Exercícios

Tarefa 9.4: Avaliando expressões

Quais das seguintes expressões são legais em OCaml? Para aquelas que são legais, qual é o tipo e o valor da expressão?

1.

```
match 1 with
  1 -> 2
| _ -> 3
```

2.

```
match 2 with
  1 + 1 -> 2
| _ -> 3
```

3.

```
let _ as s = "abc" in s ^ "def"
```

4.

```
let h (1 | 2) as i = i + 1 in
h 2
```

Tarefa 9.5: Decomposição de tuplas

Um dos problemas com tuplas é que não há uma função geral para selecionar um de seus componentes. Suponha que tenhamos que tentar definir tal função para triplas.

```
let nth i (x, y, z) =
  match i with
  | 1 -> x
  | 2 -> y
  | 3 -> z
  | _ -> raise (Invalid_argument "nth")
```

1. Qual é o tipo da função `nth`?
2. Há alguma maneira de reescrever esta função de forma que ela permita que os elementos da tupla sejam de tipos diferentes?

Nos exercícios seguintes use casamento de padrão para definir as funções solicitadas.

Tarefa 9.6: Implicação lógica

A função `implica` a seguir, do tipo `bool -> bool -> bool`, implementa a operação de implicação lógica de acordo com a tabela verdade que aprendemos em Lógica.

```
let implica a b =
  match (a, b) with
  (true, true) -> true
| (true, false) -> false
| (false, true) -> true
| (false, false) -> true
```

Faça duas definições mais compactas de `implica` onde são usadas apenas duas alternativas no casamento de padrão.

Tarefa 9.7: Vogais

Defina uma função `e_vogal` do tipo `char -> bool` que verifica se um caracter é uma vogal. Use casamento de padrão. Não use expressões condicionais.

Tarefa 9.8: Cabeça e cauda de uma lista não vazia

Defina as funções `hd` e `tl` dos tipos `'a list -> 'a` e `'a list -> 'a list` respectivamente, que selecionam a cabeça e a cauda respectivamente de uma lista não vazia.

Exemplos:

```
hd ["ivo"; "ana"; "maria"; "inês"] ~> "ivo"
tl ["ivo"; "ana"; "maria"; "inês"] ~> ["ana"; "maria"; "inês"]
hd [1] ~> 1
tl [1] ~> []
hd [] ~> exceção
tl [] ~> exceção
```

Observação: já existem as funções `List.hd` e `List.tl` na biblioteca padrão similares a estas funções.

Tarefa 9.9: Tamanho de uma lista

Defina a função `length` do tipo `'a list -> int` que recebe uma lista `l` e resulta no número de elementos em `l`.

Exemplos:

```
length [] ~> 0
length [10; 20; 30; 40; 50] ~> 5
```

Faça duas definições, uma sem recursividade de cauda e outra com recursividade de cauda.

Observação: já existe a função `List.length` na biblioteca padrão similar a esta função.

Tarefa 9.10: Pertinência a uma lista

Defina a função `mem` do tipo `'a -> 'a list -> bool` que recebe um valor e uma lista e verificam se o valor é um elemento da lista.

Exemplos:

```
mem "ana" ["ivo"; "ana"; "maria"; "inês"] ~> true
mem 5 [1; 8; 10; 6; 48] ~> false
mem 4.5 [] ~> false
```

Observação: já existe a função `List.mem` na biblioteca padrão similar a esta função.

Tarefa 9.11: Todos positivos?

Defina uma função `all_positive` que recebe uma lista de inteiros e verifica se todos os elementos da lista são positivos.

```
val all_positive :: int list -> bool
```

Exemplos:

```
all_positive [1; 8; 10; 6; 48] ~> true
all_positive [1; 8; -10; -6; 48] ~> false
all_positive [] ~> true
```

Tarefa 9.12: Seleção de um prefixo de uma lista

Defina a função `prefixo` do tipo `int -> 'a list -> 'a list` que recebe um número inteiro n e uma lista l e resulta na lista dos n primeiros elementos de l .

Exemplos:

```
prefixo 0 [10; 20; 30; 40; 50] ~> []  
prefixo 2 [10; 20; 30; 40; 50] ~> [10; 20]  
prefixo 9 [10; 20; 30; 40; 50] ~> [10; 20; 30; 40; 50]
```

Tarefa 9.13: Pesquisa em banco de dados

Suponha que você esteja implementando um banco de dados relacional sobre funcionários de uma empresa onde o banco de dados é uma lista de tuplas formadas pelo nome, pela idade e pelo salário dos funcionários.

```
let db =  
  [ "Pedro",    23, 50.1;  
    "Paulo",    40, 107.3;  
    "Percival", 44, 100.0;  
    "Maria",    32, 12.7;  
    "Bruno",    29, 107.3;  
    "Raul",     25, 88.5;  
    "Manoel",   30, 107.3;  
    "Vilma",    22, 71.0;  
  ]
```

- Escreva uma função `find_salary : string -> (string * int * float) list -> float` que retorna o salário de um funcionário, dado o seu nome e o banco de dados.
- Escreva uma função para calcular a idade média dos funcionários.
- Escreva uma função para determinar a porcentagem de funcionários que tem idade superior à idade média de todos os funcionários.
- Escreva uma função para determinar o nome e o salário dos funcionários com idade inferior a uma dada idade.
- Escreva uma função para determinar qual é o maior salário e os nomes dos funcionários que o recebem.

Tarefa 9.14: Concatenação de listas

A função `append : 'a list -> 'a list -> 'a list` concatena duas listas. Ela pode ser definida como

```
let rec append l1 l2 =  
  match l1 with  
  | [] -> l2  
  | h :: t -> h :: append t l2
```

Escreva uma versão de `append` que apresente recursividade de cauda.

Tarefa 9.15: Invertendo uma lista

Escreva uma função `rev` para inverter os elementos de uma dada lista.

Exemplos:

```
rev [] ~> []
rev [10; 1; 8; 6; -5] ~> [-5; 6; 8; 1; 10]
rev ['a'; 'e'; 'o'] ~> ['o'; 'e'; 'a']
```

Observação: já existe a função `List.rev` na biblioteca padrão similar a esta função.

Tarefa 9.16: Soma dos elementos de uma lista

Escreva uma função `sum` para somar todos os elementos de uma dada lista de números inteiros.

Exemplos:

```
sum [] ~> 0
sum [10; 1; 8; 6; -5] ~> 20
```

Tarefa 9.17: Soma cumulativa

Defina uma função que recebe uma lista de números inteiros e resulta na lista da soma cumulativa de seus elementos, isto é, uma nova lista onde o i -ésimo elemento é a soma dos primeiros i elementos da lista original. Por exemplo, a soma cumulativa de `[1; 2; 10; 8; 4]` é `[1; 3; 13; 21; 25]`.

Tarefa 9.18: Lista ordenada?

Escreva uma função `is_sorted` que verifica se uma dada lista está ordenada em ordem crescente.

Exemplos:

```
is_sorted [] ~> true
is_sorted [10; 1; 8; 6; -5] ~> false
is_sorted [-5; 1; 6; 8; 10] ~> true
```

Tarefa 9.19: Convertendo entre strings e listas de caracteres

Escreva uma função que converte uma lista de caracteres em string e outra que converte uma string em uma lista de caracteres.

Tarefa 9.20: Remove elementos da uma lista

Defina a função `strip` : 'a list -> 'a list -> 'a list que, dadas duas listas, retira da segunda todos os elementos que ocorrem na primeira.

Exemplos:

```
strip [1; 5] [1; 2; 5; 5; 3; 5; 1] ~> [2; 3]
strip ['a'] ['b'; 'a'; 't'; 'a'; 't'; 'a'] ~> ['b'; 't'; 't']
```

Tarefa 9.21: Marcadores em listas

Uma lista pode ser dividida em sublistas, com base em elementos especiais chamados *marcadores* que dizem quando começa uma lista e quando acaba outra.

1. Defina a função `splitToken` : 'a -> 'a **list** -> 'a **list list** que recebe um valor e uma lista e retorna uma lista de listas utilizando o valor dado como marcador.

Exemplos:

```
splitToken 2 [1; 2; 3; 4; 2; 5; 67; 8; 9; 0; 3]
  ~> [[1]; [3; 4]; [5; 67; 8; 9; 0; 3]]
splitToken 3 [1; 2; 3; 4; 2; 5; 67; 8; 9; 0; 3]
  ~> [[1; 2]; [4; 2; 5; 67; 8; 9; 0]; []]
splitToken 0 [0; 1; 2; 0; 3; 4; 0; 0]
  ~> [[]; [1; 2]; [3; 4]; []; []]
splitToken ' ' ['O'; 'C'; 'a'; 'm'; 'l'; ' '; 'i'; 's'; ' '; 'c'; 'o'; 'o'; 'l']
  ~> [['O'; 'C'; 'a'; 'm'; 'l']; ['i'; 's']; ['c'; 'o'; 'o'; 'l']]
splitToken ' ' ['c'; 'o'; 'o'; 'l']
  ~> [['c'; 'o'; 'o'; 'l']]
```

2. Defina a função `joinToken` : 'a -> 'a **list list** -> 'a **list** que recebe um valor e uma lista de listas e retorna a concatenação das sublistas usando o primeiro parâmetro como separador.

Exemplo:

```
joinToken ' ' [['t'; 'h'; 'i'; 's']; ['i'; 's']; ['a']; ['t'; 'e'; 's'; 't']]
  ~> ['t'; 'h'; 'i'; 's'; ' '; 'i'; 's'; ' '; 'a'; ' '; 't'; 'e'; 's'; 't']
```

Tarefa 9.22: Agrupamento

Defina a função `group` :: 'a **list** -> 'a **list list** que agrupa elementos semelhantes em sublistas.

Exemplo:

```
group [2; 3; 3; 2; 1; 4; 4; 4] ~> [[2]; [3; 3]; [2]; [1]; [4; 4; 4]]
```

10 EXPRESSÕES LAMBDA

Resumo

Expressões lambdas são funções anônimas que podem ser usadas sem restrições, como qualquer outro valor de primeira classe da linguagem.

Nesta aula vamos aprender sobre expressões lambda.

Sumário

10.1 Valores de primeira classe	10-1
10.1.1 Valores de primeira classe	10-1
10.1.2 Valores de primeira classe: Literais	10-1
10.1.3 Valores de primeira classe: Variáveis	10-2
10.1.4 Valores de primeira classe: Argumentos	10-2
10.1.5 Valores de primeira classe: Resultado	10-2
10.1.6 Valores de primeira classe: Componentes	10-3
10.2 Expressão lambda	10-3
10.2.1 Expressões lambda with fun	10-3
10.2.2 Exemplos de expressões lambda	10-3
10.2.3 Uso de expressões lambda	10-4
10.2.4 Exercícios	10-4
10.2.5 Expressões lambda with function	10-5
10.3 Aplicação parcial de funções	10-6
10.3.1 Aplicação parcial de funções	10-6
10.3.2 Aplicação parcial de funções: exemplos	10-6
10.4 Currying	10-7
10.4.1 Funções <i>curried</i>	10-7
10.4.2 Por que <i>currying</i> é útil?	10-8
10.4.3 Convenções sobre <i>currying</i>	10-8
10.5 Utilidade de expressões lambda	10-8
10.5.1 Exercícios	10-10

10.1 Valores de primeira classe

10.1.1 Valores de primeira classe

- Tipo de **primeira classe**: não há restrições sobre como os seus valores podem ser usados.
- São valores de primeira classe:
 - números
 - caracteres
 - tuplas
 - listas
 - *funções*

entre outros

10.1.2 Valores de primeira classe: Literais

- Valores de vários tipos podem ser escritos *literalmente*, sem a necessidade de se dar um nome a eles:

valor	tipo	descrição
<code>true</code>	<code>bool</code>	o valor lógico <i>verdadeiro</i>
<code>'G'</code>	<code>char</code>	o caracter <i>G</i>
<code>456</code>	<code>int</code>	o número 456
<code>2.45</code>	<code>float</code>	o número em ponto flutuante 2.45
<code>"ocaml"</code>	<code>String</code>	a cadeia de caracteres <i>ocaml</i>
<code>[1; 6; 4; 5]</code>	<code>int list</code>	a lista dos inteiros 1, 6, 4, 5
<code>("Ana", false)</code>	<code>string * bool</code>	o par formado por <i>Ana</i> e <i>falso</i>

- Funções também podem ser escritas sem a necessidade de receber um nome:

valor	tipo	descrição
<code>fun x -> 3*x</code>	<code>int -> int</code>	função que calcula o triplo
<code>fun n -> n mod 2 = 0</code>	<code>int -> bool</code>	função que verifica se é par
<code>fun (p, q) -> p+q</code>	<code>int * int -> int</code>	função que soma par

10.1.3 Valores de primeira classe: Variáveis

- Valores de vários tipos podem ser *nomeados*:

```
let matricula = 456
let sexo      = 'M'
let aluno     = ("Jane Eyre", 101408, 'M', "COM")
let disciplinas = ["BCC222"; "BCC221"; "MTM153"; "PRO300"]
let livroTexto = ("Programming in OCaml", "G. Hutton", 2007)
```

- Funções também podem ser nomeadas:

```
let triplo = fun x -> 3 * x
```

Esta equação define a variável `triplo`, associando-a a um valor que é uma função.

OCaml permite escrever esta definição de forma mais sucinta:

```
let triplo x = 3 * x
```

10.1.4 Valores de primeira classe: Argumentos

- Valores de vários tipos podem ser *argumentos* de funções:

```
sqrt 2.45
not true
List.length [1; 6; 4; 5]
```

- Funções também podem ser argumentos de outras funções:

```
List.map triplo [1; 2; 3] ~> [3; 6; 9]
```

A função `triplo` é aplicada a cada elemento da lista `[1; 2; 3]`, resultando na lista `[3; 6; 9]`

10.1.5 Valores de primeira classe: Resultado

- Valores de vários tipos podem ser *resultados* de funções:

```
not false ~> true
List.length [1; 6; 4; 5] ~> 4
snd ("Ana", 'F') ~> 'F'
List.tl [1; 6; 4; 5] ~> [6; 4; 5]
```

- Funções também podem ser resultados de outras funções. Considere a operação de composição de funções definida como

```
let (%) f g =
  fun x -> f (g x)
```

Assim:

```
(abs_float % sin) (3.0 *. 3.1415 /. 2.0) ~> 0.999999990342226308
(sqrt % abs_float) (-9.0) ~> 3.0
```

O operador binário infix `(%)` faz a composição de duas funções.

10.1.6 Valores de primeira classe: Componentes

- Valores de vários tipos podem ser *componentes* de outros valores:

```
("Ana", 'F', 18)
["BCC222"; "BCC221"; "MTM153"; "PRO300"]
[("Ailton", 101408); ("Lidiane", 102408)]
```

- Funções também podem ser componentes de outros valores:

```
List.map (fun g -> g (-3.1415)) [abs_float; sin; cos]
~> [3.1415; -9.26535896604902578e-05; -0.999999995707656186]
```

O segundo argumento de `List.map` é a lista das funções `abs_float`, `sin` e `cos`.

10.2 Expressão lambda

10.2.1 Expressões lambda with `fun`

- Da mesma maneira que um número inteiro, uma string ou um par podem ser escritos sem ser nomeados, uma função também pode ser escrita sem associá-la a um nome.
- **Expressão lambda** é uma função anônima (sem nome), formada por uma sequência de padrões representando os argumentos da função, e um corpo que especifica como o resultado pode ser calculado usando os argumentos:

```
fun padrão1 ... padrãon -> expressao
```

- O termo *lambda* provém do cálculo lambda (teoria de funções na qual as linguagens funcionais se baseiam), introduzido por Alonzo Church nos anos 1930 como parte de uma investigação sobre os fundamentos da Matemática.
- No cálculo lambda expressões lambda são introduzidas usando a letra grega λ . Em OCaml usa-se a palavra reservada `fun`.

10.2.2 Exemplos de expressões lambda

Função anônima que calcula o dobro de um número:

```
fun x -> x + x
```

O tipo desta expressão lambda é `int -> int`.

Função anônima que mapeia um número x a $2x + 1$:

```
fun x -> 2*x + 1
```

cujos tipos são `int -> int`.

Função anônima que recebe três argumentos e calcula a sua soma:

```
fun a b c -> a + b + c
```

cujos tipos são `int -> int -> int -> int`.

Definições de função usando expressão lambda:

```
let f          = fun x -> 2*x + 1
let somaPar    = fun (x, y) -> x + y
```

é o mesmo que

```
let f x        = 2*x + 1
let somaPar (x, y) = x + y
```

10.2.3 Uso de expressões lambda

- Apesar de não terem um nome, funções construídas usando *expressões lambda* podem ser usadas da mesma maneira que outras funções.
- **Exemplos** de aplicações de função usando expressões lambda:

```
(fun x -> 2*x + 1) 8
~> 17
```

```
(fun a -> (a, 2*a, 3*a)) 5
~> (5, 10, 15)
```

```
(fun x y -> sqrt (x*.x +. y*.y)) 3. 4.
~> 5.
```

```
(fun (x1, y1) (x2, y2) ->
  sqrt ((x2 -. x1)**2. +. (y2 -. y1)**2.))
(6. , 7.)
(9. , 11.)
~> 5.
```

10.2.4 Exercícios

Tarefa 10.1

Escreva uma função anônima que recebe uma tripla formada pelo nome, peso e altura de uma pessoa e resulta no seu índice de massa corporal, dado pela razão entre o peso e o quadrado da altura da pessoa.

Tarefa 10.2

Escreva uma expressão para selecionar (filtrar) os elementos múltiplos de 3 em uma lista de números. Utilize a função `List.filter : ('a -> bool) -> 'a list -> 'a list`. Especifique a função que determina a propriedade a ser satisfeita pelos elementos selecionados usando uma expressão lambda.

Tarefa 10.3

Determine o tipo mais geral da seguinte expressão:

```
fun a (m, n) -> if a then (m+.n)**2. else (m+.n)**3.
```

Tarefa 10.4

Composição de funções é uma operação comum em Matemática, que a define como

$$(f \circ g)(x) = f(g(x))$$

Em OCaml podemos definir uma função para compor duas outras funções dadas como argumentos. O resultado é uma função: a função composta.

Defina a função `composta` que recebe duas funções como argumentos e resulta na função composta das mesmas. Use uma definição local para definir a função resultante:

```
let composta f g =  
  let ...  
  in  
  ...
```

Tarefa 10.5

1. Escreva outra definição para a função `composta` usando uma expressão lambda para determinar o seu resultado. Nesta versão não use definições locais.
2. Determine o tipo mais geral da função `composta`.
3. Teste a função `composta` calculando o tipo e o valor da expressão

```
(composta (fun x -> x mod 2 = 0) String.length) "linguagens modernas"
```

10.2.5 Expressões lambda with `function`

Existe uma outra forma de escrever uma expressão lambda em OCaml onde é possível escolher entre várias alternativas baseando-se no casamento de padrão. A função poderá ter um único argumento, porém pode-se especificar o parâmetro formal usando vários padrões, cada um associado a uma alternativa, de maneira similar à expressão `match`. Esta forma de expressão lambda é introduzida pela palavra reservada `function`.

```
function padrão1 -> res1  
      |  
      |  
      | padrãon -> resn
```

Veja alguns exemplos de expressão lambda com múltiplas alternativas:

Função anônima que calcula o dobro de um número:

```
function x -> x + x
```

O tipo desta expressão lambda é `int -> int`.

Função anônima que recebe uma lista e calcula a soma de seus 2 primeiros elementos:

```
function [] -> 0  
  | x::[] -> x  
  | x::y::_ -> x + y
```

cujo tipo é `int list -> int`.

10.3 Aplicação parcial de funções

10.3.1 Aplicação parcial de funções

- Uma função com *múltiplos argumentos* pode também ser considerada como uma função que retorna outra função como resultado.

10.3.2 Aplicação parcial de funções: exemplos

- Seja a seguinte função:

```
let f x y = 2*x + y
```

A função `f` recebe dois argumentos inteiros `x` e `y` e resulta na soma $2*x + y$.

- Alternativamente esta função pode ser definida em duas etapas:

```
let f' x =  
  let h y = 2*x + y  
  in h
```

A função `f'` recebe um argumento inteiro `x` e resulta na função `h`, que por sua vez recebe um argumento inteiro `y` e calcula $2*x + y$.

- Aplicando a função:

```
% f' 2 3  
~> (f' 2) 3  
~> h 3  
~> 2*2 + 3  
~> 7
```

- As funções `f` e `f'` produzem o mesmo resultado final, mas `f` foi definida de uma forma mais breve.
- Podemos ainda definir a função usando uma expressão lambda:

```
% let f'' x =  
  fun y -> 2*x + y
```

Da mesma forma que `f'`, a função `f''` recebe um argumento inteiro `x` e resulta em uma função. Esta função recebe um argumento inteiro `y` e calcula $2*x + y$.

- Aplicando a função:

```
% f'' 2 3  
~> (f'' 2) 3  
~> (fun y -> 2*2 + y) 3  
~> 2*2 + 3  
~> 7
```

- Podemos ainda definir a função usando duas expressões lambda:

```
% f''' =  
  fun x -> (fun y -> 2*x + y)
```

- Aplicando a função:

```
% f''' 2 3
~> (fun x -> (fun y -> 2*x + y)) 2 3
~> (fun y -> 2*2 + y) 3
~> 2*2 + 3
~> 7
```

- Todas as versões apresentadas para a função f (f , f' , f'' e f''') são equivalentes.
- Portanto a função f pode ser considerada como uma função que recebe um argumento e resulta em outra função que, por sua vez, recebe outro argumento e resulta na soma do dobro do primeiro argumento com o segundo argumento.
- Isto permite a *aplicação parcial* da função:

```
let g = f 5 in (g 8, g 1)
~> (18, 11)

List.map (f 2) [1; 8; 0; 19; 5]
~> [5; 12; 4; 23; 9]

(f 2 % String.length) "entendeu?"
~> 13

List.filter (not % even % f 10) [1; 8; 0; 19; 5]
~> [1; 19; 5]
```

- Outro exemplo: multiplicação de três números:

```
let mult x y z = x * y * z
```

A função `mult` recebe três argumentos e resulta no produto destes argumentos.

- Na verdade `mult` recebe um argumento de cada vez. Ou seja, `mult` recebe um inteiro x e resulta em uma função que por sua vez recebe um inteiro y e resulta em outra função, que finalmente recebe um inteiro z e resulta no produto $x * y * z$.
- Este entendimento fica claro quando usamos expressões lambda para definir a função de maneira alternativa:

```
let mult' =
  fun x -> fun y -> fun z -> x * y * z
```

10.4 Currying

10.4.1 Funções *curried*

- Outra opção para passar vários argumentos em uma aplicação de função é formar uma estrutura de dados com os dados desejados e passar a estrutura como argumento.
- Neste caso fica claro que haverá um único argumento, que é a estrutura de dados.
- Exemplo: usando uma tupla:

```
somaPar (x, y) = x + y
```

A função `somaPar` recebe *um único* argumento que é um par, e resulta na soma dos componentes do par.

- Evidentemente este mecanismo não permite a aplicação parcial da função.
- Funções que *recebem os seus argumentos um por vez* são chamadas de **funções *curried***¹, celebrando o

¹Funções *curried* às vezes são chamadas de **funções *currificadas*** em português.

trabalho de **OCaml Curry** no estudo de tais funções.

- Funções com mais de um argumento *curried*, resultando em funções aninhadas.

10.4.2 Por que *currying* é útil?

- Funções *curried* são mais flexíveis do que as funções com tuplas, porque muitas vezes funções úteis podem ser obtidas pela *aplicação parcial* de uma função *curried*.
- Por exemplo:

```
(* função que divide 100 pelo seu argumento *)  
(/) 100 :: int -> int  
  
(* função que verifica se 'a' é elemento de uma lista de caracteres *)  
List.mem 'a' :: char list -> bool
```

10.4.3 Convenções sobre *currying*

- Para evitar excesso de parênteses ao usar funções *curried*, duas regras simples foram adotadas na linguagem OCaml:
- A seta `->` (construtor de tipos função) *associa-se à direita*.
- Exemplo:

```
int -> int -> int -> int
```

significa

```
int -> (int -> (int -> int))
```

- A *aplicação de função* tem *associatividade à esquerda*.
- Exemplo:

```
mult x y z
```

significa

```
((mult x) y) z
```

- A menos que seja explicitamente necessário o uso de tuplas, todas as funções em OCaml são normalmente definidas na forma *curried*.

10.5 Utilidade de expressões lambda

- Expressões lambda podem ser usadas para dar um sentido formal para as funções definidas usando *currying* e para a *aplicação parcial de funções*.
- **Exemplo:**
A função

```
let soma x y = x + y
```

pode ser entendida como

```
let soma = fun x -> (fun y -> x + y)
```

isto é, soma é uma função que recebe um argumento x e resulta em uma função que por sua vez recebe um argumento y e resulta em $x+y$.

```
soma  
~> fun x -> (fun y -> x + y)
```

```
soma 2  
~> (fun x -> (fun y -> x + y)) 2  
~> fun y -> 2 + y
```

```
soma 2 3  
~> (fun x -> (fun y -> x + y)) 2 3  
~> (fun y -> 2 + y) 3  
~> 2 + 3  
~> 5
```

- Expressões lambda também são úteis na definição de *funções que retornam funções como resultados*.
- **Exemplo:**
A função `const` definida na biblioteca retorna como resultado uma função constante, que sempre resulta em um dado valor:

```
let const x _ = x
```

```
const 6 0 ~> 6  
const 6 1 ~> 6  
const 6 2 ~> 6  
const 6 9 ~> 6  
const 6 75 ~> 6
```

```
h = const 6 ~> _ -> 6
```

```
h 0 ~> 6  
h 4 ~> 6  
h 75 ~> 6
```

A função `const` pode ser definida de uma maneira mais natural usando expressão lambda, tornando explícito que o resultado é uma função:

```
const x =  
  fun _ -> x
```

- Expressões lambda podem ser usadas para evitar a nomeação de funções que são *referenciados apenas uma vez*.

10.5.1 Exercícios

Tarefa 10.6

Mostre como a definição de função *curried*

```
let mult x y z = x * y * z
```

pode ser entendida em termos de expressões lambda.

Dica: Redefina a função usando expressões lambda.

11 FUNÇÕES DE ORDEM SUPERIOR

Resumo

Uma função é conhecida como *função de ordem superior* quando ela tem uma função como argumento ou resulta em uma função. Nesta aula vamos aprender sobre funções de ordem superior.

Sumário

11.1 Funções de Ordem Superior	11-1
11.2 Composição de funções	11-1
11.3 A função <code>List.filter</code>	11-2
11.4 A função <code>map</code>	11-3
11.5 <code>fold_left</code>	11-3
11.6 <code>fold_right</code>	11-4
11.7 Cupom fiscal do supermercado	11-5

11.1 Funções de Ordem Superior

Uma **função de ordem superior** é uma função que

- tem outra *função* como *argumento*, ou
- produz uma *função* como *resultado*.

11.2 Composição de funções

Composição de funções é uma operação comum na Matemática. Dadas duas funções f e g , a função composta $f \circ g$ é definida por

$$(f \circ g)(x) = f(g(x))$$

Ou seja, quando a função composta $f \circ g$ é aplicada a um argumento x , primeiramente g é aplicada a x , e em seguida f é aplicada a este resultado gx .

Embora a operação de composição de funções seja muito utilizada na programação funcional, ela não é definida na biblioteca padrão do OCaml. Porém ela pode ser facilmente definida, como pode ser visto a seguir. Chamaremos a função de `compose`.

A função `compose` recebe duas funções como argumento e resulta em uma terceira função que é a *composição* das duas funções dadas.

Observe que a função `compose` é uma função de ordem superior, pois recebe duas funções como argumento e resulta em outra função.

Exemplos de composição de funções

```
compose sqrt abs_float
  ~> a função composta de sqrt e abs_float

(compose sqrt abs_float) 9.
  ~> 3.

(compose sqrt abs_float) (16. -. 25.)
  ~> 3.

(compose sqrt (compose abs_float sin)) (3. *. 3.1415 /. 2.)
  ~> 0.999999995171113154

(compose not (function [] -> true | _ -> false)) [10; 20; 30]
  ~> true

(compose sqrt (compose abs_float snd)) ('Z', -36.0)
  ~> 6.
```

Definição de compose

```
let compose f g =
  fun x -> f (g x)
```

O tipo de compose é:

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

11.3 A função List.filter

A função `List.filter` da biblioteca padrão recebe uma função e uma lista como argumentos, e seleciona (filtra) os elementos da lista para os quais a função dada resulta verdadeiro.

Note que `filter` é uma função de ordem superior, pois recebe outra função como argumento.

Exemplos de aplicação de filter

```
List.filter (fun x -> x mod 2 = 0) [1; 8; 10; 48; 5; -3]
  ~> [8; 10; 48]

List.filter (fun x -> x mod 2 <> 0) [1; 8; 10; 48; 5; -3]
  ~> [1; 5; -3]

List.filter (function '0' .. '9' -> true | _ -> false)
  ['A'; '1'; '8'; '6'; 'B'; '7'; '0']
  ~> ['1'; '8'; '6'; '7'; '0']

List.filter (compose not (fun s -> String.length s = 0))
  ["abc"; ""; "ok"; ""]
  ~> ["abc"; "ok"]
```

Definição de filter

```
let rec filter lst =  
  match lst with  
  | [] -> []  
  | h::t -> if f h then  
             x : filter f t  
             else  
               filter f t
```

O tipo de filter é:

```
('a -> bool) -> 'a list -> 'a list
```

11.4 A função map

A função `List.map` da biblioteca padrão recebe uma função e uma lista como argumentos, e aplica a função a cada um dos elementos da lista, resultando na lista dos resultados. `List.map` é uma função de ordem superior, pois recebe outra função como argumento.

Exemplos de aplicação de map

```
List.map sqrt [0.0; 1.0; 4.0; 9.0]  
  ~> [0.0; 1.0; 2.0; 3.0]  
  
List.map succ [1; 8; 6; 10; -49]  
  ~> [2; 9; 7; 11; -48]  
  
List.map (fun n -> n mod 2 = 0) [8; 10; -3; 48; 5]  
  ~> [true; true; false; true; false]  
  
List.map (fun x -> x >= '0' && x <= '9') ['A'; '1'; '8'; ' '; 'B'; '7']  
  ~> [false; true; true; false; false; true]  
  
List.map String.length ["ciência"; "da"; "computação"]  
  ~> [7; 2; 12]  
  
List.map (compose sqrt (compose abs_float snd)) [('A', 100.0); ('Z', -36.0)]  
  ~> [10.; 6.]
```

Definição de map

```
let rec map f lst =  
  match lst with  
  | [] -> []  
  | h::t -> f h :: map f t
```

O tipo de map é:

```
('a -> 'b) -> 'a list -> 'b list
```

11.5 fold_left

`List.fold_left` reduz uma lista, usando uma função binária e um valor inicial, de forma associativa à esquerda. O elemento inicial e todos os elementos da lista são combinados, da esquerda para a direita, usando a função dada, para calcular o resultado.

$$\begin{aligned} \text{fold_left } (\oplus) \ e \ [x_0, x_1, \dots, x_{n-1}] \\ \equiv \\ (\dots ((e \oplus x_0) \oplus x_1) \dots) \oplus x_{n-1} \end{aligned}$$

Exemplos de aplicação de `List.fold_left`

```
List.fold_left (+) 0 [] ~> 0
List.fold_left (+) 0 [1] ~> 1
List.fold_left (+) 0 [1; 2] ~> 3
List.fold_left (+) 0 [1; 2; 4] ~> 7
List.fold_left ( * ) 1 [5; 2; 4; 10] ~> 400
List.fold_left (&&) true [2>0; 6 mod 2 = 0; 5 mod 2 <> 0] ~> true
List.fold_left (||) false [2>3; 6 mod 2 = 0; 5 mod 2 <> 0] ~> true
```

Definição

```
let rec fold_left f z lst =
  match lst with
  | [] -> z
  | h::t -> fold_left f (f z h) t
```

O tipo de `List.fold_left` é:

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

11.6 fold_right

`List.fold_right` reduz uma lista, usando uma função binária e um valor inicial, de forma associativa à direita. Todos os elementos da lista e o elemento inicial são combinados, da direita para a esquerda, usando a função dada, para calcular o resultado.

$$\begin{aligned} \text{fold_right } (\oplus) \ [x_0, \dots, x_{n-2}, x_{n-1}] \ e \\ \equiv \\ x_0 \oplus (\dots (x_{n-2} \oplus (x_{n-1} \oplus e)) \dots) \end{aligned}$$

Exemplos de aplicação de `fold_right`

```
List.fold_right (+) [] 0 ~> 0
List.fold_right (+) [1] 0 ~> 1
List.fold_right (+) [1; 2] 0 ~> 3
List.fold_right (+) [1; 2; 4] 0 ~> 7
List.fold_right ( * ) [5; 2; 4; 10] 1 ~> 400
List.fold_right (&&) [2>0; 6 mod 2 = 0; 5 mod 2 <> 0] true ~> true
List.fold_right (||) [2>3; 6 mod 2 = 0; 5 mod 2 <> 0] false ~> true
```

Definição

```
let rec fold_right f lst z =
  match lst with
  | [] -> z
  | h::t -> f h (fold_right f t z)
```

O tipo de `List.fold_right` é:

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

11.7 Cupom fiscal do supermercado

Nas tarefas que se seguem temos por objetivo desenvolver uma aplicação em OCaml para **automatizar o caixa de um supermercado** usando técnicas de manipulação de listas empregando funções de ordem superior e expressões lambda.

Suponha que um leitor de código de barras é usado no caixa de um supermercado para produzir uma lista de códigos de barras a partir dos produtos que se encontram em um carrinho de compras contendo os produtos a serem adquiridos. Usando os códigos de barra deve ser criada uma nota descritiva da compra. Considere por exemplo a seguinte lista de códigos de barra:

```
[1234; 4719; 3814; 1112; 1113; 1234]
```

Esta lista deve ser convertida para uma conta como mostra a figura a seguir:

```
OCaml Stores

Dry Sherry, 1lt.....5.40
Fish Fingers.....1.21
Orange Jelly.....0.56
Hula Hoops (Giant).....1.36
Unknown Item.....0.00
Dry Sherry, 1lt.....5.40

Total.....13.90
```

Primeiro devemos decidir como modelar os objetos envolvidos. Códigos de barra e preços (**em centavos**) podem ser representados por números inteiros, e nomes de mercadorias podem ser representados por strings. Então usaremos os seguintes tipos:

```
type nome    = string
type preco   = int
type codigo  = int
```

A conversão dos códigos de barras será baseada em um banco de dados que relaciona códigos de barras, nomes de mercadorias, e preços. Usaremos uma lista para representar o banco de dados de mercadorias:

```
type mercadorias = (codigo * nome * preco) list
```

O banco de dados para o exemplo dado é:

```
let tabelaMercadorias : mercadorias =
  [ (4719, "Fish Fingers",      121 )
  ; (5643, "Nappies",          1010)
  ; (3814, "Orange Jelly",     56  )
  ; (1111, "Hula Hoops",       21  )
  ; (1112, "Hula Hoops (Giant)", 133 )
  ; (1234, "Dry Sherry, 1lt",  540 )
  ]
```

O objetivo do programa é primeiramente converter uma lista de códigos de barra em uma lista de pares do tipo `nome * preco` por meio de uma consulta à tabela de mercadorias. Em seguida esta lista de pares deve ser convertida em uma string para exibição na tela. Usaremos as seguintes definições de tipo:

```
type carrinho = codigo list
type conta    = (nome * preco) list
```

para representar um carrinho de compras e uma conta (cupom fiscal) correspondente a uma compra.

Tarefa 11.1: Formatação do preço em reais

Defina uma função `formata_centavos` : `preco` -> **string** que recebe o preço em centavos e resulta em uma string representando o preço em reais.

Por exemplo:

```
formata_centavos 1023 ~> "10.23"
formata_centavos 56015 ~> "560.15"
formata_centavos 780 ~> "7.80"
formata_centavos 309 ~> "3.09"
formata_centavos 15 ~> "0.15"
formata_centavos 5 ~> "0.05"
```

Use as funções `(/)`, `mod` e `string_of_int`. Observe que ao dividir o preço em centavos por 100, o quociente corresponde à parte inteira do preço em reais, e o resto corresponde à parte fracionária do preço em reais. Preste atenção no caso do resto menor do que 10: deve-se inserir um 0 à esquerda explicitamente.

Tarefa 11.2: Formatação de uma linha do cupom fiscal

Defina uma função `formata_linha` : `nome * preco` -> **string** que recebe um par formado pelo nome e preço de uma mercadoria e resulta em uma string representando uma linha da conta do supermercado.

Por exemplo:

```
formata_linha ("Dry Sherry, 1lt", 540) ~> "Dry Sherry, 1lt.....5.40\n"
formata_linha ("Nappies, 1lt", 1010) ~> "Nappies.....10.10\n"
```

O tamanho de uma linha em uma conta deve ser 30. Use a variável abaixo para representar este valor.

```
let tamanhoLinha : int = 30
```

Use as funções `(^)`, `string_of_int`, `String.length` e `String.make` da biblioteca padrão, e a função `formata_centavos` da tarefa 11.1.

A função `String.make` : `int` -> `char` -> **string** recebe um número inteiro `n` e um caracter `c` e resulta em uma string de comprimento `n` onde todos os elementos são `c`. Por exemplo:

```
String.make 5 'A' ~> "AAAAA"
String.make 8 '.' ~> "....."
```

Tarefa 11.3: Formatação de várias linhas do cupom fiscal

Defina a função `formata_linhas` : `(nome * preco) list -> string` que recebe uma lista de pares formados pelos nomes das mercadorias e seus respectivos preços em uma compra, e resulta na string correspondente ao corpo da conta do supermercado.

Por exemplo:

```
formata_linhas [ ("Dry Sherry, 1lt",    540)
                 ; ("Fish Fingers",    121)
                 ; ("Orange Jelly",    056)
                 ; ("Hula Hoops (Giant)", 136)
                 ; ("Unknown Item",    000)
                 ; ("Dry Sherry, 1lt",    540)
                 ]
```

~~

```
"Dry Sherry, 1lt.....5.40
Fish Fingers.....1.21
Orange Jelly.....0.56
Hula Hoops (Giant).....1.36
Unknown Item.....0.00
Dry Sherry, 1lt.....5.40"
```

Use a função `formata_linha` da tarefa 11.3 para obter as linhas correspondentes a cada produto, e concatene estas linhas usando a função `(^)` da biblioteca padrão. Não use recursividade explícita, mas use as funções `List.map` e `List.fold_right` ou `List.fold_left` da biblioteca padrão.

Tarefa 11.4: Formatação do total

Defina a função `formata_total` : `preco -> string` que recebe o valor total da compra, e resulta em uma string representando a parte final da conta do supermercado.

Por exemplo:

```
formata_total 1390 ~> "\nTotal.....13.90"
```

Use as dicas da tarefa 11.2.

Tarefa 11.5: Formatação do cupom fiscal

Defina a função `formata_conta : conta -> string` que recebe a lista dos itens comprados e resulta na string representando a conta do supermercado, já formatada.

Por exemplo:

```
formata_conta [ ("Dry Sherry, 1lt", 540)
                ; ("Fish Fingers", 121)
                ; ("Orange Jelly", 056)
                ; ("Hula Hoops (Giant)", 136)
                ; ("Unknown Item", 000)
                ; ("Dry Sherry, 1lt", 540)
                ]
```

resulta na string que é exibida pela função `print_string` como

```
OCaml Stores

Dry Sherry, 1lt.....5.40
Fish Fingers.....1.21
Orange Jelly.....0.56
Hula Hoops (Giant).....1.36
Unknown Item.....0.00
Dry Sherry, 1lt.....5.40

Total.....13.90
```

Use as funções definidas nas tarefas 11.3 e 11.4.

Tarefa 11.6: Cálculo do valor total da compra

Defina a função `calcula_total : conta -> preco` que recebe uma conta (lista de pares formados pelo nome e preço das mercadorias de uma compra), e resulta no preço total da compra.

Por exemplo:

```
calcula_total [("a", 540); ("b", 121); ("c", 12)] ~> 673
calcula_total [("vinho", 3540); ("carne", 7201)] ~> 10741
calcula_total [] ~> 0
```

Não use recursividade explícita, mas use as funções `List.map` e `List.fold_left` da biblioteca padrão.

Tarefa 11.7: Pesquisa do código de um produto

Defina uma função `procura_codigo : mercadorias -> codigo -> nome * preco` que recebe o banco de dados com os nomes e preços das mercadorias disponíveis no supermercado e o código de barras da mercadoria comprada, e resulta no par formado pelo nome e pelo preço da mercadoria, de acordo com o banco de dados. Se o código de barras não constar no banco de dados, o resultado deve ser o par `("Unknown Item", 0)`.

Por exemplo:

```
procura_codigo tabelaMercadorias 5643 ~> ("Nappies", 1010)
procura_codigo tabelaMercadorias 9999 ~> ("Unknown Item", 0)
```

Tarefa 11.8: Criação da conta da compra

Defina a função `cria_conta` : `mercadorias` -> `carrinho` -> `conta` que recebe o banco de dados com os nomes e preços das mercadorias disponíveis no supermercado, e a lista de códigos de barra correspondente a uma compra, e resulta na lista dos pares do tipo `nome * preco` para as mercadorias compradas.

Por exemplo:

```
cria_conta tabelaMercadorias [3814; 5643]
~~~ [("Orange Jelly", 56); ("Nappies", 1010)]
```

Use uma aplicação parcial da função `procura_codigo` definida na tarefa 11.7 e a função `List.map` da biblioteca padrão. Não use recursão explícita.

Tarefa 11.9: Criação do cupom fiscal

Defina a função `faz_compra` : `mercadorias` -> `carrinho` -> `string` que recebe o banco de dados com os nomes e preços das mercadorias disponíveis no supermercado, e a lista de códigos de barra correspondente a uma compra, e resulta na string correspondente à nota da compra.

Use a função `cria_conta` (definida na tarefa 11.8) para criar a conta a partir dos argumentos, e a função `formata_conta` (definida na tarefa 11.5) para converter a conta para string. Use composição de funções.

Tarefa 11.10: main

Defina a variável `main` : `()` como uma expressão sequência cuja avaliação produz uma interação com o usuário. Quando `main` for avaliada, o usuário deve digitar os códigos de barras das mercadorias compradas e em seguida a conta correspondente do supermercado deve ser exibida na tela.

Tarefa 11.11

Compile a aplicação gerando um **programa executável**. Teste a aplicação.