Introdução à Programação Funcional em OCaml

Mário Pereira and Simão Melo de Sousa
RELEASE - RELiablE And SEcure computation Group
LIACC & Departamento de Informática
Universidade da Beira Interior

Documento de Trabalho. Versão de 26 de Fevereiro de 2012

Conteúdo

Aviso Prévio

- A redacção deste documento baseou-se fortemente na bibliografia indicada. Parece-nos então óbvio que a leitura e a aprendizagem directa pelas obras originais é recomendada, e mesmo essencial à compreensão profunda das noções aqui apresentadas;

- O português não é a língua materna do principal autor e o presente documento encontra-se em fase de elaboração pelo que se agradece e até se incentiva qualquer sugestão ou correcção.

Bibliografia

Referências principais usadas neste documento e das quais se aconselha leitura são [4, 6, 2] (Com a autorização concedida gentilmente por Jean-Christophe Filliâtre e por Xavier Leroy).
As referências secundária utilizadas (por consultar igualmente) são [7, 3, 9, 8, 5, 10]

1 Introdução

A linguagem *LISP*, criada no *M.I.T.* nos anos 1960, é o patriarca das linguagens funcionais. Muitas variantes desta linguagem apareceram, como o *ELISP* que é a linguagem de programação associada ao editor de texto *Emacs*. Outro exemplo é a linguagem *Scheme* que está, por exemplo, na base do sistema de processamento de imagens *GIMP*.

CAML é o acrónimo para *Categorical Abstract Machine Language* que sublinha as ligações que esta linguagem alimenta com o cálculo λ e a teoria das categorias. A CAM (de *Categorical Abstract Machine*) é uma máquina abstracta (diz-se também máquina virtual) capaz de definir e de executar funções. A outra filiação do nome OCaml é a linguagem ML (de *Meta Language*). As linguagens da família *ML* foram desenvolvidas no início dos anos 1980. Na origem esteve a linguagem desenvolvida por Robert Milner em 1978. Da descendência destacamos os três dialectos principais que são o SML-NJ (New-Jersey, USA) , Moscow-ML (Rússia) e o OCaml (França). Um ramo "dissidente" desta descendência mas importante é a família das linguagens

qualificadas de *preguiçosas*[1] como o Haskell, que são capazes naturalmente de trabalhar com tipos de dados de tamanho infinito.

O primeiro compilador Caml produzindo código para a máquina CAM foi escrito em 1984. Actualmente o compilador OCaml produz código para uma máquina virtual diferente e mais eficiente, chamada ZINC e escrita pelo Xavier Leroy. Desde então a linguagem OCaml tem conseguido superar um desafio que poucas linguagens conseguiram: aliar uma boa fundamentação teórica, sinónima de robustez e fiabilidade, à eficácia do código produzido. Um argumento, subjectivo, é por exemplo o comparativo de linguagens desenvolvido em [1] em que a linguagem Caml fica imediatamente atrás da linguagem C. Se a linguagem OCaml é uma linguagem popular nos meios académicos[2] e no mundo da investigação, é também uma linguagem utilizada em ambiente industrial. Citemos por exemplo o *Ocaml Consortium* [3] que testemunha tal interesse pelos industriais, a indústria das comunicações (como a France Telecom), a industria aeronáutica (Dassault,...), a finança (Janes Street Capital, NY) ou até mesmo Microsoft (não só para a sua divisão *Research*, mas também desenvolvimento).

1.1 Herança ML

A linguagem OCaml, como todas as linguagens herdeiras de ML, possuí as características seguintes:

- As funções são valores de "primeira classe": estas podem ser argumentos de outras funções ou mesmo o resultado de um cálculo. Neste sentido diz-se da linguagem OCaml que é uma linguagem de funções de ordem superior.

- A linguagem é fortemente tipificada: a qualquer valor está associado um tipo. A verificação sistemática e rigorosa da compatibilidade entre os tipos dos parâmetros formais e os tipos dos parâmetros efectivos é uma das características da linguagem. Tal permite eliminar grande parte dos erros introduzidos por "distracção" e contribuí para uma execução segura, fiável e robusta.

[1] Preguiçosas, no bom sentido: só executam certas tarefas quando estritamente necessárias.

[2] O Caml-light é por exemplo uma variante Caml particularmente adaptada à aprendizagem da programação.

[3] http://caml.inria.fr/consortium/.

- A tipagem é inferida automaticamente: o sistema de tipo de OCaml é particularmente expressivo e o problema da inferência de tipo continua no entanto decidível. Isto é, o programador não tem de declarar o tipo das expressões que define. É uma opção. O sistema de tipos subjacente à linguagem OCaml é suficientemente poderoso para permitir que se deduza esta informação a partir da sintaxe da expressão considerada.

- A tipagem é estática: todos os tipos podem ser (e são) deduzidos na fase da compilação. Desta forma tal verificação não é necessária durante a execução tornando essa última mais rápida.

- A tipagem é polimórfica: o algoritmo de inferência de tipo do OCaml sabe reconhecer tipos polimórficos. Voltaremos a este assunto secção precedente, mas podemos desde já realçar que tal característica permite, de forma elegante, uma programação modular e genérica.

- Mecanismos de definição e de gestão de excepções: permite uma programação capaz de interferir com fluxo de controlo (nome genérico para definir o decorrer de uma execução).

- Mecanismos de filtragem de motivos (concordância de padrão): este mecanismo possibilita uma programação, muito conveniente, baseada na analise dos casos possíveis.

- A memória é gerida por um recuperador automático de memória (Garbage Colector ou GC, em inglês): não há gestão explícita da memória. O GC trata de tudo automaticamente e de forma eficiente.

Além dessas características, OCaml:

- dispõe de mecanismos próprios à programação imperativa;

- dispõe de uma camada objecto;

- permite uma integração completa de programas C com programas OCaml;

- suporta a programação concorrente, distribuída e paralela;

- dispõe de uma base contributiva importante, diversificada e completa (como atestam o sub-site *Hump* do site oficial ou ainda o agregador

de projectos Caml *ocamlforge*, etc.). Estes pontos de entrada permitem, por exemplo, ter acesso a bibliotecas ricas que complementam a biblioteca standard.

2 Compilação, Execução e Bibliotecas

Um programa OCaml apresenta-se como uma sequência de expressões OCaml eventualmente colocadas num ficheiro ".ml" e/ou ".mli" (um ficheiro ".mli" agrupa a interface do ficheiro ".ml" ao qual está relacionado, tal como um ficheiro ".h" é o "header" relacionado com um ficheiro ".c" na linguagem C).

Podemos executar programas OCaml de três formas diferentes. Para cada uma delas serão precisos procedimentos diferentes.

2.1 Execução

Interpretação pelo *toplevel*: O ciclo de interacção (*toplevel* na terminologia inglesa) interpreta frase por frase (ou expressão OCaml por expressão) o programa OCaml. Cada frase termina por dois *pontos e virgulas*: "`;;`". Para cada expressão interpretada o *toplevel* devolve o tipo e o valor da expressão em caso de sucesso. O toplevel é um programa chamado `ocaml`. Os exemplos deste documento foram todos escrito e executados utilizando o toplevel.

Para carregar uma biblioteca (OCaml ou pessoal) é preciso dispor do ficheiro ".cmo" da biblioteca. O comando no toplevel é #load "fich.cmo";;. Pode-se também optar por uma inclusão textual (equivalente a um *copy-past* do ficheiro para o toplevel). Neste caso o comando é #use "fich.ml";;

Dica: preferir o comando `ledit ocaml` ou `rlwrap ocaml` ao comando `ocaml` se os programas `ledit` ou `rlwrap` estiverem instalado na máquina hóspede. Estes programas permitem guardar um histórico dos comandos/expressões OCaml introduzidos, à semelhança do `doskey` do `DOS`.

Execução pela máquina virtual: a semelhança do Java, os programas OCaml podem ser compilados para serem executados por uma máquina virtual. Temos assim um programa portável e seguro (a máquina virtual tem mecanismos de segurança embebidos) que pode ser executado, sem qualquer modificação, numa grande variedade de arquitecturas. Para obter tal programa, a compilação terá de ser efectuada utilizando o compilador para bytecode. A máquina virtual OCaml chama-se `ocamlrun`. A execução dum programa bytecode, digamos `prog`, a partir dum terminal pode assim ser realizada alternativamente de duas formas:

1. `ocamlrun prog`
2. `./prog`

No caso do programa bytecode ter a máquina virtual embebida (graças à opção `-custom` do compilador), basta utilizar a segunda alternativa para a execução do programa;

Execução pela máquina hospeda: o utilizador pode querer privilegiar a eficiência em relação a portabilidade do programa definido. Neste caso preferirá utilizar o compilador nativo `ocamlopt` para produzir código máquina bem mais rápido do que o código interpretado. Nesta configuração:

- o programa produzido é específico à máquina em que foi compilado mas ganha-se um factor de rapidez à volta de 10;
- o código produzido é código máquina e é executado directamente pela máquina hospeda (como qualquer programa executável compilado pelo compilador `gcc` por exemplo).

2.2 Compilação

Existem dois compiladores (e duas variantes optimizadas de cada um deles, sufixadas por ".opt").

ocamlc: este compilador gera código intermédio adequado para a máquina virtual OCaml;

ocamlopt: este compilador gera código máquina adequado para ser executado directamente pela máquina hospeda.

Ambos aceitam as opções habituais dos compiladores da GNU (como o gcc).

Destacamos as opções seguintes:

- `-i`: gera para a saída standard toda a informação de tipos contida no ficheiro compilado. Útil para obter um ficheiro ".mli" por defeito;
- `-c` compila mas não faz a edição das ligações (não faz o *linking*). Aquando da utilização do compilador `ocamlc`, permite gerar o ".cmo" correspondente;
- `-custom` permite a criação de executáveis autónomos. Aquando da utilização do compilador `ocamlc`, permite incorporar uma máquina virtual ao código produzido. Com um executável deste tipo não é assim necessária a presença duma máquina virtual na máquina hospeda para que o programa possa ser executado..
- `-o` *`nome`*: indica o nome desejado para o executável;
- `-I` *`caminho`*: acrescenta o caminho na lista das pastas onde procurar bibliotecas.
- `-annot`: produz informação suplementar sobre tipos, *tail-calls*, *bindings*, etc... para o ficheiro `<filename>.annot`. Útil para quem utiliza o editor emacs.
- `-a`: compila para o formato biblioteca.
- `-pp <command>`: activa o preprocessamento `<command>` via camlp4 ou camlp5.
- `-unsafe`: gera código sem verificação dos acessos a strings e vectores.

2.3 Acesso às bibliotecas

As funções, valores das bibliotecas (também designados de *módulos* em OCaml) são acessíveis de duas formas: directamente (se as bibliotecas foram previamente carregadas por um `open` ou `use`) ou por nome qualificados. Por exemplo: seja `f` uma função da biblioteca `B`. O acesso directo a `f` é `f`, o acesso qualificado é `B.f`. Repare que o acesso directo é condicionado ao facto de não haver ambiguidades no nome `f`. Imagine que duas bibliotecas abertas

fornecem uma função `f`, terá neste caso de qualificar o nome `f` para ter a certeza de chamar a função realmente desejada. Para ser preciso, quando dois módulos abertos disponibilizam o mesmo valor `f`, então, pelas regras habituais de visibilidade, a invocação de `f` designa o valor `f` do ultimo módulo aberto.

Os comandos para o carregamento de bibliotecas são, se tomarmos o exemplo da biblioteca chamada `bib`:

- `#use "bib.cmo"` no toplevel;
- `open Bib` no ficheiro *.ml* destinado para a compilação

No caso dos nomes qualificados, o nome da biblioteca começa sempre por uma maiúscula.

2.4 Ambiente de Programação

O ambiente clássico para a programação OCaml é o editor emacs ou o editor xemacs equipados com o modo tuareg.

Existe igualmente modos OCaml para os IDEs tradicionais. Citemos por exemplo o ODT e o OCAIDE para o IDE eclipse.

De uma forma geral, os editores de texto populares como o editor *crimson* ou o *notepad++* disponibilizam mecanismos satisfatórios de coloração de sintaxe e de compilação em linha de comando.

A instalação de uma distribuição OCaml de base em ambiente *Windows* necessita de um compilador de *assembly* tradicionalmente fornecido por plataformas como o *MinGW* ou *CygWin*. Um tutorial de instalação neste sistema operativo pode ser visualizado aqui.

O utilizador curioso poderá também experimentar o site http://try.ocamlpro.com/ que apresenta um ambiente de execução OCaml online completo.

A distribuição mínima de OCaml encontra-se no site http://caml.inria.fr. Existe no entanto duas plataformas (complementares e relacionadas) que permitem gerir a instalação e a actualização de uma quantidade apreciável de ferramentas, aplicações, bibliotecas, etc.. ligados à linguagem OCaml. Essas são o GODI e o Oasis.

Para a compilação de projectos OCaml existam duas formas alternativas muito práticas disponíveis: primeiro, um Makefile genérico para OCaml e, segundo, uma ferramenta de compilação de projecto Ocamlbuild.

A biblioteca standard de OCaml é, para o uso geral, relativamente completa, mas de facto de dimensão reduzida quando comparada com API de outras linguagens populares. Vários complementos, que visam colmatar esta situação, existem. Citemos por exemplo o OCaml batteries ou a Janes Street core Suite.

De uma forma geral, existem vários repositórios web que agrupam projectos, informações, documentação, recursos, apontadores, etc... ligados ao mundo OCaml. Citemos o Caml Hump, ou ainda Forge OCaml .

3 Núcleo funcional

Comecemos por alguns aspectos fundamentais próprios à linguagem OCaml, que serão em grande parte justificados nesta secção:

OCaml é uma linguagem funcional fortemente tipificada.
Um programa OCaml é uma sequência de expressões ou de declarações
Uma expressão OCaml reduz-se a um valor
Qualquer expressão OCaml tem um e um só tipo bem determinado

OCaml é uma linguagem funcional fortemente tipificada. Como tal, trata-se duma linguagem onde as expressões (nas quais incluímos expressões funcionais) têm um papel central. As construções essenciais da linguagem são assim mecanismos de definição e aplicação de funções.

O resultado do cálculo duma expressão é um valor da linguagem e a execução dum programa é o cálculo de todas as expressões constituindo o programa.

É importante realçar de que se trata duma linguagem em que valor e função pertencem à mesma classe. Não há diferenças de tratamento, por exemplo, entre um inteiro e uma função. Ambos são valores para OCaml. Poderão assim ser calculados ou analisados por uma função. Isto é, poderão ser da mesma forma parâmetro ou até mesmo devolvidos como resultado de uma função. Daí a denominação de funções de ordem superior para as funções de tais linguagens funcionais.

Outro pendente da segunda afirmação (*Um programa* OCaml *é uma sequência de expressões*) é a inexistência de distinção entre instruções e expressões presente tradicionalmente nas linguagens imperativas. O que entendemos ser instruções no paradigma imperativo são expressões como quaisquer

outras, devolvem valores e podem ser usadas como sub-expressões. É assim possível escrever expressões como

```
(1 + (if x>4 then 7 else 9))
```

Admitindo que x seja o valor 5, esta expressão tem por valor 7 e tipo inteiro (`int` em OCaml, como veremos mais adiante).

Apresentaremos os protótipos de funções descritas neste documento utilizando a notação OCaml para as interfaces. De forma resumida, as interfaces OCaml, agrupadas nos ficheiros *.mli* são os equivalente OCaml dos ficheiros `headers` (os ficheiros *.h*) da linguagem C.

```
val nome_da_função_ou_valor : assinatura
```

3.1 Linguagens fortemente tipificadas e Polimorfismo

Antes de passar aos tipos de base e às construções de tipo que OCaml fornece, é importante voltar a frisar que OCaml é uma linguagem *fortemente tipificada* com *tipos polimórficos*.

Que significa tal afirmação? A primeira parte indica que qualquer valor OCaml tem necessariamente um tipo e que o compilador/*toplevel* de OCaml sancionará qualquer falha a essa regra (com um erro na compilação ou na interpretação). Uma propriedade interessante do OCaml é a sua capacidade em adivinhar (o termo exacto é *inferir*) o tipo de cada valor. Não necessitamos então fornecer explicitamente em cada valor o tipo ao qual ele pertence. OCaml pode fazê-lo por nós. Deveremos no entanto ter o cuidado em definir funções (mais genericamente expressões, valores) que respeitam essa classificação em tipo. OCaml rejeitará qualquer definição, por exemplo, em que é dado um real a uma função esperando um inteiro, mesmo se esse real é 1.0!! Realcemos que, embora pareça falta de flexibilidade, esta característica, comum a muitas linguagens modernas que se preocupam com programação fiável e robusta, é considerada importante e desejável.

A segunda parte, o polimorfismo, diz respeito a capacidade de uma função operar correctamente para elementos de vários tipos que têm uma estrutura em comum. Por exemplo, vectores de inteiros e vectores de reais são sequências de acesso directo que só diferem nos valores que essas sequências contêm. De facto, esta informação pouco importa à função `comprimento` que devolve o comprimento dum vector. O cálculo deste valor pode perfeitamente se *abstrair* do tipo dos elementos. Podemos neste caso considerar o tipo dos valores do vector como uma incógnita, uma *variável (de tipo)*. A nossa função `comprimento` tem assim *várias formas*, conforme o tipo de vector ao qual é

aplicada. Pode *adquirir a forma* ou ainda *especializar-se* numa função que calcula comprimentos de vectores de inteiros, ou ainda pode *adquirir a forma*, de uma função que calcula comprimentos de vectores de caracteres.

Uma linguagem que permite tal abstracção é chamada *polimórfica* (**etimologia**: que tem várias formas). OCaml é uma tal linguagem.

O polimorfismo surge nos tipos OCaml da seguinte forma: um tipo é polimórfico quando a sua definição contempla variáveis, chamadas variáveis de tipo, que podem ser, quando conveniente, instanciadas por outros tipos. As variáveis de tipo são assinaladas por identificadores prefixados por uma plica. Por exemplo `'a T` denota o tipo OCaml `T` polimórfico numa variável `'a`.

Assumindo, por enquanto, a existência do tipo `int` dos inteiros e do tipo `char` dos caracteres, exemplos de instâncias são `int T`, `char T`, `(int T) T` ou mesmo `('a T) T`. Nestes casos `'a` foi instanciado (tomou o valor) respectivamente por `int`, `char`, `(int T)` e `('a T)`. Se imaginarmos que `T` seja um tipo pensado para ser *contentor* de elementos de outros tipos (como os vectores por exemplo), então tais instanciações permitam dispor, respectivamente, de um contentor de: inteiros; caracteres; contentores de inteiros (à semelhança de matrizes de duas dimensões de inteiros); ou finalmente de contentores de um tipo ainda por definir.

Listas, tuplos, vectores, referências são exemplos de tipos polimórficos que descrevemos neste documento.

Mais formalmente, este tipo de polimorfismo designa-se por *polimorfismo paramétrico* e implementa a estratégia de *todos para um*. Uma função, um tipo, aquando da sua definição pode visar todos os casos aplicativos possíveis (e introduz assim variáveis de tipo de acordo com esta visão). Mas quando instanciada, esta é irremediavelmente especializada para o caso do valor da instanciação. Um tipo especializado não pode voltar a ser generalizado.

3.1.1 Conversões explícitas

Para terminar sobre generalidades sobre a tipificação forte, e admitindo a existência dos tipos de base `int`, `float`, `char` ou `string` que descreveremos já a seguir, listamos algumas funções de conversão:

```
# int_of_char;;
- : char -> int = <fun>
# int_of_string;;
```

```
- : string -> int = <fun>
# int_of_float;;
- : float -> int = <fun>
# char_of_int;;
- : int -> char = <fun>
# float_of_string;;
- : string -> float = <fun>
# float_of_int;;
- : int -> float = <fun>
```

Estas são obviamente úteis quando pretendemos realizar um *cast* possível dos tipos dos valores manipulados[4]. Relembramos que OCaml é fortemente tipificado e que ajustes explícitos (no sentido, realizados explicitamente pelo programador) de tipos podem ser requeridos pelo programador. Estas funções podem devolver um valor do tipo desejado mas com perdas de precisão relativamente ao valor original.x

Por exemplo:

```
# int_of_char 'B' ;;
- : int = 66
# int_of_float 1.6 ;;
- : int = 1
```

3.2 Variáveis e funções

Vamos nesta secção introduzir duas características da linguagem OCaml (de facto, das linguagens funcionais) que muda fundamentalmente a forma de desenhar soluções algorítmicas e programar quando comparado com a prática do habitual paradigma imperativo. O programador habitual de C, Java ou C++ normalmente deverá cuidadosamente explorar e incorporar estas duas noções para conseguir a transição para linguagens funcionais.

De forma resumida, a capacidade dum programa em realizar *efeitos colaterais* para além do calculo propriamente dito é algo que as linguagens funcionais suportam de uma forma muito controlada. A linguagem OCaml não foge a esta regra.

[4]Em OCaml, estas funções não transformam o tipo dum determinado valor, mas sim, calculam/devolvem um novo valor do tipo adequado

Assim uma característica saliente, para além das que já foram referidas, das linguagens funcionais às quais OCaml pertence é o caractere implicitamente imutável das variáveis.

A imutabilidade das variáveis em OCaml lhes confere um estatuto ou comportamento semelhante às variáveis matemáticas: representam um e um só valor e pertençam a um conjunto (tipo).

Uma variável quando é introduzida num programa OCaml é necessariamente inicializada com um valor e, não fugindo aos princípios de tipificação forte, está associada a um só tipo.

Os processos algorítmicos desenhados pelo programador não podem quebrar esta regra. Em regra geral quando um algoritmo necessita, dada a sua formulação própria, a alteração (repetida ou não) do valor de uma variável, isto implica, em primeiro, que a formulação não é funcional (segue o paradigma imperativo), logo não pode directamente ser implementada recorrendo à noção básica de variáveis OCaml. A solução é então usar a camada imperativa disponível em OCaml (neste caso as variáveis do algoritmo devem ser implementadas como referencias OCaml) ou então é necessário obter uma formulação funcional do algoritmo (para poder usar a noção primitiva de variável ou de valor).

Voltaremos a esta noção quando serão apresentados os mecanismos de definição de variáveis (secção 3.5).

Outra vertente do mesmo fenómeno é o mecanismo de base para a composição de acções. A sequência, habitual no paradigma imperativo, não é o mecanismo primitivo em OCaml (isto é: executar A; executar B; etc ..., como é habitual em linguagens imperativas). A composição de funções é o mecanismo de base para encadear as acções algorítmicas por implementar.

3.3 Valores, funções e tipos de base

Nesta secção iremos explorar de forma sucinta os diferentes tipos de dados e as funções de base associadas.

3.3.1 Valores Numéricos

Tipo inteiro: `int`
Formato: o formato habitual (sequência dos algarismo definindo o inteiro, sem ponto nem virgulas)
Limites: entre $[-2^{30}, 2^{30}-1]$ nas máquinas 32 bits e entre $[-2^{62}, 2^{62}-1]$ nas

máquinas 64 bits.

Tipo real: `float`
Formato: OCaml segue a norma **IEEE 754** para a representação de flutuantes. Como tal, um flutuante é um número real $m \times 10^n$ representado por um número contendo necessariamente um ponto (que representa a virgula) cujos limites de representação são 53 bits para m e $n \in [-1022, 1023]$.

inteiros (tipo `int`)		**flutuantes** (tipo `float`)	
representação de *um*	1	representação de *um*	1.0 ou 1.
adição	`+`	adição	`+.`
subtracção (ou negação)		subtracção (ou negação)	`.`
multiplicação	$*$	multiplicação	$*.$
divisão (inteira)	/	divisão	/.
resto	**mod**	exponenciação	$**$

algumas funções sobre inteiros	
valor absoluto	`val abs : int -> int`
maior inteiro representável	`val max_int : int`
menor inteiro representável	`val min_int : int`
etc...	

algumas funções sobre flutuantes	
arredondamento para o inteiro superior (em formato `float`)	`val ceil : float -> float`
arredondamento para o inteiro inferior (em formato `float`)	`val floor : float -> float`
raíz quadrada	`val sqrt : float -> float`
exponencial	`val exp : float -> float`
logaritmo base e	`val log : float -> float`
logaritmo base 10	`val log10 : float -> float`
coseno	`val cos : float -> float`
seno	`val sin : float -> float`
tangente	`val tan : float -> float`
arc-coseno	`val acos : float -> float`
arc-seno	`val asin : float -> float`
arc-tangente	`val atan : float -> float`
etc...	

Consultar as bibliotecas standard para uma lista detalhada das funções predefinidas. Existem também várias bibliotecas fornecendo alternativas aos tipos `int` e `float`, como a biblioteca aritmética em precisão infinita.

3.3.2 Caracteres e strings

Os caracteres OCaml são valores ASCII delimitados por plicas. o carácter `b` em OCaml é : `'b'`.

As strings OCaml são cadeias de caracteres delimitadas por aspas e de comprimento conhecido (menor do que $2^{26} - 6$). a string *"Bom Dia"* em OCaml é: `''Bom Dia''`. a concatenação de `''Bom Dia''` com `'' Tudo Bem?''` é `''Bom Dia'' ^ '' Tudo Bem?''` o que resulta na string `''Bom Dia Tudo Bem?''`.

algumas funções sobre strings	
concatenação	`val (^) : string -> string -> string`
comprimento	`val length : string -> int`
n-ésimo caracter	`val get : string -> int -> char`
tudo em maiúsculas	`val uppercase : string -> string`
tudo em minúsculas	`val lowercase : string -> string`
etc...	

As parêntesis a volta do operador `^` significam que `^` é infixo. Consultar a biblioteca `string` para mais pormenores.

3.3.3 Booleanos

O tipo `bool` tem os dois elementos habituais `true` e `false`.

Operadores sobre booleanos	
negação	`val not : bool -> bool`
conjunção (e)	`val (&&) : bool -> bool -> bool`
disjunção (ou)	`val (||) : bool -> bool -> bool`
Operadores de comparação	
$=$	`val (=) : 'a -> 'a -> bool`
$\neq$	`val (<>) : 'a -> 'a -> bool`
$\leq$	`val (<=) : 'a -> 'a -> bool`
$<$	`val (<) : 'a -> 'a -> bool`
$\geq$	`val (>=) : 'a -> 'a -> bool`
$>$	`val (>) : 'a -> 'a -> bool`

3.3.4 Tipo *unidade*

Existe um tipo particular, chamado em OCaml `unit` que só possui um elemento: `()`.

```
# () ;;
- : unit = ()
```

Este tipo é particularmente importante quando se trata de escrever funções com efeitos colaterais. Aí o tipo `unit` e o valor `()` têm o papel do `void` do C

3.3.5 Produto Cartesiano, tuplos

O conjunto (tipo) $A_1 \times A_2 \times \ldots \times A_n$, formado a partir dos tipos A_1, A_2, ..., A_n é representado em OCaml por `A1*A2*A3*...*An`. Os elementos, tuplos, são representados por `(a1,a2,a3,...,an)`.

No caso de pares (de tipo, por exemplo, `A1*A2`) dispomos das funções de projecção `fst` e `snd`

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
# fst ( "Outubro", 12 ) ;;
- : string = "Outubro"
# snd ( "Outubro", 12 ) ;;
- : int = 12
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = 65, 'B', "ascii"
```

Uma característica interessante, que fazem dos tipos produto cartesiano tipos populares na programação OCaml, é a capacidade de serem calculados e devolvidos sem a necessidade de declarar o seu tipo previamente. Trata-se assim de uma facilidade muito cómoda que permite de agregar informação pontualmente e naturalmente (por exemplo aquando da definição de uma função que deve retornar vários valores, que podemos assim agrupar comodamente num tuplo).

Exemplos são precisamente o par e o triplo introduzidos na listagem anterior.

3.3.6 Listas

As listas são colecções polimórficas imutáveis e de acesso sequencial de elementos. Uma lista pode conter elementos de qualquer tipo, a restrição é que

todos os elementos contidos têm de ser do mesmo tipo (fala-se de colecções homogéneas). Podemos ter assim listas de inteiros, listas de caracteres. O tipo polimórfico OCaml correspondente é `'a list`. Assim o tipo das listas de inteiro é `int list`.

valores e operadores sobre listas	
lista vazia	`[]`
construtor de listas	`::`
concatenação de listas	`@`
listas por extensão	`[ 1 ; 2 ; 3 ]` ou `1 :: 2 :: 3 :: []`
comprimento	`val length : 'a list -> int`
cabeça duma lista	`val hd : 'a list -> 'a`
lista sem a cabeça	`val tl : 'a list -> 'a list`
n-ésimo elemento	`val nth : 'a list -> int -> 'a`
inversão de lista	`val rev : 'a list -> 'a list`
pertence	`val mem : 'a -> 'a list -> bool`
aplicação de uma função sobre elementos duma lista `map f [a1; ...; an] = [f a1; ...; f an]`	`val map : ('a -> 'b) -> 'a list -> 'b list`
`fold_left f a [b1;...; bn] =` `f (...(f (f a b1) b2)...) bn`	`val fold_left : ('a ->'b ->'a) ->'a ->'b list ->'a`
`for_all p [a1; ...; an] =` `(p a1) && (p a2) && ... && (p an)`	`val for_all : ('a -> bool) -> 'a list -> bool`
`exists p [a1; ...; an]=` `(p a1) || (p a2) || ... || (p an)`	`val exists : ('a -> bool) -> 'a list -> bool`
filtragem,`filter p [a1; ...; an]=` a lista de todos os elementos verificando `p`	`val filter : ('a -> bool) -> 'a list -> 'a list`
ordenação	`val sort : ('a -> 'a -> int) -> 'a list -> 'a list`

As listas são, em OCaml, definidas como um tipo soma (indutivo) polimórfica (sobre o tipo dos elementos que contém) da seguinte forma:

```
type 'a list = [] | :: of 'a * 'a list
```

Assim a lista vazia e o operador `::` são os construtores do tipo indutivo.

```
# [];;
- : 'a list = []
# [1;2;3];;
- : int list = [1; 2; 3]
# [1.5; 7.9; 9.3];;
- : float list = [1.5; 7.9; 9.3]
# ['O';'C'; 'a'; 'm'; 'l'];;
- : char list = ['O'; 'C'; 'a'; 'm'; 'l']
# [(1.5,'O'); (7.9, 'C'); (9.3,'a')];;
```

```
- : (float * char) list = [(1.5, 'O'); (7.9, 'C'); (9.3, 'a')]
# length [1.5; 7.9; 9.3];;
- : int = 3
# [1;2;3]@[1;2;3];;
- : int list = [1; 2; 3; 1; 2; 3]
# [1;2;3]@[1.5; 7.9; 9.3];;
Characters 9-12:
  [1;2;3]@[1.5; 7.9; 9.3];;
           ^^^
Error: This expression has type float but an expression was
       expected of type  int
# fold_left (fun a e -> a+e) 0 ([1;2;3]@[1;2;3]);;
- : int = 12
# sort compare [1;6;2;89;12;8;1;6;2;81];;
- : int list = [1; 1; 2; 2; 6; 6; 8; 12; 81; 89]
```

Consultar a biblioteca `list` para mais pormenores.

3.4 Estruturas Condicionais

Sem surpresas, o *se então senão* é *if then else* em OCaml.

Sintaxe:

`if expressão1 then expressão2 else expressão3`.

Restrição: `expressão2` e `expressão3` têm necessariamente o mesmo tipo. Este facto é devido ao facto de o if ser uma expressão como qualquer outra: devolve um valor. Logo os dois ramos do if devem ser do mesmo tipo. O equivalente na linguagem C é de facto a construção `(b)?t:e;`.

O `else` é obrigatório. Se for omitido é então acrescentado implicitamente `else ()` (significado: *senão não fazer nada*). Neste caso a expressão do `then` de ser também do tipo `unit`.

3.5 Declaração de valores

Já sabemos que toda a expressão OCaml tem um valor.

Se introduzirmos `4+6*2` num ficheiro OCaml então a expressão, aquando da sua execução, é avaliada (resultando em 16) e o valor é simplesmente devolvido.

```
# 4+6*2;;
- : int = 16
```

Se por ventura a expressão é novamente necessária, é preciso, nestes moldes, introduzir novamente a expressão e proceder novamente na sua avaliação.

Obviamente num contexto de programação a noção clássica de (declaração de) variável resolve precisamente esta situação.

Estas podem ser globais ou locais, mas serão sempre variáveis imutáveis.

3.5.1 Declarações globais

Sintaxe:

`let nome = expressão;;`

A semântica, sem surpresa, é: a `expressão` é avaliada e o tipo t da expressão é inferido. A seguir um espaço do tamanho dos valores do tipo t é alocado na memória. O valor calculado é colocado neste espaço e este será designado (acedido) via o identificado `nome`.

Esta atribuição é feita uma e uma só vez. Usar novamente o mecanismo de declaração não tem por efeito de alterar o valor da variável, mas sim gerar uma nova variável, com o mesmo nome, que se referirá a um novo espaço memória. As regras de visibilidade fazem o resto. A última declaração esconde as outras e será assim acedida.

```
# let x = 4;;
val x : int = 4
# print_int x;;
4- : unit = ()
# let x = 5;;
val x : int = 5
# print_int x;;
5- : unit = ()
# let x = 5.5;;
val x : float = 5.5
# print_int x;;
Characters 10-11:
  print_int x;;
             ^
Error: This expression has type float but an expression was
       expected of type  int
```

3.5.2 Declarações locais

Sintaxe:

`let nome = expressão1 in  expressão2 ;;`

Ao contrário da declaração global, o identificador `nome` só vai, neste caso, ser visível (e com o valor da `expressão1`) dentro de `expressão2`.

Existe a possibilidade de declarações anónimas. Tais são úteis quando só o valor computado na parte direita do igual interessa (no caso deste conter efeitos laterais por exemplo):

Sintaxe:

`let _ = expressão;;` ou `let _ = expressão1 in  expressão2 ;;`

Alguns exemplos:

```
# let x = 9;;
val x : int = 9
# let x = 2 + 4 in
  let y = x + 2 in
    y + 7;;
- : int = 15
# let x = 5 in
  (let x = x + 3 in x + 1) + ( let y = x + 3 in x + y);;
- : int = 22
```

Ambas as declarações locais e globais têm variantes, consultar as referências para mais pormenores

3.6 Expressões funcionais e funções

As funções são valores como quaisquer outros. Assim como `5` é um valor (que pode ser usado directamente como sub-expressão numa outra expressão, como por exemplo no caso parâmetro de uma chamada duma função), ou como `(if 3>4 then 5 else 8)` é o valor `8`, a função *identidade* ou a função *sucessor* são valores primitivos e podem ser usados como tal.

3.6.1 Declaração de base

Sintaxe:

`function p ->  expressão`

Neste caso a função contemplada é unária (um só parâmetro). É a sintaxe OCaml para o conhecido $\lambda x.expr$ do clculo λ. Este mecanismo introduz a noção de *função anónima* (no sentido de que representa um valor – uma função, sem nome).

Por exemplo:

```
# function x -> x;;
- : 'a -> 'a = <fun>
# function x -> x + 1;;
- : int -> int = <fun>
# function v -> if v > 10 then true else false;;
- : int -> bool = <fun>
# let x = 7;;
val x : int = 7
# if x > 5 then (function x -> x + 2) x else (function z -> z - 7) (x + 1);;
- : int = 9
```

Obviamente, estas definições não fogem à regra das outras expressões e, se não lhes forem atribuídas um nome, ficam utilizáveis uma só vez no momento da avaliação ficando assim perdidas "no limbo".

Para tal podemos usar o mecanismo de declaração global já introduzido

```
# let f = function  x -> x;;
val f : 'a -> 'a = <fun>
# f 7;;
- : int = 7
```

Um aspecto por realçar já nestes exemplos simples é a ausência da menção do valor de retorno, clássica no paradigma imperativo. Tal se deve, mais uma vez, ao facto de que funções são expressões. Avaliam-se em valores e são estes que constituem os valores de retorno.

```
# let u = function  x -> (if x>4 then (function v -> x+v) 6 else 3;;
val u : int -> int = <fun>
# u 7;;
- : int = 13
# u 2;;
- : int = 3
```

Para funções binárias (por exemplo) será preciso usar a sintaxe:

`function p ->  function q ->  expressão`

Apesar de parecer relativamente pesada, esta notação representa a forma mais primitiva de definição de função mas também introduz uma das mais interessantes perspectiva desta noção, quando comparada a noção de função no paradigma imperativo: a curryficação (currying, em inglês, em referência à Haskell Curry) e o seu pendente, a avaliação parcial.

Por exemplo a função `function (p,q,r) -> p * q +r` (um só parâmetro formal, um triplo `(p,q,r)`) na sua versão curryficada é

`function p -> function q -> function r -> p * q +r`

Esta versão denota uma função que espera um parâmetro inteiro p e que devolve uma função que por seu turno espera um inteiro q e devolve uma terceira função que finalmente do seu parâmetro r devolve $p * q + r$. Vantagens? vejamos o exemplo seguinte:

```
# let g = function (p,q,r) -> p * q + r;;
val g : int * int * int -> int = <fun>
# let h = function p -> function q -> function r -> p * q + r;;
val h : int -> int -> int -> int = <fun>
# g (1, 2, 3);;
- : int = 5
# h 1 2 3;;
- : int = 5
# h 1;;
- : int -> int -> int = <fun>
# h 1 2;;
- : int -> int = <fun>
# let a = h 1;;
val a : int -> int -> int = <fun>
# a 2 3;;
- : int = 5
```

Enquanto a avaliação de uma chamada a função `g` obriga à referência explícita de todos os seus parâmetros (ou ao contrário a ausência completa deles), a função h revela-se muito mais flexível. É possível fornecer parcialmente os seus parâmetros. Assim `(h 1 2)` é avaliada na função que espera um parâmetro r e que devolve então $1 * 2 + r$.

3.6.2 Sintaxe alternativa para funções n-árias

Por razões de conveniência é possível utilizar a sintaxe seguinte:

`fun p1 p2 ... pn -> expressão`

para as funções n-árias. Esta definição é o açúcar sintáctico para

`function p1 -> function p2 ->  ... -> function pn -> expressão`

3.6.3 Declarações de valores funcionais

Tal como qualquer valor, é possível declarar nomes para funções.

Sintaxes:

`let nome = function p ->  expressão`

ou ainda

`let nome = fun p1 p2 ... pn ->  expressão`

ou ainda

`let nome p1 p2 ... pn =  expressão`

No caso de uma função recursiva a sintaxe é:

`let rec nome p1 p2 ... pn =  expressão`

É aqui importante realçar a necessidade da palavra chave `rec`. Esta é realmente necessária (não pode ser omitida e assim o seu efeito inferido em tempo de execução), como mostra o exemplo seguinte.

```
# let fact n = n;;
val fact : 'a -> 'a = <fun>
# fact 4;;
- : int = 4
# let fact n = if n<=1 then 1 else n * fact (n - 1);;
val fact : int -> int = <fun>
# fact 4;;
- : int = 12
# let rec fact n =  if n<=1 then 1 else n * fact (n - 1);;
val fact : int -> int = <fun>
# fact 4;;
- : int = 24
```

Destaquemos finalmente o caso das funções sem retorno (procedimentos) ou sem argumentos. Estes dois casos são possíveis e expressam-se com o recurso ao tipo `uni`, porque afinal são expressões tipificadas como as restantes expressões OCaml.

```
# let x = 9;;
val x : int = 9
# let  f () = x + 2;;
val f : unit -> int = <fun>
# let g n = print_int (n + (f ()));;
val g : int -> unit = <fun>
# f();;
- : int = 11
# g 6;;
17- : unit = ()
```

3.6.4 Declaração de funções locais

Como para qualquer outra expressão, podemos declarar funções locais com o recurso, sem surpresa, da expressão `let..in`

```
# let f n =
  let g x = x + x*3 in
  let rec h x = if x > 0 then 2*x + x*3 else h (-x) in
    h (g n);;
val f : int -> int = <fun>
# f 6;;
- : int = 120
# g 7 ;;
Characters 0-1:
  g 7;;
  ^
Error: Unbound value g
```

3.6.5 Funções de ordem superior

Aquando da introdução da noção de curryficação introduziu-se sem mais justificações a noção de função que devolve uma função. Tal função é dita, de forma geral, de ordem superior.

A ordem de uma função qualifica a sua natureza relativamente ao tipo dos parâmetros de que necessita ou ao tipo do valor devolvido. De uma forma geral uma função que recebe parâmetros de ordem i e devolve um valor de ordem j é de ordem $max(i, j) + 1$.

No contexto do estudo das funções, uma função sem parâmetro é uma (função) constante ou de ordem 0, um valor inteiro (ou string, char etc ...) é também de ordem 0. Uma função que recebe como parâmetro um valor de ordem 0 e que devolve um valor de ordem 0 (as funções como habitualmente as entendemos) são de ordem 1 (primeira ordem). Uma função que recebe parâmetros de ordem 1 e devolve um valor de ordem 1 é de segunda ordem. Uma função que devolve (ou que aceita em parâmetro) uma função de primeira ordem é de segunda ordem etc ...

No limite, uma função que aceita ou devolve valores de ordem i qualquer que seja i é dita de ordem superior. Este é o caso mais geral e é um caso comum em programação funcional.

Imaginemos o caso simples da função identidade.

```
# let id = fun x -> x;;
val id : 'a -> 'a = <fun>
```

O facto de ser polimórfica sobre o seu argumento permite-lhe receber um argumento de qualquer tipo e devolvê-lo. Este pode ser uma função, um valor numérico etc... ou seja, de qualquer ordem. A função `id` é uma função polimórfica de ordem superior.

```
# id (fun x -> x + 1);;
- : int -> int = <fun>
# id 5;;
- : int = 5
```

Algumas Aplicações. As funções de ordem superior conseguem capturar de forma elegante padrões algorítmicos, tornando-as num dos mecanismos clássicos e elegantes para a modularidade e reutilização de código no paradigma funcional.

Por exemplo a função de ordem superior `fold_left`, definido no módulo `List` e cuja definição pode ser

```
#(*fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn. *)
let rec fold_left f a l =
  match l with
  | [] -> a
  | el::li -> fold_left f (f a el) li;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

captura a noção de calculo sobre uma sequência de valores (a lista) e cujo valor, inicializado, é acumulado até esgotar os elementos da sequência. O padrão algorítmico é, concretamente, o da varredura da esquerda para a direita dos elementos acompanhado da aplicação da função `f` sobre o elemento visitado.

Se ligarmos este conceito à álgebra da programação, o mecanismo implementado pela função `fold_left` é um caso particular de *catamorfismo*.

Em termos de comparação com o estilo imperativo, trata-se simplesmente do padrão algorítmico seguinte (em pseudo-código):

```
v := a;
for i:=0 to (length l) - 1 do
v := f v (nth i l);
done;
return v;
```

Visualmente podemos resumir este padrão com a ajuda alternativa das figuras 1 e 2.

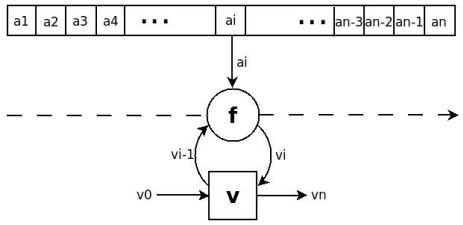


Figura 1: fold_left, vizualmente - versão mecânica

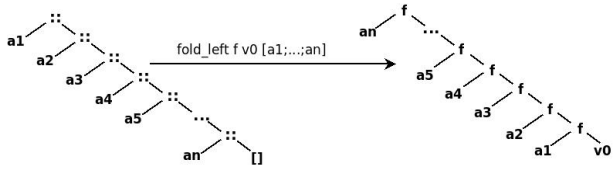


Figura 2: fold_left, vizualmente - versão algébrica

Na prática esta função pode ser usada para calcular valores que envolvem potencialmente todos os elementos da lista. Por exemplo,

```
# let somal l = fold_left (fun a e -> a + e) 0 l;;
val somal : int list -> int = <fun>
# somal [1;2;3;4;5;6;7;8;9];;
- : int = 45
# let inverte l = fold_left (fun a e -> e::a) [] l;;
val inverte : 'a list -> 'a list = <fun>
# inverte [1;2;3;4;5;6;7;8;9];;
- : int list = [9; 8; 7; 6; 5; 4; 3; 2; 1]
# let filtra pred l =
 fold_left
        (fun (a,b) e -> if (pred e) then (e::a,b) else (a,e::b))
        ([],[]) l;;
  val filtra : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
# filtra (fun i -> i mod 2 = 0)  [1;2;3;4;5;6;7;8;9];;
- : int list * int list = ([8; 6; 4; 2], [9; 7; 5; 3; 1])
# let existe pred l =
     fold_left (fun a e -> if pred e then true else a ) false l;;
val existe : ('a -> bool) -> 'a list -> bool = <fun>
# existe (fun i -> (i mod 2 = 0) && (i mod 3 = 0)) [1;2;3;4;5;6;7;8;9];;
- : bool = true
```

Funções como `fold_left` designam-se de classicamente de *combinadores*. Notemos igualmente que as funções `inverte`, `filtra` e `existe` são igualmente combinadores e constam da biblioteca standard no módulo `List` sob os nomes, respectivamente, `rev`, `partition` e `exists`.

Um combinador muito utilizado no contexto da manipulação de estruturas de dados da categoria dos contentores (como as listas, ou vectores ou ainda árvores) é a função `map`. Graficamente, o padrão algorítmico que implementa é representado pela figura 3.

```
# map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fun a -> a + 2)  [1;2;3;4;5;6;7;8;9];;
- : int list = [3; 4; 5; 6; 7; 8; 9; 10; 11]
# map (fun  i -> (i mod 2 = 0) && (i mod 3 = 0))  [1;2;3;4;5;6;7;8;9];;
- : bool list =
[false; false; false; false; false; true; false; false; false]
```

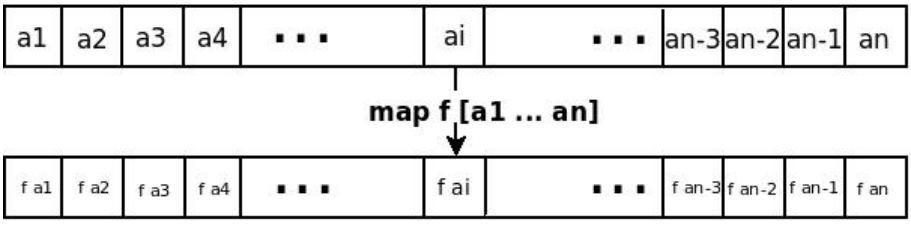


Figura 3: map, vizualmente

Alias, é bastante simples reformular a função `map` a custa da função `fold_left`.

```
# let mapa f l =  rev (fold_left (fun a e -> (f e)::a) [] l);;
val mapa : ('a -> 'b) -> 'a list -> 'b list = <fun>
# mapa (fun  i -> (i mod 2 = 0) && (i mod 3 = 0))  [1;2;3;4;5;6;7;8;9];;
- : bool list =
[false; false; false; false; false; true; false; false; false]
```

Outro exemplo pode ser a noção de composição de funções

```
# let ( *+) f g =  fun x -> f (g x);;
val ( *+ ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let f = (fun x -> 2 * x) *+ (fun x -> x + 3);;
val f : int -> int = <fun>
# f 6;;
- : int = 18
```

Para terminar, ilustremos a noção de construção incremental de funções. Imaginemos que pretendemos construir uma função a medida que se fica a conhecer informação sobre o seu domínio e co-domínio (i.e. os dados que a definam).

```
# let f x = raise Not_found;;
val f : 'a -> 'b = <fun>
# f 5;;
Exception: Not_found.
# let f x = if x = 5 then 7 else f x;;
val f : int -> int = <fun>
```

```
# f 5;;
- : int = 7
# f 3;;
Exception: Not_found.
# let f x = if x = 3 then 1 else f x;;
val f : int -> int = <fun>
# f 5;;
- : int = 7
# f 3;;
- : int = 1
# let f x = if x = 5 then 2 else f x;;
val f : int -> int = <fun>
# f 5;;
- : int = 2
# f 3;;
- : int = 1
# f 2;;
Exception: Not_found.
```

Neste exemplo, a medida que conhecemos os pares (*valor de entrada*, *valor devolvido*) sobrecarregamos (à custa da noção de fecho e de porte) a função de base `let f x = raise Not_found;;` que representação a função cujo domínio é vazio.

Este exemplo exemplifica uma forma, que pode parecer original para quem descobre a noção de função de ordem superior, de encarar as funções. Estas podem ser vistas como mapas (*maps* em inglês) ou suporte a estruturas de tipo tabela de *hash* sem colisões (repare como o par $5 \mapsto 7$ foi substituído pelo par $5 \mapsto 2$).

3.6.6 Fecho

Em OCaml as funções são fechadas. Isto é, todas as variáveis contidas na definição e que não são parâmetros são avaliadas e substituídas pelos seus valores por forma a permitir uma definição da função que tire proveito mas que fique independente do ambiente global no qual foi definida.

Por exemplo:

```
# let m = 3 ;;
val m : int = 3
```

```
# let f= (function x -> x + m);;
val f : int -> int = <fun>
# f 5;;
- : int = 8
# let m = 8;;
val m : int = 8
# f 5;;
- : int = 8
```

Este exemplo põe em evidência como são de facto consideradas e manipuladas as funções em OCaml: uma função é definida pelo seu código e pelo ambiente ao qual ela tem acesso. Este ambiente faz parte integrante da sua definição. Fala-se assim, para este par de informação (ambiente e código) de fecho de uma função.

Diferenças com os apontadores de função (à la C). A funções de ordem superior são mais gerais do que as funções que usam o mecanismo dos apontadores de função.

Para a maioria dos casos (simples), de facto o mecanismo de apontadores de função, como o encontramos na linguagem C, permite uma expressão equivalente do conceito de função de ordem superior.

Mas, se misturarmos os fenómenos de valores locais, fecho e avaliação parcial, a situação muda completamente.

Por exemplo

```
# let f x = let x2 =  2 * x + x * x in function y -> x2 + y;;
val f : int -> int -> int = <fun>
# let g x = f x;;
val g : int -> int -> int = <fun>
# let g1 = g 3;;
val g1 : int -> int = <fun>
# let g2 = g 4;;
val g2 : int -> int = <fun>
# g1 2;;
- : int = 17
# g2 2;;
- : int = 26
```

De facto cada instanciação parcial de `f` permite a redução da expressão local `x2` para um valor particular, diferente em cada uma delas. A função unária resultante tem então no seu fecho uma cópia do valor de `x2`. Este fenómeno de fecho não tem equivalente no uso dos apontadores de função. Para disponibilizar de tal comportamento numa linguagem imperativa, será necessária codificar e manipular explicitamente a noção de fecho de uma função (a função e os dados a que ela tem acesso para a sua adequada execução).

3.6.7 Ordem Superior e Polimorfismo

O polimorfismo paramétrico do OCaml segue a seguinte política: *para todos ou para um*. Assim a função `id` que é de tipo `'a -> 'a` aceita qualquer parâmetro, e devolve na idêntica este. Neste caso a variável de tipo `'a` sofre uma instanciação para o tipo do seu parâmetro. Neste sentido a função *id* funciona *para todos*. No entanto, após instanciada, a variável de tipo `'a` manterá o seu vínculo. `let id_int = function (x:int) -> id x` é de tipo `int -> int` e só funcionará *doravante* como a função identidade para inteiro, exclusivamente – ou seja *para um*. Outro exemplo pode ser o valor `[]` que na sua generalidade é de tipo `'a list`, enquanto o `[]` da lista `1::2::3::[]` é `int list`.

```
# tl [1];;
- : int list = []
# [];;
- : 'a list = []
# tl [1] = tl ['a'];;
Characters 9-17:
  tl [1] = tl ['a'];;
           ^^^^^^^^
Error: This expression has type char list
       but an expression was expected of type int list
```

A tipagem de OCaml é estática, ou seja, é exclusivamente realizada em tempo de compilação. O polimorfismo paramétrico na sua generalidade pode levar a situações em que é impossível inferir o tipo (mais geral) de um valor e neste momento incorrecto atribui-lhe um tipo sem alguma cautela. Para tal existe a noção de *polimorfismo fraco*.

Veremos ilustrações deste mesmo fenómenos mais adiante com a noção de referências. Concentremo-nos aqui num exemplo intrigante com funções.

Misturar polimorfismo, ordem superior e tipos funcionais (tipos que representam funções) leva-nos a algumas situações limites na inferência de tipos que o sistema de tipos de OCaml detecta perfeitamente e trata com o devido cuidado (de uma forma diferente ao que esperaríamos num contexto em que não haveria tipos funcionais envolvidos).

```
# let f = fun x -> x;;
val f : 'a -> 'a = <fun>
# f f;;
- : '_a -> '_a = <fun>
# f (f f);;
- : '_a -> '_a = <fun>
# let g = f (f f);;
val g : '_a -> '_a = <fun>
# g 6 ;;
- : int = 6
# g;;
- : int -> int = <fun>
```

A variável de tipo `'_a` é uma variável de *tipo fraco*. Ainda não é conhecida o seu valor, mas o resto do programa poderá permitir ganhar este conhecimento. E nesta altura será inequivocamente instanciada. É o que demonstra o exemplo da função `g`. A função `g` tem o tipo `'_a -> '_a` aquando da sua definição. O que significa que no momento da sua definição o tipo (mais geral) de `g` ainda não é conhecido (mas já se sabe que é uma função de um tipo para ele próprio) e que a decisão final é adiada. Quando esta decisão é conhecida (afinal `'_a` é `int`), então é aplicada *retro-activamente*.

3.7 Uma digressão sobre a noção de igualdade

Notemos que em OCaml existem dois símbolos para a igualdade. o predicado `=` que permite expressar a igualdade (estrutural) dos valores comparados, e `==` que representa a igualdade física, ou seja que testa se dois objectos representam o mesmo espaço memória.

Ambas são polimórficas

```
# (==);;
- : 'a -> 'a -> bool = <fun>
# (=);;
```

```
- : 'a -> 'a -> bool = <fun>
```

No entanto, e à semelhança da considerações de tipagem no contexto de ordem superior e de tipos funcionais, o suporte às funções obrigam alguma cautela.

```
# 'a' = 'b';;
- : bool = false
# 1 = 1;;
- : bool = true
# "ola" == "ola";;
- : bool = false
#  "ola" = "ola";;
- : bool = true
# [1;2;3]=[3;2;1];;
- : bool = false
# [1;2;3] = [1;2;3];;
- : bool = true
#  [1;2;3] == [1;2;3];;
- : bool = false
# (function x -> x + 5) == (function y -> y + 5);;
- : bool = false
# (function x -> x + 5) = (function y -> y + 5);;
  Exception: Invalid_argument "equal: functional value".
#
let f = (function x -> x + 5);;
val f : int -> int = <fun>
#   f == f;;
- : bool = true
# f = f;;
Exception: Invalid_argument "equal: functional value".
```

3.8 Declaração de tipos e filtragem de motivos

3.8.1 Pattern matching (filtragem)

A filtragem de motivos, ou concordância de padrão, permite uma programação baseada em analise de casos.

Sintaxe:

```
match expr with
| p1 -> expr1
.
.
.
| pn -> expr
```

Significa: olhar para o valor de `expr`, se este valor se assemelha com `p1` então avaliar `expr1` senão verificar semelhança com `p2`, etc...

Considerações:

- os motivos `p1...pn` têm de cobrir todos os casos;
- os motivos têm de ser lineares. Isto é, se conterem variáveis, então essas só ocorrem uma única vez.
- quando dois motivos cobrem o mesmo caso (isto é, quando existe uma sobreposição entre dois motivos e que o caso avaliado é coberto por ambos), é então escolhido o motivo que aparece primeiro.

Motivo universal (que se assemelha sempre) é denotado pelo carácter `_`. O motivo `_` significa "outros casos" ou ainda "tudo". A construção `let` pode ser vista como um caso particular de filtragem quando existe uma só possibilidade de concordância.

As construções de concordância de padrão, e os próprios padrões podem ser aninhados.

Exemplos de filtragem:

```
# let rec fold_left f a l =
  match l with
    [] -> a
  | cabeca::resto -> fold_left f (f a cabeca) resto ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [8;4;10];;
- : int = 22
# let vazio l =
  match l with
      [] -> true
    | _ -> false;;
      val vazio : 'a list -> bool = <fun>
```

```
# vazio [1;2;3];;
- : bool = false
# let soma_par c =
  let (esq,dir) = c in
    esq+dir;;
  val soma_par : int * int -> int = <fun>
# soma_par (1,2);;
- : int = 3
# let has_two_elements l =
   match l with
    [] -> false (*|l| = 0*)
   | el::li -> match li with
                 [] -> false (*|l|=1*)
               | el2 :: lli -> match lli with
                                 [] -> true (* |l| = 2 *)
                               | el3 :: _ -> false;;
             val has_two_elements : 'a list -> bool = <fun>
# let has_two_elements l =
   match l with
   | [el1;el2] -> true
   | _ -> false;;
      val has_two_elements : 'a list -> bool = <fun>
# has_two_elements [1;2];;
- : bool = true
# has_two_elements [1;2;4];;
- : bool = false
```

Existem extensões interessantes aos mecanismos de concordância aqui apresentados, para conhecê-los consultar a bibliografia.

3.8.2 Declaração de tipos

É possível estender os tipos de base com tipos definidos pelo programador.

Sintaxe:

```
type nome = definicao_do_tipo;;
```

No caso de tipos mutuamente recursivos (que dependem mutuamente uns dos outros) a sintaxe é estendida em:

```
type nome1 = definicao_do_tipo1
and nome2 = definicao_do_tipo2
.
.
.

and nome_n = definicao_do_tipo_n;;
```

No caso dos tipos serem polimórficos, é necessário indicar as variáveis de tipo.

Sintaxe:

```
type ('a1,...,'an) nome = definição_do_tipo;;
```

Obviamente, podemos definir tipos polimórficos mutuamente recursivos.

Quais são as possibilidades para `definição_do_tipo`? composição de tipos de base (como por exemplo pares de listas de inteiros, etc...), registos, tipos soma.

Exemplos:

```
# type 'param par_inteiro = int * 'param ;;
type 'a par_inteiro = int * 'a
# type par_totalmente_instanciado = float par_inteiro ;;
type par_totalmente_instanciado = float par_inteiro
# let (x:par_totalmente_instanciado) = (3, 3.14) ;;
val x : par_totalmente_instanciado = 3, 3.14
```

3.8.3 Registos

Sintaxe:

```
type ('a1,...,'an) nome =
    { nome_campo1 : tipo1; ...; nome_campo_n : tipo_n };;
```

Um exemplo:

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
```

Criação de registos Sintaxe:

```
{ nome_campo1 = valor1; ...; nome_campo_n = valor_n };;
```

Não é necessário listar os campos na ordem da definição, mas é preciso que todos sejam referidos.

```
# let c = {re=2;im=3};;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

Acesso aos campos Sintaxe:

```
expr.nome_campo
```

Consulte a bibliografia para mais pormenores.

3.8.4 Tipos soma

Estes tipos codificam o que usualmente referimos por tipos ou conjuntos indutivos.

Sintaxe:

```
type    nome_tipo =
        | Nome_construtor_1 of t11 * ...* t1k.
        .
        .
        .
        | Nome_construtor_n of tn1 * ...* tnk
```

Os identificadores de construtores começam sempre por uma maiúscula. A parte `of t1 * ...* tk` na declaração dum construtor não é obrigatória. Só é requerida se o construtor tiver argumentos.

Exemplos de tipo soma:

```
# type natural = Z | S of natural;;
type natural = Z | S of natural
# let rec (+.+) x = function Z -> x | S p -> S (x +.+ p);;
val ( +.+ ) : natural -> natural -> natural = <fun>
# S Z +.+ S (S Z);;
```

```
- : natural = S (S (S Z))
# type ('a, 'b) list2 =
       Nil
    |  Acons of  'a * ('a, 'b) list2
    |  Bcons of  'b * ('a, 'b) list2 ;;
# type semana =
Domingo | Segunda | Terca | Quarta | Quinta | Sexta | Sabado;;
# type  tarefas =
    Chemistry | Companhia  | Trabalho_de_Casa
   | Armazem | Faculdade_da_Cerveja;;
# type actividade =
Trabalho of tarefa | Compras | Descanco;;
```

e exemplos de função com filtragem sobre elementos de tipos soma

```
# let rec extract_odd l =
match l with
     Nil -> []
   | Acons(_, x) -> extract_odd x
   | Bcons(n, x) -> n::(extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
# let que_faco_hoje dia =
  match dia with
      Segunda-> Trabalho Faculdade_da_Cerveja
    | Terca -> Trabalho Companhia
    | Quarta -> Trabalho Chemistry
    | Quinta -> Trabalho Trabalho_de_Casa
    | Sexta -> Trabalho Armazem
    | Sabado -> Compras
    | Domingo-> Descanco;;
val que_faco_hoje : semana -> actividade = <fun>
# que_faco_hoje Quinta;;
- : actividade = Trabalho Trabalho_de_Casa
```

3.9 Excepções

3.9.1 Definição

Sintaxe:

```
exception Nome of parâmetros;;
```

Regras:

- os identificadores de excepções são de facto construtores, logo devem começar por uma maiúscula;
- o polimorfismo não é aceite nas excepções.

```
# exception MINHA_EXCEPCAO;;
exception MINHA_EXCEPCAO
# MINHA_EXCEPCAO;;
- : exn = MINHA_EXCEPCAO
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth(4)
```

3.9.2 Lançar uma excepção

Sintaxe:

```
raise (Nome t1 ... tn)
```

Exemplos:

```
# raise ;;
- : exn -> 'a = <fun>
# raise MINHA_EXCEPCAO;;
Uncaught exception: MINHA_EXCEPCAO
# 1+(raise MINHA_EXCEPCAO);;
Uncaught exception: MINHA_EXCEPCAO
# raise (Depth 4);;
Uncaught exception: Depth(4)
```

3.9.3 Recuperar excepções

Sintaxe:

```
try expressão with
| motivo_excepcao1 -> expr1
.
.
.
| motivo_excepcao_n -> expr_n
```

No caso da avaliação de `expressão` levantar uma excepção então há filtragem com os diferentes motivos listados. no caso de haver concordância a expressão correspondente é avaliada.

Exemplos:

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
     [] -> 1
   | 0 :: _ -> raise Found_zero
   | n :: x -> n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
    try mult_rec l with Found_zero -> 0 ;;
val mult_list : int list -> int = <fun>
# mult_list [1;2;3;0;5;6] ;;
- : int = 0
```

4 Programação imperativa

4.1 Estruturas de Dados modificáveis

4.1.1 Vectores

Os vectores em OCaml têm por delimitadores os símbolos `[|` e `|]`.

Exemplo:

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

Acesso Sintaxe:

```
expr1 . ( expr2 )
```

Atribuição Sintaxe:

```
expr1 . ( expr2 ) <- expr3
```

Exemplos:

```
# v.(1) ;;
- : float = 6.28
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 9.42|]
# let t = [|
            [|1|];
            [|1; 1|];
            [|1; 2; 1|];
            [|1; 3; 3; 1|];
            [|1; 4; 6; 4; 1|];
            [|1; 5; 10; 10; 5; 1|]
          |] ;;
val t : int array array =
  [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|];
    [|1; 4; 6; 4; ...|]; ...|]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

Operadores usuais sobre vectores	
Comprimento	`val length : 'a array -> int`
criação `make n x = [|x; ...(n vezes)... ;x|]`	`val make : int -> 'a -> 'a array`
inicialização `init n f = [|f 1; ... ;f n|]`	`val init : int -> (int -> 'a) -> 'a array`
concatenação de vectores	`val append : 'a array -> 'a array -> 'a array`
duplicação de vectores	`val copy : 'a array -> 'a array`
Conversões de/para listas	`val to_list : 'a array -> 'a list` `val of_list : 'a list -> 'a array`
`map` para vectores	`val map : ('a -> 'b) -> 'a array -> 'b array`
`fold_left` para vectores	`val fold_left :` `('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`
ordenação	`val sort : ('a -> 'a -> int) -> 'a array -> unit`

Consultar a biblioteca `array` para mais pormenores.

4.1.2 Strings como vectores

Acesso directo Sintaxe: `expr1 . [expr2]`

Atribuição Sintaxe: `expr1 . [expr2] <- expr3`
Exemplos

```
# let s = "Ola Tudo Bem?";;
val s : string = "Ola Tudo Bem?"
# s.[2];;
- : char = 'a'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "OlZ Tudo Bem?"
```

4.1.3 Registos e campos modificáveis

O OCaml permite que campos de registos possam ser modificáveis. Para tal é preciso que os referidos campos sejam declarados com a palavra chave `mutable`.

Declaração Sintaxe: `type nom = { ...;  mutable nome_campo_i : t ; ...}`

Atribuição Sintaxe: `expr1 . nom <- expr2`
Exemplos:

```
# type ponto = { mutable xc : float; mutable yc : float } ;;
type ponto = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : ponto = {xc=1; yc=0}
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : ponto = {xc=3; yc=0}
# let moveto p dx dy =
   let () = p.xc <- p.xc +. dx
   in p.yc <- p.yc +. dy ;;
val moveto : ponto -> float -> float -> unit = <fun>
```

```
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : ponto = {xc=4.1; yc=2.2}
```

4.1.4 Referências

O conceito de variáveis OCaml difere do conceito de variáveis de linguagens s como o C. De facto assemelham-se do conceito de variáveis em matemática: *uma variável, um valor.*

Existe no entanto a possibilidade de ter variáveis como em C. Estas são consideradas como referências para valores e devem ser declaradas como sendo do tipo `ref`.

Definição Na sua base o tipo ref é definido da seguinte forma:

```
type 'a ref = {mutable contents:'a}
```

Declaração Sintaxe: `ref expr`

Acesso Sintaxe: `!nome`

Atribuição Sintaxe: `nome:= expr`

Exemplos:

```
# let x = ref 3 ;;
val x : int ref = {contents=3}
# x ;;
- : int ref = {contents=3}
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5
```

Fecho, referências e funções É possível tirar proveito do fenómeno de fecho para obter funções que partilham e manipulam valores escondidos ao utilizador (fala-se de encapsulamento). A ideia e declarar tais funções e as variáveis por esconder simultaneamente e devolver simplesmente as funções por disponibilizar ao programador. De facto, no fecho de cada função devolvida estará uma referência para os valores locais escondidos (inacessíveis) ao programador.

Eis um exemplo:

```
# let (inicializa, afecta, incrementa, mostra) =
 let valor_por_esconder : int ref =
    ref 0 in
   let elt_init () =
    valor_por_esconder :=  0 in
   let elt_get () =
    !valor_por_esconder in
   let elt_incr () =
    valor_por_esconder := 1+ !valor_por_esconder in
   let elt_set n =
    valor_por_esconder := n in
   ( elt_init,elt_set,elt_incr,elt_get);;
val inicializa : unit -> unit = <fun>
val afecta : int -> unit = <fun>
val incrementa : unit -> unit = <fun>
val mostra : unit -> int = <fun>
# inicializa();;
- : unit = ()
# mostra();;
- : int = 0
# incrementa();;
- : unit = ()
# mostra();;
- : int = 1
# afecta 5;;
- : unit = ()
# mostra();;
- : int = 5
# valor_por_esconder;;
```

```
Unbound value valor_por_esconder
```

4.2 Entradas e saídas

As entradas e saídas, de forma geral, são efectuadas por funções que não calculam nenhum valor em particular mas que efectuam efeitos colaterais. Assim uma função de entradas e saídas devolverá habitualmente um elemento do tipo `unit`.

As entradas e saídas em OCaml são geridas a custa de *canais de comunicação*. Existem dois tipos predefinidos de canais: o tipo dos canais de entrada `in_channel` e o tipo dos canais de saída `out_channel`. Quando o fim de ficheiro é atingido a excepção `End_of_file` é lançada. Existem três canais predefinidos, a semelhança do C: `stdin`, `stdout` e `stderr`.

4.2.1 Abertura de ficheiro

```
# open_in;;
- : string -> in_channel = <fun>
# open_out;;
- : string -> out_channel = <fun>
```

4.2.2 Fecho de ficheiro

```
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```

4.2.3 Leitura e Escrita

Eis algumas funções de leitura e escrita em canais abertos:

```
# input_line ;;
- : in_channel -> string = <fun>
# input ;;
- : in_channel -> string -> int -> int -> int = <fun>
# output ;;
- : out_channel -> string -> int -> int -> unit = <fun>
# read_line ;;
```

```
- : unit -> string = <fun>
# read_int ;;
- : unit -> int = <fun>
# read_float ;;
- : unit -> float = <fun>
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
# print_endline ;;
- : string -> unit = <fun>
```

Exemplo:

```
# let () = print_string "um," in
  let () = print_string " dois," in
  let () = print_string " e tres" in
   let () = print_string " zero" in
    print_newline ();;
um, dois e tres zero
- : unit = ()
```

As funções da família `scanf` e `printf` familiares à programação existem numa declinação particular em OCaml. Ver as bibliotecas associadas.

4.3 Estruturas de controlo

4.3.1 Sequência

É possível pedir ao OCaml que uma lista de expressões sejam avaliados sequencialmente, tal como aconteceria em C. Tal sequência segue o seguinte formato: `expr1;expr2; ...;expr_n`. É importante perceber que esta não foge às regras da programação funcional. Uma sequência `expr1;expr2; ...;expr_n` é uma e uma só expressão como tal tem um valor.

O valor da sequência é o valor da ultima expressão (neste caso `expr_n`). O tipo das expressões `expr1;expr2;...;expr_(n-1)` deve ser `unit`. Se tal não for o caso, uma mensagem de aviso (Warning) é mostrada.

A função `val ignore: 'a -> unit` permite remediar este problema.

Ao permitir mecanismos tradicionalmente imperativos, mesmo de forma controlada como é o caso aqui, é necessário disponibilizar (também de forma controlada) a sequência para poder permitir o encadeamento de efeitos colaterais. A sequência é o mecanismo para este efeito.

Exemplos:

```
# print_int 1; 2 ; 3;;
Characters 13-14:
  print_int 1; 2 ; 3;;
               ^
Warning S: this expression should have type unit.
1- : int = 3
# print_int 1; ignore 2; 3 ;;
1- : int = 3
# print_string ("1+1 = "^string_of_int (1+1));print_newline (); 4+1 ;;
1+1 = 2
- : int = 5
```

4.3.2 Blocos de programa

Para delimitar grupos de expressões podemos utilizar alternativamente as construções seguintes:

```
( expr )
```

ou

```
begin expr end
```

4.3.3 Ciclos

Sintaxe:

```
for nome = expr1 to  expr2 do expr3 done
for nome = expr1 downto expr2 do expr3 done
while expr1 do expr2 done
```

Exemplos:

```
# let rec emais_emenos n =
   print_string "Introduza um numero : ";
   let i = read_int () in
   if i = n then print_string "BRAVO\n\n"
    else
      begin
        if i < n
          then print_string "E+\n" else print_string "E-\n";
               emais_emenos n
      end ;;
val emais_emenos : int -> unit = <fun>
# for i=10 downto 1 do
     print_int i; print_string " "
  done;
  print_newline() ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
# let r = ref 1
 in while !r < 11  do
      print_int !r ;
      print_string " " ;
      r := !r+1
    done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

4.3.4 Tipagem, aplicação parcial e fechos

Voltemos agora ao conceito de tipagem fraca, mas desta vez na presença de referências.

Imaginemos que queremos definir e utilizar uma referência para uma lista. A sua definição poderá ser

```
# let rl = ref [];;
val rl : '_a list ref = {contents = []}
# let reset () = rl :=  [];;
val reset : unit -> unit = <fun>
```

Nesta situação, é de facto impossível prever o tipo dos elementos da lista

referenciada. Tal poderá acontecer quando se juntará o primeiro elemento a lista referenciada por `rl`.

Imaginemos que estamos numa situação em que não haja o mecanismo dos tipos fracos. Após a primeira introdução de elemento (digamos um inteiro, como em `rl := 1 ::`rl!), então a lista referenciada passa a ser de tipo `int list` (ou seja `rl` teria agora por tipo `int list ref`).

Se utilizarmos a seguir a função reset, a referência `rl` apontaria para uma lista totalmente nova, vazia, logo com a variável de tipo por instanciar (de tipo `'a list`).

Nesta situação seria totalmente viável juntar um caracter à lista (`rl := 'a' ::`rl!) e assim instanciar o tipo da lista referenciada para `char list`.

Neste caso `rl` teria agora por tipo `char list ref` ou que introduziria uma inconsistência no sistema de tipo: a referência `rl` não pode ser de dois tipos fechados (fechados = completamente instanciados) diferentes.

O tipo de `val rl : '_a list ref = {contents = []}` permite evitar exactamente esta situação.

```
#  let rl = ref [];;
val rl : '_a list ref = {contents = []}
# let reset () = rl :=  [];;
val reset : unit -> unit = <fun>
# rl := 1 :: !rl;;
- : unit = ()
# rl ;;
- : int list ref = {contents = [1]}
# reset ();;
- : unit = ()
# rl ;;
- : int list ref = {contents = []}
# rl := 'a' :: !rl;;
Error: This expression has type char but an expression was expected of type
         int
```

Para terminar, vejamos porque os traços imperativos devem ser misturados com processos funcionais com muita cautela.

```
# let f x = rl := x :: !rl; fun y -> y + 1;;
```

```
val f : int -> int -> int = <fun>
# f 5 2;;
- : int = 3
# rl ;;
- : int list ref = {contents = [5]}
# f 6 4;;
- : int = 5
# rl;;
- : int list ref = {contents = [6; 5]}
# let g x = f x;;
val g : int -> int -> int = <fun>
# g 8;;
- : int -> int = <fun>
# rl;;
- : int list ref = {contents = [8; 6; 5]}
# let h = f 9;;
val h : int -> int = <fun>
# rl;;
- : int list ref = {contents = [9; 8; 6; 5]}
# h 7;;
- : int = 8
# rl;;
- : int list ref = {contents = [9; 8; 6; 5]}
# h 10;;
- : int = 11
# rl;;
- : int list ref = {contents = [9; 8; 6; 5]}
```

Como podemos ver no exemplo anterior, cada aplicação completa de `f` permite, para além de devolver o valor inteiro desejado, a execução do efeito colateral declarada no corpo da função. A aplicação parcial de `f` resultante na função `h` faz com que a execução do efeito colateral seja efectuada uma só vez (no momento da definição de `h`), ficando assim separada do calculo do valor inteiro consecutivo. A avaliação parcial quando combinada com efeitos colaterais pode dar resultados surpreendentes à primeira vista.

Outra manifestação dos efeitos subtis da avaliação quando combinada com efeitos colaterais é o efeito da ordem de avaliação das expressões.

```
#print_int 1; print_int 2; print_int 3; print_int 4 ;;
```

```
1234- : unit = ()
# let a = (print_int 1; 1) * (print_int 2; 2) in
  let b = (print_int 3; 3) * (print_int 4; 4) in
  a + b;;
2143- : int = 14
# let b = (print_int 3; 3) * (print_int 4; 4) in
  let a = (print_int 1; 1) * (print_int 2; 2) in
  a + b;;
4321- : int = 14
# (print_int 1; 1) * (print_int 2; 2) +
  (print_int 3; 3) * (print_int 4; 4);;
4321- : int = 14
```

Conclusão, a ordem de avaliação de expressões não é especificada nos documento de referência de OCaml e resulta numa escolha de desenho do compilador utilizado. O programador deverá evitar assim que os seus programas dependem desta ordem de avaliação, ou então conhecer bem o compilador usado.

5 Avaliação Ansiosa, Avaliação preguiçosa e estruturas de dados infinitas

Uma característica fundamental das linguagens de programação é a estratégia de avaliação. É tão fundamental que molda por completo o desenho da linguagem em causa e até mesmo a forma de programar na dita.

Não vamos nesta introdução explicitar toda a envolvente, deveras muito instrutiva. Aconselhamos o leitor curioso em consultar os manuais clássicos de semântica das linguagens de programação, ou os tratados clássicos sobre o cálculo lambda.

O conceito de estratégia da avaliação resume-se à escolha de desenho feita pelo *designer* da linguagem sobre a forma de calcular (avaliar) uma chamada a uma função. Começamos por avaliar os parâmetros ou aguardamos que estes sejam necessários na aplicação da função?.

Por exemplo, para obter o valor de `Ackermann (x + 2) (fact 4)`, optamos por dar prioridade à função ackermann ou aos seus parâmetros?

A primeira escolha nos daria uma redução (i.e. um passo de cálculo) para:

```
if (x + 2) = 0
  then (fact 4) + 1
  else  if (x + 2) >= 0 && (fact 4) = 0
        then  ackermann ((x + 2) - 1) 1
        else  ackermann ((x + 2) - 1) (ackermann (x + 2) ((fact 4) - 1))
```

enquanto a segunda nos levaria à avaliação de `ackermann x+2 24`.

A primeira escolha é designada por avaliação preguiçosa (*lazy evaluation*; no sentido seguinte: avaliamos uma expressão só quando o seu uso se revela necessário)

A segunda escolha é designada de avaliação ansiosa (*eager evaluation*).

No que diz respeito a avaliação das chamadas à funções, o desenho da linguagem OCaml repousa, à semelhança de uma vasta gama de linguagens de programação, sobre esta última avaliação. A avaliação ansiosa é mecanismo de avaliação por defeito (i.e. implícito). Veremos mas adiante que podemos no entanto optar explicitamente pela primeira.

Assim antes de explicar mais em detalhe o mecanismo de avaliação em OCaml, citemos um exemplo famoso que abona na avaliação preguiçosa (que no entanto tem também os seus desafios por resolver).

```
# let ciclo n = while true do () done; n;;
val ciclo : 'a -> 'a = <fun>
# let tricky n m = n + 1;;
val tricky : int -> 'a -> int = <fun>
#(* Não tente isso em casa, esta avaliação
    é feita por especialistas treinados*)
 tricky (5 + 2) (ciclo 7);;
```

Numa linguagem, como Haskell, onde a avaliação é por defeito preguiçosa o resultado teria sido `tricky (5 + 2) (ciclo 7) ->  (5 + 2) + 1 -> 8`.

5.1 Avaliação em OCaml

Comecemos por relembrar as regras que oportunamente ao longo deste documento foram dadas sobre a ordem de avaliação em OCaml. Completaremos este lembrete com alguns aspectos técnicos relevantes para esta secção.

A avaliação de uma sequência de instruções (que, lembremos, é separada por `;`) é avaliada da esquerda para a direita.

```
# print_string "1 "; print_string "2 ";print_string "3 "; print_string "4 ";;
1 2 3 4 - : unit = ()
```

Recordemos os efeitos, de aparência esotérica, da mistura de efeitos laterais (com base por exemplo nestas sequências) com avaliação parcial.

A expressão `let x = e1 in e2` é avaliada na seguinte ordem: primeiro `e1`, depois a atribuição do valor `e1` à variável `x` e finalmente é avaliada a expressão `e2`. Este caso é mais uma ilustração da avaliação ansiosa.

```
# let x = 8;;
val x : int = 8
# let x = print_int x; x + 1 in
let x = print_string "ola"; x + 2 in
    print_string ("-->"^string_of_int x^"\n"); x+3;;
8ola-->11
- : int = 14
```

A expressão `if b then e1 else e2` avalia-se da seguinte forma: primeiro é avaliado a expressão `b` e depois, consoante o valor da avaliação de `b`, é avaliada a expressão `e1` ou (de forma exclusiva) então `e2`.

```
# if true then print_string "ola" else (raise Not_found);;
ola- : unit = ()
```

Podemos dizer aqui que na expressão `if b then e1 else e2` a expressão `b` é avaliada de forma ansiosa enquanto `e1` e `e2` são avaliados de forma preguiçosa.

Encontramos este tipo de desenho em outras situações em OCaml.

```
# (true) || (print_string "ola\n"; false);;
- : bool = true
# (false) || (print_string "ola\n"; false);;
ola
- : bool = false
# (false) && (print_string "ola\n"; false);;
- : bool = false
```

A avaliação de uma expressão, como em `f x y z`, segue a regra seguinte: primeiro é avaliada a lista de expressões (`x`,`y`, e `z`) depois é avaliado a função `f` com os valores resultantes da avaliação de `x`,`y`, e `z`.

Um ponto subtil e importante é referente à ordem de avaliação da sequência de expressões `x`,`y`, e `z`. Esta ordem não é especificada. Uma das razões é deixar espaço aos *designers* de compiladores OCaml para optimizações futuras que podem ser sensíveis à ordem de avaliação destas situações.

```
# let f a b = a + b;;
val f : int -> int -> int = <fun>
# let v = f ((1 * 2) + (3 + 4))   ((5 * 6) + (7 + 8));;
val v : int = 54
# let v1 = f (print_string "A"; ( ( print_string "B";1) *
                    ( print_string "C";2)) + (( print_string "D";3) +
                    ( print_string "E";4)))
              (print_string "F"; ( ( print_string "G";5) *
                    ( print_string "H";6)) + (( print_string "I";7) +
                    ( print_string "J";8)));;
FJIHGAEDCBval v1 : int = 54
#
```

Como vemos neste exemplo a versão actual da distribuição de OCaml utilizada avalia listas de expressões (como em `((1 * 2) + (3 + 4)) ((5 * 6) + (7 + 8))`) da direita para esquerda.

Esta escolha, num contexto puramente funcional, não tem importância nenhuma. De facto, em $(5 + 6) * (7 + 8)$, avaliar $(5 + 6)$ antes de $(7 + 8)$, ou vice versa, é irrelevante. Infelizmente num contexto em que se mistura efeitos co-laterais com computação, tal deixa de ser sem consequências. O programador deverá ter consciência desse fenómeno.

5.2 Avaliação Preguiçosa em OCaml

É no entanto possível, e mesmo fácil, ter avaliação preguiçosa de funções em OCaml de forma *ad-hoc*.

Para tal pode se usar o mecanismo originalmente designado por *thunk* (em inglês designações alternativas são *suspension*, *suspended computation* ou ainda *delayed computation*, em português optaremos por *mecanismo de avaliação atrasada*).

Este mecanismo baseia-se na percepção de que ambas as expressões *expr* e $fun() \rightarrow expr$ se avaliam sempre para o mesmo valor. A diferença, essencial, é que a avaliação de *expr* tem lugar no momento da definição enquanto a avaliação de $fun() \rightarrow expr$ não é realizada no momento da definição, mas sim no momento da sua invocação/aplicação.

Assim, podemos definir uma função `preg` (para `preg`uiça) que transforma qualquer expressão fechada (i.e. expressão que toma um valor não funcional) em expressão com avaliação atrasada. Adaptando igualmente a função `ciclo` previamente definida por forma a atrasar a sua avaliação (basta juntar um último parâmetro de tipo `unit`) então temos de forma ad-hoc a avaliação preguiçosa.

```
# let preg x = fun () -> x;;
val preg : 'a -> unit -> 'a = <fun>
# let ciclo2 n () =
while true do () done; n;;
  val ciclo2 : 'a -> unit -> 'a = <fun>
# let tricky2 n m = (n()) + 1;;
val tricky2 : (unit -> int) -> 'a -> int = <fun>
# tricky2 (preg (5 + 2))  (ciclo2 7);;
- : int = 8
```

Se este exemplo pode parecer excessivamente artificial para ter a avaliação preguiçosa explicitamente OCaml, citemos dois argumentos que poderão aumentar o interesse do programador.

Primeiro, a avaliação preguiçosa possibilita em certos cenários optimizações óbvias (calcular o que é estritamente necessário calcular e evitar calcular o que é desnecessário). Permite também trabalhar com estruturas de dados potencialmente infinitas.

Segundo, existe um suporte nativo à avaliação preguiçosa em OCaml. Este é feito com a ajuda ao módulo `Lazy` descrito mais adiante.

Mas, antes de descrever este suporte nativo, continuamos com a nossa exploração do infinito e mais além em OCaml. Desta vez ao nível das estruturas de dados.É possível definir de forma simples estrutura de dados simples infinitas. A questão aqui é que o seu uso num contexto habitual em OCaml pouco interessante é porque muito limitado ou sujeitos a muitos riscos...

```
# let rec l = 1 :: l;;
```

```
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   ...]
# hd l;;
- : int = 1
# tl l;;
- : int list =
[1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
#  (* Já foi avisado, nada disso em casa...*)
  length l;;
(* \infty + 1 ...*)
```

Uma ilustração destes limites pode ser a seguinte.

```
# let rec l = let n = ref (-1) in (n:=!n+1; !n)::l;;
val l : int list =
```

```
  [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   ...]
# let rec l =  let v = try hd l with _ -> 0 in (v+1)::l;;
Characters 13-53:
  let rec l =  let v = try hd l with _ -> 0 in (v+1)::l;;
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: This kind of expression is not allowed as right-hand side of `let rec'
```

Então, à semelhança da avaliação preguiçosa de funções, como podemos definir estruturas de dados infinitas e manipula-las de forma segura?

```
# type 'a fluxo = Nil | Cons of 'a * (unit -> 'a fluxo);;
type 'a fluxo = Nil | Cons of 'a * (unit -> 'a fluxo)
# let rec de n = Cons (n, fun () -> de (n+1));;
val de : int -> int fluxo = <fun>
# let cab s = match s with Nil -> failwith "cab" | Cons (x,_) -> x;;
val cab : 'a fluxo -> 'a = <fun>
# let cau s =   match s with Nil -> failwith "cau" | Cons (_,y) -> y();;
val cau : 'a fluxo -> 'a fluxo = <fun>
# let rec n_esimo s n = if n=0 then cab s else n_esimo (cau s) (n-1);;
val n_esimo : 'a fluxo -> int -> 'a = <fun>
# let from_list l = fold_right (fun x s -> Cons (x, fun () -> s)) l Nil;;
val from_list : 'a list -> 'a fluxo = <fun>
# let rec from_function f = Cons  (f(), fun () -> (from_function f));;
val from_function : (unit -> 'a) -> 'a fluxo = <fun>
# let rec toma  s n =
    if n <= 0 then [] else
```

```
    match s with
        Nil -> []
      | _ -> cab s :: toma (cau s) (n -1);;
        val toma : 'a fluxo -> int -> 'a list = <fun>
# let rec map f s = match s with Nil -> Nil
  | _  -> Cons (f (cab s), fun () -> map f (cau s));;
 val map : ('a -> 'b) -> 'a fluxo -> 'b fluxo = <fun>
# let rec filtro f s =
  match  s with
      Nil -> Nil
    | Cons (x,g) ->
      if f x then Cons (x,fun () -> filtro f (g ()))
      else filtro f (g ());;
          val filtro : ('a -> bool) -> 'a fluxo -> 'a fluxo = <fun>
```

Experimentemos estas funções sobre os fluxos.

```
# let rec  uns  = Cons (1,(fun () -> uns));;
val uns : int fluxo = Cons (1, <fun>)
# toma uns 10;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
# cab uns;;
- : int = 1
# n_esimo uns 4;;
- : int = 1
# let cont = ref 0;;
val cont : int ref = {contents = 0}
# let incr () = cont:= !cont +1 ; !cont;;
val incr : unit -> int = <fun>
# let fl = from_function incr;;
val fl : int fluxo = Cons (1, <fun>)
# n_esimo fl 8;;
- : int = 9
# let inteiros = de 0;;
val inteiros : int fluxo = Cons (0, <fun>)
# toma inteiros 15;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14]
# toma (map fact inteiros) 14;;
```

```
- : int list =
[1; 1; 2; 6; 24; 120; 720; 5040; 40320; 362880; 3628800; 39916800; 479001600;
 -215430144]
# let par n = n mod 2 = 0;;
val par : int -> bool = <fun>
# toma (filtro par inteiros) 20;;
- : int list =
[0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20; 22; 24; 26; 28; 30; 32; 34; 36; 38]
# let fib =
  let rec fibgen a b = Cons (a,fun () -> fibgen b (a+b)) in fibgen 1 1;;
  val fib : int fluxo = Cons (1, <fun>)
# toma fib 15;;
- : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610]
# n_esimo fib 12345;;
- : int = 351394967
#(*remover múltiplos de n do fluxo*) let del_mult n =  filtro (fun m -> m mod n <>
val del_mult : int -> int fluxo -> int fluxo = <fun>
# let rec crivo s =
  match s with  Nil -> Nil
    | Cons (p,g) -> Cons (p , fun () -> crivo (del_mult p (g())));;
    val crivo : int fluxo -> int fluxo = <fun>
# let primos = crivo (de 2);;
val primos : int fluxo = Cons (2, <fun>)
# toma primos 30;;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97; 101; 103; 107; 109; 113]
# n_esimo primos 10000;;
- : int = 104743
```

5.3 O Módulo Lazy

Reconsideremos o nosso exemplo `tricky` e tentemos generalizar o nosso tratamento das *avaliações atrasadas*.

```
# type 'a pregv =
    Avaliado of 'a
  | Atrasado of  (unit -> 'a);;
```

```
    type 'a pregv = Avaliado of 'a | Atrasado of (unit -> 'a)
# type 'a pregt = {mutable c: 'a pregv};;
type 'a pregt = { mutable c : 'a pregv; }
# let avalia e =
  match  e.c with
      Avaliado x -> x
    | Atrasado f -> let u = f () in e.c <- Avaliado u; u;;
      val avalia : 'a pregt -> 'a = <fun>
# let preg = fun x -> {c = Atrasado (fun () -> x)};;
val preg : 'a -> 'a pregt = <fun>
```

o tipo `pregv` agrupa os valores avaliados de forma preguiçosa. Ou estes estão avaliados ou estão a espera de serem avaliados.

O tipo `pregt` encapsula um valor preguiçoso, ou seja representa um fecho de um valor preguiçoso.

Dotamo-nos também de duas funções. Uma, que nós é já familiar, `preg` permite encapsular uma expressão numa envolvente preguiçosa. A função `avalia` trata de forçar a avaliação de uma expressão preguiçosa. Se esta já se encontra avaliada então o valor é novamente retornado.

Equipado com estas definições podemos, assim o esperamos, definir funções com avaliação preguiçosa de uma forma menos ad-hoc. Para fins de ilustração, tentemos redefinir a função factorial nestes termos. Para isso redefinimos igualmente a condicional `if then else` de forma preguiçosa.

```
# let se_atrasado b e1 e2 = if avalia b then avalia e1 else avalia e2;;
val se_atrasado : bool pregt -> 'a pregt -> 'a pregt -> 'a = <fun>
# let rec fact_atrasado1 n =
  se_atrasado
     (preg (n=0))
     (preg 1)
     (preg (n*(fact_atrasado1 (n-1))));;
val fact_atrasado1 : int -> int = <fun>
# fact_atrasado1 3;;
Stack overflow during evaluation (looping recursion?).
```

Correu inesperadamente mal (será???)... mas porquê?

O fenómeno, já presentes em exemplos anteriores, explica-se pelo facto de estarmos a misturar avaliação ansiosa com avaliação preguiçosa. De facto a

avaliação de `(preg (n*(fact_atrasado1 (n-1))))` é feito de forma ansiosa, mesmo se o retorno é uma expressão preguiçosa. Logo *looping recursion*...

Se queremos evitar este fenómeno, temos de optar pela definição seguinte:

```
# let rec fact_atrasado n =
  se_atrasado
    {c=Atrasado (fun () -> (n=0))}
    {c=Atrasado (fun () -> 1)}
    {c=Atrasado (fun () ->  (n*(fact_atrasado (n-1))))};;
        val fact_atrasado : int -> int = <fun>
# fact_atrasado 3;;
- : int = 6
```

O que torna a utilidade de `preg` muito relativa. É para contrariar este fenómeno que o módulo `Lazy` providencia um tipo `Lazy.t` (em todo semelhante ao nosso tipo `pregt`), a construção `lazy` e a função `force` (para além de algumas variantes, que podem ser consultadas no manual de referência). Esta última é equivalente à função `avalia`. A construção `lazy` (é uma palavra chave providenciada por este módulo, não é uma função no sentido clássico) tem o mesmo papel do que a função `preg` mas sem o seu grande inconveniente: o seu parâmetro não é avaliado.

Tal não deve ser considerado pelo leitor como uma excepção ao mecanismo de avaliação de chamadas à funções de OCaml. A expressão `lazy x` é simplesmente entendido como uma directiva ao preprocessador de OCaml para a sua substituição em algo como `{c=Atrasado (fun () -> x)}`.

```
# open Lazy;;
# let ciclo  = fun n ->  (while true do () done; n);;
val ciclo : 'a -> 'a = <fun>
# let tricky = fun n m -> (force n) + 1;;
val tricky : int Lazy.t -> 'a -> int = <fun>
# tricky (lazy (5 + 2)) (lazy (ciclo 7));;
- : int = 8
```

5.4 Streams: fluxos ou listas infinitas em OCaml

Os fluxos (stream na terminologia OCaml) são listas polimórficas de elementos de mesmo tipo, potencialmente infinitas.

São a solução OCaml para as listas infinitas que descrevemos anteriormente nesta secção. Como tal são também preguiçosas.

OCaml oferece esta funcionalidade pelo intermédio do seu preprocessador, o `camlp4`. Como tal a opção `-pp camlp4o` deverá estar presente na linha de comando de qualquer compilação de programa usando fluxos.

O tipo dos fluxos em si é abstracto (encapsulado no módulo `Stream`). Este comportamento é uma consequência evidente do seu estatuto preguiçoso (ou a sua natureza infinita). O valores são fornecido à pedido explícito. Para ter acesso aos fluxos e as suas diversas construções deveremos então recorrer às funções oferecidas por este módulo.

5.4.1 Construção

O fluxo vazio tem sintaxe por `[<>]`. Um fluxo apresentando a sequência 1, 2, 3 e 4 define-se por `[<'1; '2; '3; '4>]`.

Qualquer expressão que não seja precedida por uma apóstrofe é considerada como um sub-fluxo do fluxo considerado.

Assim `[<'1; '2; [< '3 ; '4; '5>]; '6 ; '7>]` = `[<'1; '2; '3 ; '4; '5; '6 ; '7>]`

```
# #load "dynlink.cma";;
# #load "camlp4o.cma";;
Camlp4 Parsing version 3.11.2

# let concat a b = [<a;b>];;
val concat : 'a Stream.t -> 'a Stream.t -> 'a Stream.t = <fun>
# concat [<'1; '2; [< '3 ; '4; '5>] >] [<'6 ; '7>];;
- : int Stream.t = <abstr>
# let rec nat_from n = [<'n ; nat_from (n+1) >];;
val nat_from : int -> int Stream.t = <fun>
# let inteiros  = nat_from 0;;
val inteiros : int Stream.t = <abstr>
```

O módulo `Stream` apresenta 4 funções de base para a construção de fluxos a partir de produtores de elementos externos.

```
val from : (int -> 'a option) -> 'a Stream.t
val of_list : 'a list -> 'a Stream.t
val of_string : string -> char Stream.t
```

`val of_channel : in_channel -> char Stream.t`

`Stream.from f` devolve um fluxo alimentado pela função `f`. A função em causa tem por requisito principal devolver `Some x` enquanto houver elemento por considerar, e devolver `None` quando o fluxo tem de terminar.

As funções `of_list`, `of_string` e `of_channel` funcionam da forma esperada (um fluxo é criado com base no parâmetro, e a ordem dos elementos é conservada).

5.4.2 Consumo e percurso de fluxos

Para a exploração de um fluxo o módulo nos fornece a função `next` que devolve o elemento à cabeça do fluxo considerado. De reter o efeito lateral que é o consumo do elemento extraído.

```
# next inteiros;;
- : int = 0
# next inteiros;;
- : int = 1
# next [<>];;
Exception: Stream.Failure.
```

De forma geral, o percurso de um fluxo é destrutivo. Cada valor extraído é consumido (removido do fluxo).

Assim se explica o comportamento da função `empty` que testa se o fluxo em parâmetro é vazio ou não.

```
# empty [<>];;
- : unit = ()
# empty inteiros;;
Exception: Stream.Failure.
```

Neste caso este comportamento assegura que o fluxo em parâmetro não é consumido (pela forma com que esta função é implementada e se comporta).

Uma função útil é a função `iter` (com a assinatura seguinte `val iter : ('a -> unit) -> 'a Stream.t -> unit`) que varre o fluxo todo, aplicando a função `f` a cada elemento consumido. Obviamente o seu uso requer algumas precauções (o fluxo pode ser infinito...).

```
# let v =  [<'1;'2;'3;'4>];;
val v : int Stream.t = <abstr>
# Stream.iter (fun x -> print_endline (string_of_int x)) v;;
1
2
3
4
- : unit = ()
# next v;;
Exception: Stream.Failure.
```

Vamos ilustrar algumas outras funções disponíveis com base em exemplos simples. Para mais informação, convida-se o leitor à consulta do manual de referência.

```
# let v =  [<'1;'2;'3;'4;'5;'6;'7;'8;'9;'10;'11;'12;'13>];;
val v : int Stream.t = <abstr>
# peek  v;;
- : int option = Some 1
# next v;;
- : int = 1
# junk v;;
- : unit = ()
# next v;;
- : int = 3
# (* how many element where discarded until now*)
count v;;
- : int = 3
# next v;;
- : int = 4
# npeek 3 v;;
- : int list = [5; 6; 7]
# next v;;
- : int = 5
```

5.4.3 Filtragem destrutiva de fluxos

Para programar sobre fluxos OCaml fornece um mecanismo de filtragem adaptado às particularidades desta estrutura, a filtragem destrutiva. Realça-

se que qualquer motivo filtrado é consumido do fluxo.

A sintaxe é

```
match  fluxo  with parser
    [< 'p1  >] -> expr1
  |     ...
  |  [<'pn  >] -> exprn
```

Em alternativa, e a semelhança do mecanismo de filtragem na definição de função, podemos usar a sintaxe `parser [< 'p1  >] -> expr1 | ... |  [<'pn  >] -> exprn`.

Com base neste mecanismo, podemos redefinir a função `next`.

```
# let prox s = match s with parser  [< 'x >] -> x;;
val prox : 'a Stream.t -> 'a = <fun>
# prox v;;
- : int = 6
# let new_prox  = parser  [< 'x >] -> x;;
val new_prox : 'a Stream.t -> 'a = <fun>
# new_prox v;;
- : int = 7
```

Uma consequência interessante da natureza destrutiva dos fluxos é a filtragem do fluxo vazio. De facto dado um fluxo `s`, este é sempre igual a `[< [<>]; s >]`. Logo é sempre possível encontrar o motivo `[<>]` à cabeça de qualquer fluxo.

Assim, o motivo `[<>]` tem o papel do motivo wildcard `_` nas filtragens tradicionais. Logo, num conjunto de motivos, o motivo `[<>]` tem de ser considerado em última posição. Assim:

```
 # let rec aplica f s = match s with parser
    [<'x ; cont>] ->  f x ; aplica f cont
  | [<>] -> ();;
    val aplica : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# let rec aplica f s = match s with parser
    [<'x >] ->  f x ; aplica f s
  | [<>] -> ();;
    val aplica : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
```

Realça-se o facto da segunda versão de `aplica` (uma variante da função `iter`) tirar proveito do consumo de `x` do fluxo (ou seja `s` é de facto igual a `cont` depois da filtragem do primeiro motivo).

Vejamos agora outro fenómeno interessante. Tentemos definir a função que nos diz se o comprimento de um fluxo é par ou não. Obviamente esta função tem uma utilidade duvidosa em casos de fluxo infinito.

```
# let rec comp_par s = match s with parser
    [<'x ;'y>] ->  comp_par  s
  | [<'z >] -> false
  | [<>] -> true;;
      Characters 78-80:
    | [<'z >] -> false
        ^^
Warning U: this match case is unused.
val comp_par : 'a Stream.t -> bool = <fun>
# comp_par [<'1; '1; '1; '1;'1>];;
Exception: Stream.Error "".
```

A mensagem de aviso já deveria nos preocupar... e de facto uma execução simples revela o problema.

A origem é simples, mas subtil. Ao tentar emparelhar o fluxo final (no qual só resta um único 1) o primeiro motivo é experimentado. Claro este falha porque só há um elemento. De facto o que acontece é o seguinte:

O emparelhamento é destrutivo. Logo o ultimo 1 foi consumido, com sucesso, via o padrão x e a tentativa de emparelhar o y fracassa.É esta situação que provoca a excepção (e o aviso de que o padrão seguinte não serve para nada).

A solução é "Nunca ignorar uma mensagem de aviso" e aninhar filtragens.

```
# let rec comp_par s = match s with parser
    [<'x >] ->
      (match s with parser
          [<'y >] ->  comp_par  s
        | [<>] -> false)
  | [<>] -> true;;
          val comp_par : 'a Stream.t -> bool = <fun>
# comp_par [<'1; '1; '1; '1;'1>];;
- : bool = false
```

Para terminar mostramos uma característica muito interessante da filtragem de motivo: o consumo Cavaco Silva. Ou seja o consumo interno, porque o que é nosso é bom.

```
# let rec comprimento s =  match s with parser
          [<'x; c = comprimento  >] -> 1 + c
        | [<>] -> 0;;
    val comprimento : 'a Stream.t -> int = <fun>
# comprimento [<'1; '1; '1; '1;'1>];;
- : int = 5
```

Esta função poderia ter sido escrita classicamente como:

```
# let rec comprimento s =  match s with parser
          [<'x >] -> 1 + (comprimento s)
        | [<>] -> 0;;
    val comprimento : 'a Stream.t -> int = <fun>
# comprimento [<'1; '1; '1; '1;'1>];;
- : int = 5
```

Veremos mais adiante, na secção dos exemplos completos, que esta pequena característica torna o uso dos fluxos muito interessante.

6 Módulos

Esta secção é fortemente baseada nos apontamentos cedidos gentilmente por Jean-Christophe Filliâtre (Initiation à la programmation fonctionnelle. Master Informatique M1, Université Paris-Sud. `http://www.lri.fr/~filliatr/m1/cours-ocaml.en.html`).

Quando os programas tornam-se grandes,é importante que a linguagem de programação de suporte forneça boas ferramentas de *Engenharia de Software*. Tais ferramentas deverão, em particular, permitir uma estruturação do código em unidades de tamanho razoável (*modularidade*), ocultar a representação concreta de certos dados (*encapsulamento*) e evitar da melhor forma a duplicação de código.

Existe diversas formas para atingir tais objectivos. Na programação orientada à objectos, as classes e os conceitos associados asseguram esse suporte. Em OCaml, para além da camada objecto existente, estas funcionalidades são alternativamente suportadas pelo sistemas de *módulos*.

Como o nome de *módulo* o sugere, um móduloé essencialmente uma forma de introduzir modularidade num programa, i.e. de o estruturar em *unidades* de tamanho razoável.

6.1 Ficheiros e módulos

A unidade programática mais simples no contexto dos módulosé o ficheiro. Em OCaml, cada ficheiro constituí um módulo diferente. Se este, digamos `arith.ml`, conter um determinado número de declarações,

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

então a compilação deste ficheiro, visto como um módulo, faz-se invocando uma compilação sem edição de ligações graças à opção `-c` do compilador (como no caso do `gcc`).

```
% ocamlc -c arith.ml
```

O resultado desta compilação é composta por dois ficheiros: `arith.cmo` contendo o código e `arith.cmi` contendo a sua interface (.i.e. uma constante `pi` de tipo `float` e uma função `round` de tipo `float -> float`).

O nome do móduloé o nome do ficheiro, iniciado por uma maiúscula. No caso concreto do exemplo,é `Arith`.

Podemos assim referenciar os elementos deste módulo noutro ficheiro OCaml pelo uso da notação `Arith.`x. Assim podemos utilisar o módulo `Arith` num ficheiro `main.ml` :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

e compilar este último da seguinte forma:

```
% ocamlc -c main.ml
```

Para que esta compilação seja bem sucedidaé necessário que o módulo `Arith` esteja previamente compilado. De facto, quandoé encontrada uma referência a um elemento do módulo, como aqui `Arith.pi`, o compilador procura um ficheiro de interface `arith.cmi`. Podemos assim realizar a edição das ligações, fornecendo os dois ficheiros de código ao compiladorOCaml:

```
% ocamlc arith.cmo main.cmo
```

Uma forma alternativa possível teria sido compilar directamente o`main.ml` e assim realizar a compilação do ficheiro de uma só vez.

```
% ocamlc arith.cmo main.ml
```

Ou ainda realizar a compilação de ambos os ficheiros fontes e a edição das ligações de uma só vez.

```
% ocamlc arith.ml main.ml
```

Neste último caso os ficheiros são compilados na ordem apresentada, logo a dependência do ultimo ficheiro sobre o primeiroé respeitada e produz assim o efeito desejado.

6.2 Encapsulamento

Os módulos, na sua versão *ficheiro*, permitam a estruturação de um programa (em *unidades de código*). Mas os módulos permitam muito mais do que isso, por exemplo o *encapsulamento*, i.e. a ocultação de determinados detalhes de código. De facto,é possível fornecer uma *interface* aos módulos definidos. Neste caso só os elementos declarados na interface serão visíveis, à semelhança do mecanismo Java representado pela palavra chave `private` que permite controlar (limitar, aqui) a visibilidade dos atributos ou dos métodos.

Para tal, coloca-se a interface num ficheiro com a extensão `.mli` na mesma pasta onde se encontra o ficheiro fonte (de extensão `.ml`).

Assim podemos querer exportar somente a função `round` do módulo `Arith` previamente definido, criando o ficheiro `arith.mli` com o conteúdo:

```
val round : float -> float
```

Constata-se que a sintaxeé a mesma que a sintaxe utilizada pelo toplevel de OCaml na sua interacção (na resposta para ser mais preciso) com o programador.

Este ficheiro interface deve ser convenientemente compilado *antes* do ficheiro código.

```
% ocamlc -c arith.mli
% ocamlc -c arith.ml
```

Aquando da compilação do código, o compilador verificará a adequação dos tipos entre os elementos declarados na interface e os elementos de facto fornecidos no código.

No caso do exemplo anterior, compilar o módulo `main.ml` resultará no erro:

```
% ocamlc -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

No entanto a constante `pi` permanece obviamente visível e acessível aos elementos presentes no ficheiro `arith.ml`.

A interface não se limita na restrição dos valores exportados. Permite também restringir os tipos exportados ou até mesmo na definição destes.

Suponhamos que queiramos programar uma pequena biblioteca sobre conjuntos de inteiros representados por listas. O código poderia ser constituído pelo ficheiro `ensemble.ml` seguinte:

```
type t = int list
let vazia = []
let junta x l = x :: l
let pertence = List.mem
```

e uma interface possível seria:

```
type t = int list
val vazia : t
val junta : int -> t -> t
val pertence : int -> t -> bool
```

Aqui a definição do tipo `t` nãoé de nenhuma utilidade, excepto, eventualmente, facilitar a leitura da interface pela identificação dos conjuntos ao tipo `t`. Sem a menção do tipo, o compilador teria inferido ele próprio esta informação e poderia ter então atribuído um tipo mais geral mas compatível com as funções `vazia`, `junta` e `pertence`.

Mas podemos também querer esconder a representação escolhida do tipo `t`, dando na interface o conteúdo seguinte:

```
type t
val vazia : t
val junta : int -> t -> t
val pertence : int -> t -> bool
```

Um tal tipo designa-se de tipo abstracto. Dentro do ficheiro `ensemble.ml` a representação do tipo `t`é conhecida, foraé desconhecida. É uma noção essencial sobre a qual nos debruçaremos mais adiante aquando da exploração da noção de persistência.

O facto de abstrair o tipo `t` esconde por completo a forma como os valores deste tipo são formados. Se procuramos processar um valor do tipo `Ensemble.t` como uma lista de inteiros fora do dito ficheiro, obteremos um erro.

Assim, podemos modificar a codificação da noção de conjunto sem nunca perturbar os programas que usam as funcionalidades fornecidas pelo módulo `Ensemble`.

6.2.1 Compilação separada

A linguagem OCaml permite de facto a *compilação separada*. Isto é, a compilação dum ficheiro *só* depende das interfaces dos módulos que menciona e não do código que lhe estão associadas.

Assim se um módulo `B` utiliza um módulo `A`, uma alteração do código do módulo `A` não obriga a recompilação do módulo `B`. Somente uma alteração da *interface* de `A` justificaria tal recompilação. Isto porque seria, neste caso, necessário verificar se alterações aos tipos dos valores exportados por `A` preservam a boa tipificação de `B`.

A compilação separada representa um ganho de tempo considerável para o programador. Neste contexto e para um projecto constituído por cem fi-

cheiros OCaml, uma alteração do código só implicará uma *única* compilação e uma edição de ligações para reconstruir o executável.

6.2.2 Linguagem de módulos

Uma das forças do sistema de módulos de OCaml é ele não se limitar aos ficheiros. Podemos de facto definir um novo módulo da mesma forma que podemos definir um tipo, uma constante, uma função ou uma excepção.

Assim pode mos escrever:

```
module M = struct
  let c = 100
  let f x = c * x
end
```

A palavra chave `module` introduz aqui a declaração de um novo módulo, `M`, cuja definiçãoé o bloco iniciado por `struct` e concluído por `end`.

Tal blocoé designado de *estrutura* (o que justifica a palavra chave `struct`) eé composto de uma sequência de declarações e/ou de expressões OCaml, *exactamente* como no caso dum programa OCaml. Tais declarações de módulos podem ser misturadas com outras declarações e até mesmo serem aninhadas e assim conter outras declarações de módulos. Poderíamos assim escrever algo como:

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

Tais módulos "locais" ganham particular importância com a utilização dos functores que serão abordados mais adiante na secção seguinte. Mas podemos desde já constatar que estes permitam uma organização do *espaço de nomes*. Assim no exemplo anterior, pudemos definir duas funções de nome `f`, porque a função definida no módulo `B` nãoé imediatamente visível no corpo do módulo `A`. É necessário qualificar, com a notação `B.f`, para de facto ter acesso a tal função dentro do módulo `A`.

Podemos assim servirmo-nos dos módulos locais para agrupar determinados tipos e funções interligados e dar-lhes nomes genéricos (como `add`, `find`, etc.) porque a qualificação permite-lhes serem distinguidos de outras funções com o mesmo nome.

Quando tais valores são referenciados frequentemente, podemos evitar a qualificação sistemática pelo nome do módulo *abrindo* explicitamente o módulo com a ajuda do comando `open`. A consequência desta aberturaé tornar "visível"todos os elementos deste módulo.

Assim, no lugar da invocação sistemática `Printf.printf`, preferiremos abrir o módulo `Printf` no inicio do ficheiro :

```
open Printf
...
let print x y = printf "%d + %d = %d\n" x y (x+y)
...
```

Assinaturas

À semelhança de qualquer outra declaração, as declarações de módulos são tipificadas (i.e. os módulos têm tipos).

O tipo de um módulo designa-se por *assinatura*. A sintaxe de uma assinaturaé idêntica à de uma interface (os ficheiros `.mli`) eé iniciada pela palavra chave `sig` e terminada pela palavra chave `end`. Constatamo-lo na seguinte declaração no toplevel:

```
# module M = struct let a = 2 let f x = a * x end;;
module M : sig val a : int val f : int -> int end
```

É lógico que a sintaxe seja idêntica a de uma interface porque uma interface nãoé nada mais de que o tipo de um módulo-ficheiro.

Da mesma forma que se pode definir módulos locais, podemos definir assinaturas:

```
# module type S = sig val f : int -> int end;;
module type S = sig val f : int -> int end
```

Podemos então servirnos de tais assinaturas para tipificar explicitamente as declarações de módulos:

```
# module M : S = struct let a = 2 let f x = a * x end;;
module M : S
```

O interesse de tais declaraçõesé o mesmo que o interesse das interfaces: rstringir os valores ou tipos exportados (que designamos anteriormente de encapsulamento). Aqui o valor `M.a` deixou de ser visível:

```
# M.a;;
Unbound value M.a
```

Resumo

O sistema de módulos OCaml permite

- modularidade, pela estruturação do código em unidades designadas de *módulos*;
- encapsulamento de tipos e valores, e em particular da definição de *tipos abstractos*;
- uma verdadeira compilação separada;
- uma organização do espaço de nomes.

6.3 Functores

Da mesma forma que uma função OCaml pode ser genérica relativamente aos tipos dos seus parâmetros (polimórfismo) ou às funções em parâmetro (ordem superior), um módulo pode ser parametrizado por um ou mais outros módulo. Um tal módulo designa-se por *functor*.

Para devidamente justificar a utilidade dos functores, o mais simples é ilustra-los por um exemplo.

Suponhamos que queiramos codificar em OCaml uma tabela de hash da forma mais genérica possível por forma a que esse código possa ser utilizado em qualquer circunstância em que uma tabela de hash possa ser útil.

Para issoé preciso parametrizar o código por uma função de de hash (para a inserção na tabela e a pesquisa) e por uma função de igualdade (para a pesquisa)[5].

Poderíamos então imaginar utilizar a ordem superior e parametrizar todas as funções do nosso módulo por estas duas funções, o que resultaria numa interface da forma:

[5]Acontece que OCaml predefine uma função de igualdade e uma função de hash polimórfica, mas em algumas situações o programador pode desejar utilizar funções diferentes.

```
type 'a t
  (* o tipo abstracto das tabelas de hash contendo elementos de
     tipo "'a" *)
val create : int -> 'a t
  (* "create n" cria uma nova tabela de hash de tamanho inicial "n" *)
val add : ('a -> int) -> 'a t -> 'a -> unit
  (* "add h t x" junta o valor "x" na tabela "t"  com a ajuda da
      função de hash "h" *)
val mem : ('a -> int) -> ('a -> 'a -> bool) -> 'a t -> 'a -> bool
  (* "mem h eq t x" testa a ocorrência de "x" na tabela "t" para a
     função de hash "h" e a igualdade "eq" *)
```

Para além do aspecto pesado, esta interface tem os seus perigos: de facto podemos juntar um elemento na tabela com uma certa função de hash e depois pesquisar nesta mesma tabela com outra função de hash. O resultado seria errado, mas o sistema de tipos de OCaml não teria excluído esta utilização incorrecta (de facto, apesar do argumento apresentado, esta utilizaçãoé bem tipificada).

Uma solução mais satisfatória consistiria em atribuir uma função de hash no momento da criação da tabela de hash. Estas seriam então agrupadas na estrutura de dados e utilizadas pelas operações de cada vez que seria necessário.

A interface ficaria então mais razoável:

```
type 'a t
  (* o tipo abstracto das tabelas de hash contendo elementos de
     tipo "'a" *)
val create : ('a -> int) -> ('a -> 'a -> bool) -> int -> 'a t
  (* "create h eq n" cria uma nova tabela de hash de tamanho
     inicial "n", com "h" como  função de hash e  "eq" como
     função de igualdade *)
val add : 'a t -> 'a -> unit
  (* "add t x" junta o dado "x" na tabela "t" *)
val mem : 'a t -> 'a -> bool
  (* "mem t x" testa a ocorrência de "x" na tabela "t" *)
```

A representação interna da codificação poderia se assemelhar a algo como:

```
type 'a t = { hash : 'a -> int; eq : 'a -> 'a -> bool; data : 'a list array }
let create h eq n = { hash = h; eq = eq; data = Array.create n [] }
...
```

No entanto, tal codificação ainda tem os seus defeitos. Por começar, o acesso às funções de hash e de igualdade fazem-se necessariamente pelo intermédio de fechos e como tal penaliza a execução relativamente a funções conhecidas estaticamente. Por outro lado a estrutura de dados codificando a tabela contém agora funções e por esta razão torna-se complicado escrever tal representação em dispositivos físicos, como o disco, e reler posteriormente (é tecnicamente possível, mas com recurso a importantes restrições).

Felizmente, os functores trazem aqui uma solução muito satisfatória. Porque pretendemos parametrizar as nossas tabelas de hash por um tipo (o dos seus elementos) e duas funções, vamos escrever um módulo parametrizado por um outro módulo contendo estes três requisitos. Tal functor `F`, parametrizado por um módulo `X` com assinatura `S`, introduz-se com a sintaxe

```
module F(X : S) = struct ... end
```

No caso do nosso exemplo a assinatura `S`é a seguinte:

```
module type S = sig
  type elt
  val hash : elt -> int
  val eq : elt -> elt -> bool
end
```

e a codificação das tabelas de hash pode então se escrever da seguinte forma:

```
module F(X : S) = struct
  type t = X.elt list array
  let create n = Array.create n []
  let add t x =
    let i = (X.hash x) mod (Array.length t) in t.(i) <- x :: t.(i)
  let mem t x =
    let i = (X.hash x) mod (Array.length t) in List.exists (X.eq x) t.(i)
end
```

Constatamos que no corpo do functor, acedemos aos elementos do módulo-parâmetro `X` exactamente como se de um módulo (como definido mais acima) se tratasse. O tipo do functor `F` explicita o parâmetro `X` com a mesma sintaxe:

```
module F(X : S) : sig
  type t
    (* O tipo abstracto das tabelas de hash contendo elemento de tipo
       "X.elt" *)
  val create : int -> t
    (* "create h eq n" cria uma nova tabela de tamanho inicial "n" *)
  val add : t -> X.elt -> unit
    (* "add t x" junta o valor "x" na tabela "t" *)
  val mem : t -> X.elt -> bool
    (* "mem t x" testa a ocorrência de "x" na tabela "t" *)
end
```

Podemos então instanciar o functor `F` com um qualquer módulo escolhido pelo programador, desde que respeite as assinaturas pretendidas.

Assim se desejamos uma tabela de hash para arquivar inteiros, podemos começar por definir um módulo `Integers` com assinatura `S` :

```
module integers = struct
  type elt = int
  let hash x = abs x
  let eq x y = x=y
end
```

Podemos, a seguir, aplicar o functor `F` sobre este module :

```
module Hintegers = F(Integers)
```

Constatamos assim que a definição de um módulo não está limitado a uma estrutura: trata-se de uma aplicação de um functor.

Podemos então utilizar o módulo `Hintegers` como qualquer outro módulo :

```
# let t = Hintegers.create 17;;
val t : Hintegers.t = <abstr>
# Hintegers.add t 13;;
- : unit = ()
# Hintegers.add t 173;;
- : unit = ()
# Hintegers.mem t 13;;
- : bool = true
# Hintegers.mem t 14;;
- : bool = false
...
```

6.3.1 Aplicações

As aplicações dos functores são múltiplas. Utilizam-se em particular para definir:

1. *estruturas de dados parametrizadas por outras estruturas de dados*

   OCaml oferece três deste functores na sua biblioteca standard:

   - `Hashtbl.Make` : tabelas de hash semelhantes às do exemplo anterior;
   - `Set.Make` : conjuntos finitos codificados com base em árvores balenceadas;
   - `Map.Make` : tabelas de associação codificadas com base em árvores balenceadas.

2. *algoritmos parametrizados por estruturas de dados*

   Podemos por exemplo escrever o algoritmo de Dijkstra de procura de caminho mais curto num grafo sob a forma de um functor parametrizado pela estrutura de dados representando o grafo em si. O tipo de tal functor poderia ser:

```
module Dijkstra
  (G : sig
         type graph (* tipo dos grafos *)
         type vertice (* tipo dos vértices *)
```

```
          val vizinho : graph -> vertice -> (vertice * int) list
            (* conjunto dos vizinhos com peso nas arestas *)
        end) :
sig
  val shortest_path :
    G.graph -> G.vertice -> G.vertice -> G.vertice list * int
    (* "shortest_path g a b" procura o caminho mais curto de
       "a" para  "b" em "g" *)
end
```

Vemos assim que os functores são uma ferramenta poderosa para a reutilização de código, porque permite a sua escrita de forma muito genérica. Os functores podem ser comparados, em certos aspectos, aos *templates* de C++,mesmo se há também diferenças numerosas e notórias.

De notar que o sistema de módulos de OCaml é, na realidade, *independente* da linguagem de base. Forma uma linguagem por si só, que poderia ser aplicado a outra linguagem. Em particular ele poderia ser desdobrado estaticamente para dar um código sem módulos nem functores, mesmo se na prática nãoé isso que acontece.

7 Persistência

Esta secção é fortemente baseada nos apontamentos cedidos gentilmente por Jean-Christophe Filliâtre (Initiation à la programmation fonctionnelle. Master Informatique M1, Université Paris-Sud. `http://www.lri.fr/~filliatr/m1/cours-ocaml.en.html`).

Nesta secção abordaremos um aspecto essencial das estruturas de dados, a *persistência*. Esta noção não é específica à programação OCaml, nem, até, da programação funcional, mas é de facto natural nesta família de linguagens.

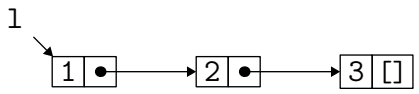
7.1 Estruturas de dados imutáveis

Como já foi referido, muitas das estruturas de dados OCaml são imutáveis. Isto é, não são alteráveis uma vez construídas. Este facto tem uma consequência muito importante: um valor de tal tipo não é afectado pelas

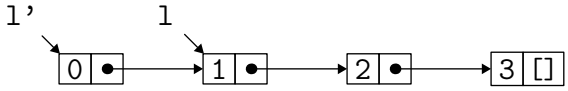
operações das quais é parâmetro. *Novos* valores são construídos e devolvidos por estas últimas.

Ilustremos este facto por exemplos simples.

Se definirmos uma lista `l` por `let l = [1; 2; 3]` então `l` é, em termos de representação memória, um apontador para o primeiro bloco contento `1` e um apontador para o segundo bloco, etc. :



Como referido previamente, se definirmos agora a lista `l'` como a junção de um outro elemento à lista `l`, com a ajuda da declaração `let l' = 0 :: l`, temos agora a situação seguinte:



Isto é, a aplicação do construtor `::` teve por efeitos alocar um novo bloco, cujo primeiro elemento é `0` e o segundo um apontador cujo valor é igual à `l`. A variável `l` continua a apontar para os mesmos blocos.

De uma forma geral, qualquer função definida que manipule listas terá esta propriedade de não alterar as listas que lhe são passadas como parâmetros.

É esta (propriedade) das estruturas de dados que designamos por *persistência*.

É muito importante perceber aqui que existe *partilha*. A declaração de `l'` aloca no máximo um bloco suplementar (já que um só construtor foi aplicada aqui). Os bloco seguintes são meramente re-utilizados mas não são alterados.

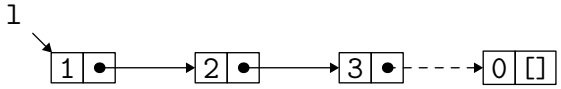
Temos bem duas listas de 3 e 4 elementos respectivamente (`[1;2;3]` e `[0;1;2;3]` para ser mais preciso). Mas somente 4 blocos memória alocados.

Em particular *não houve cópia*. De uma forma geral OCaml nunca copia valores, excepto se se escreve explicitamente uma função para tal operação. Por exemplo:

```
let rec copie = function [] -> [] | x :: l -> x :: copie l
```

Mas, par ser realista, pouca utilidade tem tal função porque as listas não são alteráveis *in-place*.

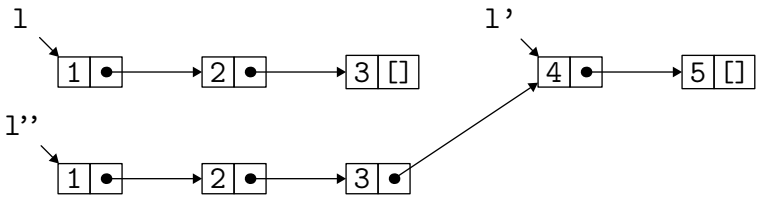
Assim percebe-se que não é tão fácil juntar um elemento na cauda de uma lista como juntar a cabeça. Isto porque tal significaria uma modificação *in-place* da lista `l`.



Para juntar um elemento em fim de lista, é necessário copiar todos os blocos da lista. Tal procedimento é seguindo, em particular, pela função `append` seguinte que constrói a concatenação de duas listas:

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

Constatamos que esta função cria tantos blocos quanto os há na lista `l1`. Os blocos de `l2` são partilhados. Assim, se declaramos `let l' = [4; 5]` e que calculemos a seguir a concatenação `l` e de `l'` com `let l■ = append l l'` teremos a situação seguinte:

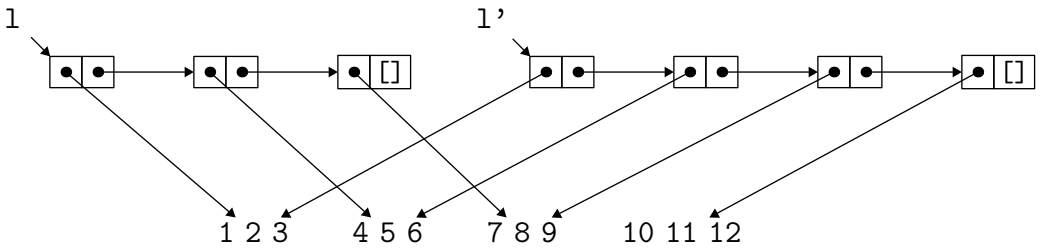


Os blocos de `l` foram copiados e os de `l'` ficaram partilhados.

É por esta razão que as listas devem ser usadas quando o contexto do seu uso considera juntar e retirar elementos na cabeça (à semelhança das estruturas de tipo *pilha*). Quando os acessos e/ou as modificações devem ser realizadas em posições arbitrárias na sequência considerada, então é preferível outra estrutura de dados.

é importante notar que os elementos próprios da lista não são copiados pela função `append`. De facto, `x` designa um elemento de tipo qualquer e nenhuma cópia é realizada sobre o próprio `x`. No caso duma lista de inteiros, isto pouca relevância tem. Mas se considerássemos uma lista `l` contendo

três elementos de um tipo mais complexo, por exemplo triplos de inteiros como no caso de `let l =[(1,2,3); (4,5,6); (7,8,9)]`, então estes ficarão partilhados por `l` e `append l [(10,11,12)]` .



Isto por parecer artificialmente custoso quando se têm o habito de trabalhar com estruturas de dados modificáveis localmente, com é tradicionalmente o caso no contexto de linguagens imperativas. Mas isto seria sub-estimar o interesse prático da persistência.

É alias importante notar que o conceito da persistência pode ser realizado de forma fácil numa linguagem imperativa. Basta manipular as listas ligadas exactamente como o compilador OCaml o faz. De forma inversa, podemos perfeitamente em OCaml manipular listas modificáveis localmente. Por exemplo, definindo:

```
type 'a lista = Vazia | Elemento of 'a element
and 'a element = { valor : 'a; mutable seguinte : 'a lista };;
```

Podemos até mesmo tornar o campo `valor` mutável se o desejarmos. Mas ao contrário das linguagens imperativas, OCaml oferece a possibilidade de definir as estruturas de dados imutáveis de forma natural e segura (porque mesmo se codificamos uma estrutura persistente em C ou Java, o sistema de tipo não poderá impedir a sua modificação local, sendo as variáveis naturalmente mutáveis).

Finalmente é preciso não esquecer que a memória não utilizada é automaticamente recuperada pelo GC (Garbagge Collector) de OCaml. Assim numa expressão como:

```
let l = [1;2;3] in append l [4;5;6]
```

os três blocos de `l` são efectivamente copiados aquando da construção da lista `[1;2;3;4;5;6]` mas são imediatamente apagados (recuperados) pelo GC porque não são mais referenciadas.

Outro exemplo ilustrativo destes fenómenos pode ser o das árvores.

```
# type tree = Empty | Node of int * tree * tree;;
type tree = Empty | Node of int * tree * tree
# let rec add  v ab =
  match ab with
     Empty -> Node (v,Empty,Empty)
   | Node (n,esq,dir) ->
         if v > n then Node (n,esq,(add v dir))
         else Node (n,(add v esq), dir);;
val add : int -> tree -> tree = <fun>
# let (ab1,ab2,ab3) = Empty, Empty,Empty;;
val ab1 : tree = Empty
val ab2 : tree = Empty
val ab3 : tree = Empty
# let ab = Node (12,ab1,(Node (20,ab2,ab3)));;
val ab : tree = Node (12, Empty, Node (20, Empty, Empty))
# add 42 ab;;
- : tree = Node (12, Empty, Node (20, Empty, Node (42, Empty, Empty)))
```

A função `add` pouco copia, mas sim partilha.

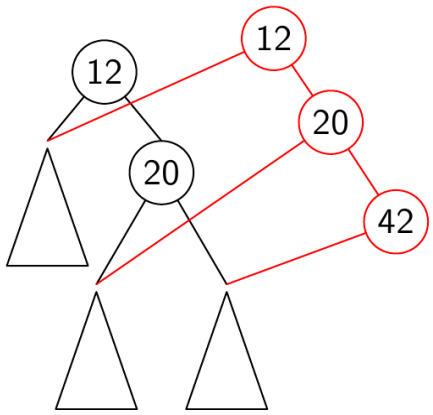


Figura 4: Partilha em árvores

7.2 Interesses práticos da persistência

Os interesses práticos da persistência são múltiplos. De forma imediata percebe-se que a persistência facilita a leitura do código e a sua correcção. De facto podemos raciocinar sobre valores manipulados por programas em termos *matemáticos*, já que estes são imutáveis, de forma equacional e sem se preocupar da ordem da avaliação. Assim é sempre fácil ficar convencido da correcção da função `append` apresentada anteriormente e dos seus requisitos funcionais (i.e. `append l1 l2` constrói a lista formada dos elementos de `l1` seguidos dos elementos de `l2`).

Um simples raciocínio indutivo sobre a estrutura de `l1` chega. Com listas alteráveis localmente, a adaptação da função `append` que redireccione (modifica) o apontador do ultimo bloco de `l1` para o bloco em cabeça de `l2` torna o argumento de correcção muito mais complicado. Um exemplo ainda mais flagrante é o de uma função, num contexto mutável, que devolve de uma lista.

A correcção de um programa não é um aspecto menor e deve sempre ser prioritário a sua eficácia. Quem se importa com um programa super rápido se este está errado?

A persistência não é somente útil para facilitar a correcção dos programas. é igualmente uma ferramenta poderosa em contextos em que a técnica do retrocesso (*backtracking*) é necessária.

Supomos, por exemplo que procuremos escrever um programa que procura a saída de um labirinto, na forma de uma função `procura` que aceita em entrada um estado (persistente) e devolvendo um booleano indicando o sucesso (ou não) da pesquisa.

as deslocações possíveis a partir de um estado são dadas na forma de uma lista por uma função `movimentacoes` e uma outra função `movimenta` calcula o resultado de uma movimentação a parti de um determinado estado, na forma de um novo estado já que pertence a um tipo de dado persistente. Vamos supor que uma função booleana `saida` indica se um estado corresponde ou não a uma saída do labirinto.

Então escreve-se trivialmente a função `procura` da forma seguinte:

```
let rec procura e =
  saida e || tenta e (movimentacoes e)
and tenta e = function
  | [] -> false
```

```
  | d :: r -> procura (movimenta d e) || tenta e r
```

onde `tenta` é uma função auxiliar testando uma por uma as movimentações possíveis de uma lista de movimentações. É a natureza persistente da estrutura de dados representando os estados que permite uma tal concisão de código. De facto, se o estado teria de ser alterado no lugar, precisaríamos então realizar a deslocação antes de efectuar a chamada recursiva a `procura` mas também *cancelar* esta deslocação no caso de esta se revelar um beco sem saída e antes de considerar movimentos alternativos. O código passaria a ser algo como:

```
let rec procura () =
  saida () || tenta (movimentacoes ())
and tenta = function
  | [] -> false
  | d :: r -> (desloca d; procura ()) || (retrocessa d; tenta r)
```

O que é sem margem para dúvida menos claro e mais propício a erros.

Este exemplo, infelizmente, não é artificial: a técnica de *backtracking* é frequentemente utilizada em informática (percurso de grafos, coloração de grafo, contagem de soluções, etc.)

Um outro exemplo típico é o do *porte* em manipulações simbólicas de programas (ou fórmulas).

Supondo, por exemplo, que se pretenda manipular programas Java triviais compostos exclusivamente por blocos, variáveis locais inteiras, de testes de igualdade entre duas variáveis e de `return`. Isto é, programas parecidos com:

```
{
  int i = 0;
  int j = 1;
  if (i == j) {
    int k = 0;
    if (k == i) { return j; } else { int i = 2; return i; }
  } else {
    int k = 2;
    if (k == j) { int j = 3; return j; } else { return k; }
```

```
  }
}
```

Tais programas podem ser representados em OCaml pelo tipo `instr list` onde o tipo soma `instr` define-se como

```
type instr =
  | Return of string
  | Var of string * int
  | If of string * string * instr list * instr list
```

Suponhamos agora que pretendemos verificar que as variáveis manipulados em tais programas são sempre previamente declaradas adequadamente. Para tal podemos escrever duas funções mutuamente recursivas `verifica_instr` e `verifica_prog` que verificam respectivamente que uma instrução e uma lista de instruções são bem formadas (no sentido da adequada declaração de variáveis manipuladas). Para esse efeito, estas duas funções tomam como parâmetro a lista das variáveis que são no momento visíveis (estão no porte/âmbito). O código é assim evidente:

```
let rec verifica_instr vars = function
  | Return x ->
      List.mem x vars
  | If (x, y, p1, p2) ->
      List.mem x vars && List.mem y vars &&
      verifica_prog vars p1 && verifica_prog vars p2
  | Var _ ->
      true
and verifica_prog vars = function
  | [] ->
      true
  | Var (x, _) :: p ->
      verifica_prog (x :: vars) p
  | i :: p ->
      verifica_instr vars i && verifica_prog vars p
```

A simplicidade deste código resulta da utilização de uma estrutura de dados persistente para o conjunto das variáveis (uma lista).

Se tivéssemos utilizado uma tabela imperativa no lugar da lista, seria então preciso retroceder entre o primeiro ramo de um `if` e a segunda, por forma a que possamos *esquecer* as variáveis locais ao primeiro ramo. O código não teria assim sido tão simples.

Este exemplo é relativamente simples, mas encontramos muitas manifestações deste fenómeno na prática, mal façamos manipulações simbólicas e que mecanismos de porte intervenham (tabela de símbolos, ambientes de tipificação, etc.). É assim muito agradável e conveniente utilizar estruturas de dados persistentes.

Vamos abordar uma última ilustração da utilidade da persistência. Imaginemos um programa manipulando uma colecção de dados Só existe uma instância desta colecção de dados a cada instante e então não há *a priori* razão nenhuma para utilizar uma estrutura persistente para a sua codificação. Mas supomos agora que as actualizações por realizar nesta base sejam complexas, isto é implicam cada uma delas um grande número de operações cujo algumas podem falhar.

Encontramo-nos numa situação em que é preciso saber *anular* os efeitos da actualização iniciada. De uma forma esquemática, o código poderá se assemelhar a:

```
try
  ... realizar a actualização ...
with e ->
  ... restabelecer a base para um estado coerente ...
  ... tratar do erro ...
```

Se utilizarmos uma estrutura de dados persistente para a colecção, basta arquivar a base numa referência, por exemplo `bd`. Assim a actualização torna-se simplesmente na actualização desta referência:

```
let bd = ref (... base inicial ...)
...
try
  bd := (... actualização de  !bd ...)
with e ->
```

```
... tratar do erro ...
```

Assim, não há absolutamente nenhuma necessidade anular qualquer coisa. De facto, a actualização, por tão complexa que seja, não faz mais do que construir uma nova base e, uma vez esta terminada, a referência `bd` é modificada para apontar para esta nova colecção. Esta última operação é atómica e não pode falhar. Se há uma qualquer excepção levantada durante a actualização propriamente dita, então a referência `bd` permanecerá inalterada.

7.3 Interface e persistência

A estrutura de dados que codifica as listas é persistente de forma evidente porque é um tipo soma da qual conhecemos a definição: concreto e imutável.

Quando um módulo OCaml implementa uma estrutura de dados na forma de um tipo *abstracto*, a sua natureza persistente (ou não) não é imediata. Evidentemente, um comentário apropriado na interface pode esclarecer o programador/utilizador deste facto. Mas na prática são os tipos das operações disponíveis que fornecem esta informação. Tomemos o exemplo de uma estrutura persistente representando os conjuntos finitos de inteiros. A interface de um tal módulo poderia ser:

```
type t                        (* o tipo abstracto dos conjuntos *)
val empty : t                 (* o conjunto vazio *)
val add : int -> t -> t       (* juntar um elemento *)
val remove : int -> t -> t    (* remover um elemento *)
...
```

A natureza persistente desta representação dos conjuntos é implícita nesta interface.

De facto, a operação `add` devolve um valor de tipo `t`, isto é, um novo conjunto. Neste ponto, em tudo semelhante é a função `remove`. Mais subtilmente é a situação do conjunto vazio: `empty` é um valor e não uma função. Assim todas as ocorrências de `empty` serão partilhadas, qualquer que seja a sua representação.

No entanto, uma estrutura de dados alterável *in-place* para os conjuntos de inteiros (por exemplo sob a forma de tabelas de hash) poderia tomar a forma de:

```
type t                          (* O tipo abstracto dos conjuntos *)
val create : unit -> t          (* o conjunto vazio *)
val add : int -> t -> unit      (* juntar um elemento *)
val remove : int -> t -> unit   (* remover um elemento *)
...
```

Aqui a função `add` não devolve nada, porque o novo elemento foi acrescentado directamente (*in-place...*) na estrutura de dados. Tal também acontece com as outras operações que alteram a estrutura.

A função `create` aceita um argumento de tipo `unit` porque de cada vez que `create` é invocada, esta deve construir uma nova instância da estrutura de dados para que as modificações *in-place* não alterem outras estruturas criadas.

Infelizmente, nada impede dar a uma estrutura de natureza imperativa uma interface *persistente* (devolvendo o valor passado em parâmetro). De forma inversa, nada impede dar uma interface *imperativa* a uma estrutura persistente (que deite fora os valores construídas). nos dois casos, é inútil e perigoso...

Mas tal não significa no entanto que uma estrutura de dados persistente seja necessariamente implementada sem efeitos colaterais. Assim a boa definição da noção de persistência é:

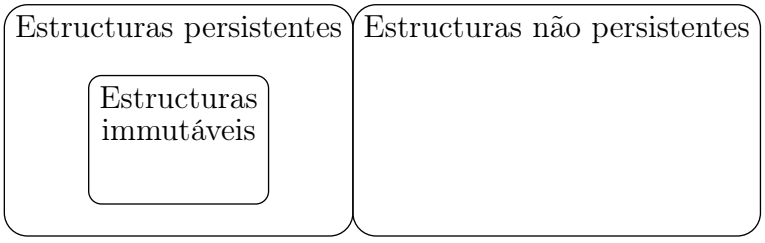
$$\textit{persistência} = \textit{imutável por observação}$$

e não *puramente funcional* (no sentido de ausência de efeitos co-laterais).
Temos uma implicação num só sentido

$$\textit{puramente funcional} \Rightarrow \textit{persistente}$$

A recíproca é falsa. Existem estruturas de dados persistentes que não são puramente funcionais, devido ao seu uso dos efeitos co-laterais.

A situação é a seguinte:



Um exemplo de estrutura persistente, apesar de alterável *in-place*, é o das árvores binárias de pesquisa que evoluem à medida das pesquisas para optimizar as pesquisas dos elementos mais requisitados, as *Splay trees*.

Isto pode ser concretizado implementando a estrutura de dados como uma referência para uma árvore. A árvore optimizada por então substituir a árvore actual por uma simples modificação desta referência.

A estrutura completa permanece observacionalmente persistente, já que não há modificações observáveis do conteúdo, mas sim da *performance*.

Um outro exemplo, simples, é o exemplo das filas (*first-in first-out* ou *queue* em inglês).

Se procuramos uma solução persistente com base em listas, a *performance* não será satisfatória. Juntar um elemento à cabeça não coloca problemas, mas remover eficazmente um elemento à cauda é outra história. O custo é proporcional ao número de elementos, em tempo e em espaço.

Uma alternativa astuta consiste em representar uma fila por duas listas. A primeira contém os elementos acabados de entrar e a segunda os elementos prestes a sair (logo arquivado no sentido contrário). As operações de acréscimo ou de remoção terão então sempre um custo constante ($O(1)$) *excepto* no caso particular em que a lista de saída se encontra vazia. Neste caso invertemos a lista de entrada para poder agora utilizá-la como lista de saída. A lista de entrada ficará, neste contexto, vazia. O custo desta inversão é proporcional ao comprimento da lista por inverter, mas isto é feito uma só vez sobre a lista considerada. A complexidade amortecida (*amortized complexity*, em inglês, i.e. a complexidade limitada ao conjunto de operações de acréscimo e de remoção) permanece em $O(1)$.

A interface mínima para as filas persistentes pode ser:

```
type 'a t
val create : unit -> 'a t
val push : 'a -> 'a t -> 'a t
exception Empty
val pop : 'a t -> 'a * 'a t
```

Podemos propor o código seguinte, com base nas ideias apresentadas mais acima:

```
type 'a t = 'a list * 'a list
let create () = [], []
```

```
let push x (e,s) = (x :: e, s)
exception Empty
let pop = function
  | e, x :: s -> x, (e,s)
  | e, [] -> match List.rev e with
    | x :: s -> x, ([], s)
    | [] -> raise Empty
```

Mas é ainda possível que sejamos obrigado a devolver repetidamente a lista de entrada `e` se a operação `pop` é efectuada repetidamente sobre uma mesma fila da forma `(e,[])`. Tal situação sempre é susceptível de acontecer já que pretendemos construir filas persistentes.

Para remediar esta situação, vamos associar o retorno de uma lista de entrada `e` a um efeito co-lateral na estrutura de dados. Isto não alterará o conteúdo — a fila `(e,[])` contém exactamente os mesmos elementos que `([],List.rev e)` — evitará a devolução da mesma lista na vez seguinte.

Substituímos assim o par de listas por um registo composto de dois campos `entrada` e *saida*, ambos `mutable`:

```
type 'a t = { mutable entrada : 'a list; mutable saida : 'a list }
let create () = { entrada = []; saida = [] }
let push x q = { entrada = x :: q.entrada; saida = q.saida }
exception Empty
```

e o código da função `pop` é apenas mais complexo:

```
let pop q = match q.saida with
    | x :: s ->
      x, { entrada = q.entrada; saida = s }
    | [] -> match List.rev q.entrada with
      | [] ->
          raise Empty
      | x :: s as r ->
          q.entrada <- []; q.saida <- r;
          x, { entrada = []; saida = s }
```

A única diferença está nas duas modificações *in-place* dos campos da fila `q`. Notemos a utilização da palavra chave `as` no filtro que permite nomear a totalidade do valor filtrado por fim a sua reconstrução.

8 Exemplos

8.1 Produto Cartesiano

O exemplo simples seguinte ilustra um uso combinado da criação *on-the-fly* de tuplos, consoante a necessidade, com o mecanismo de filtragem. A função `calcula` recebe uma lista de inteiros e calcula simultaneamente o menor elemento da lista, o maior elemento, a soma dos elementos, o número de inteiros ímpares presentes e o número de pares. A função `calcula2` apresenta uma versão que usa o combinador `fold_left`.

```
let rec calcula l  menor maior soma impares pares =
  match l with
      []     -> (menor,maior, soma, impares, pares)
    | el::li ->
       calcula li  (*li = resto da lista por processar*)
               (**cada parâmetro  representa o estado temporário
                  do conhecimento sobre os valores por calcular.
                  Cada chamada recursiva debruça-se sobre um
                  elemento da lista, logo estes parâmetros são
                  actualizados de acordo *)
               (if el < menor then el else menor)
               (if el > maior then el else maior)
               (soma + el)
               ((if el mod 2 = 0 then 0 else 1) + impares )
               ((if el mod 2 = 0 then 1 else 0) + pares )
;;


let calcula2 l =
fold_left (fun (menor,maior,soma,impares,pares) el ->
  ((if el < menor then el else menor),
   (if el > maior then el else maior),
   (soma + el),
```

```
   ((if el mod 2 = 0 then 0 else 1) + impares),
   ((if el mod 2 = 0 then 1 else 0) + pares )))
((hd l),(hd l),0,0,0) l;;
```

8.2 Brincadeiras a volta da sequência de Fibonacci

Para começar vamos definir a função `fib1` que calcula de forma simples e recursiva o n-ésimo elemento da sequência de Fibonacci. Caso o parâmetro seja negativo, a função deverá lançar uma excepção.

```
open Scanf;;
open Printf;;

(* Excepção *)
exception Input_Negativo;;

let rec fib1 x =
  if (x<0)
  then raise Input_Negativo
  else
    if (x <= 1)
    then 1
    else fib1 (x-1) + fib1 (x-2)
;;
```

O passo seguinte é a definição da função `f1` que se encarrega em pedir um valor adequado ao utilizador.

```
let rec f1 () =
  (** a variável local [valor] recebe um inteiro introduzido pelo utilizador *)
  let valor = print_string " \nVALOR? "; read_int () in
    try
      let res = (fib1 valor) in
printf ("\nResultado 1 ->  fib(%d) = %d\n") valor res
    with Input_Negativo ->
      (print_string "Queira recomeçar...\n";
       f1 ());;

(*execução*)
```

```
f1 ();;
```

A função seguinte, `f2`, obtém o parâmetro da função `fib1` da linha de comando.

```
let f2 () =
  (* [valor] recebe um inteiro extraído da linha de comando *)
  let valor = int_of_string (Sys.argv.(1)) in
  printf "valor lido da linha de comando=%d\n" valor ;
   try
      let res = (fib1 valor) in
(printf "\nResultado 2 ->  fib(%d) = %d\n"  valor res)
    with Input_Negativo ->
     print_string "Queira recomeçar...\n"
;;


(* execução *)
 f2 ();;
```

A função `f3` opta por recolher o parâmetro do ficheiro `dados.txt`. Este deverá ser previamente aberto e associado a um canal de leitura, para ser depois fechado quando o processo terminar.

```
let f3 () =
  (** [fich]: canal de leitura *)
  let fich = open_in "dados.txt" in
  (** [valor] recebe o inteiro extraído da primeira linha do canal*)
  let valor = int_of_string (input_line fich) in
    printf "valor lido do ficheiro=%d\n" valor ;
    begin
      try
let res = (fib1 valor) in
  printf "\nResultado 3 ->  fib(%d) = %d\n"  valor res
      with Input_Negativo ->
print_string "Queira recomeçar...\n"
    end;
    close_in fich;;
```

```
(*execução*)
f3 ();;
```

A função `fib2` calcula de forma iterativa o n-ésimo elemento da sequência de Fibonacci.

```
let fib2 n =
  if (n<0)
  then raise Input_Negativo
  else
  let temp =  ref 0 in
  (* [resa] = fib (n-2) - inicialmente fib 0*)
  let resa =  ref 1 in
  (* [resb] = fib (n-1) - inicialmente fib 1*)
  let resb =  ref 1 in
    for v = 1 to (n-1)  do
    (** Cada iteração obriga a actualização de fib (n-2) e de fib (n-1)*)
      temp:= !resa;
      resa := !resb;
      resb := !temp + !resa
    done;
    (** [resb] = fib n*)
    !resb;;


let rec f4 () =
  let valor = print_string " \nVALOR? " ; read_int () in
    try
      let res = (fib2 valor) in
printf "\nResultado 4 ->  fib(%d) = %d\n" valor res
    with Input_Negativo ->
      (print_string "Queira recomeçar...\n";  f4 ())
;;

(*execução*)
 f4 ();;
```

Nesta versão da função Fibonacci, optamos por uma versão recursiva terminal. Para tal precisaremos de dois acumuladores. De reparar a semelhança

dos parâmetros `acc1` e `acc2` com as referências `resa` e `resb` da versão iterativa. A estratégia de cálculo, como em `fib2` consiste em desenrolar (do inicio até ao fim) os valores da sequência até chegar ao patamar desejado.

```
 (** [acc1] = fib (n-2) - inicialmente fib 0
     [acc3] = fib (n-1) - inicialmente fib 1
     Enquanto [x] decresce, [acc1] e [acc2] vão sendo actualizados.
     Quando [x] <= 1 então [acc2] tem o valor pretendido
  *)
let rec fib3_aux x acc1 acc2 =
  if (x<0)
  then raise Input_Negativo
  else
    if (x <= 1)
    then acc2
    else fib3_aux (x-1) acc2 (acc1+acc2)
;;


let fib3 n = fib3_aux n 1 1;;



let rec f5 () =
  let _ = print_string " \nVALOR? " in
  let valor = read_int () in
    try
      let res = (fib3 valor) in
(printf "\nResultado 5 ->  fib(%d) = %d\n" valor res)
    with Input_Negativo ->
     (print_string "Queira recomeçar...\n"; f5 ())
;;

(*execução*)
 f5 ();;
```

Se juntamos estas variações num só programa obtemos o código seguinte.

Nesta versão o valor pretendido é calculado consoante a configuração da linha de comando e dos parâmetros extraídos.

```
open Arg
open Sys

exception Input_Negativo;;

(** versão recursiva terminal *)
let rec fib_fast x acc1 acc2 =
   if x<0 then raise Input_Negativo
   else
      if x=0 then acc2
  else fib_fast (x-1) acc2 (acc2+acc1)

let fib n = fib_fast n 1 1

let calcula () =
  (** primeiro verificamos quantos argumentos temos na
      linha de comando *)
  let narg = Array.length Sys.argv in

    (** se houver só um (o nome do executável), então, o
        inteiro deve ser proveniente da interacção directa
        com o utilizador *)
    if nargv = 1 then
      begin
        let x = print_string "Queira introduzir um valor inteiro: ";
                read_int() in
          fib x
      end
    else
      (** Se tivermos 2 argumentos, então o segundo argumento é o inteiro *)
      if narg = 2 then
        begin
          fib (int_of_string Sys.argv.(1))
        end
      else
        (** tivermos mais do que 3 argumentos, é erro *)
        if n>3 then failwith "Bad Use\n"
        else
```

```
            (** no caso de haver 3 argumentos e os dois último serem da
                forma [-f ficheiro], então o inteiro está no
                ficheiro [ficheiro] *)
            if ((Sys.argv.(1) = "-f") & (Sys.file_exists (Sys.argv.(2)))) then
              begin
                let fich= open_in Sys.argv.(2) in
                let v = fib (int_of_string (input_line fich)) in
                (close_in fich; v)
              end
            else failwith "Bad Use\n";;

print_int (calcula ());;
```

8.3 Exercitar a estratégia *dividir para conquistar*

Pretendemos aqui calcular a lista de todas as sub-listas de uma lista `l` com os elementos na ordem presente na lista original (ou seja `[a;c]` é sub-lista de `[a;b;c]` mas não `[c;a]`).

A estratégia que pretendemos seguir é a clássica *Dividir para Conquistar*.

O objectivo do presento exemplo é perceber como usar o resultado para uma solução menor para calcular a solução para um problema maior.

Vamos aqui usar recursividade estrutural (tenha em atenção que nem todos os problemas se conseguem resolver com este tipo de recursividade) ou seja perceber como uma solução para `[a1;a2;a3;...;an]` pode contribuir para a solução para `[a0;a1;a2;a3:...;an]`.

Comecemos assim por determinar o resultado para os casos de base: `[]`, e depois para `[c]`, para `[b;c]` e para `[a;b;c]`, etc...

O objectivo é tentar perceber que padrão existe entre a solução dum caso para o caso imediatamente maior. Vamos assim tentar organizar o resultado por forma a facilitar a descoberta deste padrão.

Descoberto, este padrão formará a solução para o caso recursivo.

```
sublista []     = [[]]
sublista [c]    = [[];[c]]
sublista [b;c]  = [[];[c];[b];[b;c]]
sublista [a;b;c]= [[];[c];[b];[b;c];[a];[a;c];[a;b];[a;b;c]]
```

Ou seja

```
sublista el::li =
sublista li @ el::(todos os elementos de (sublista li))
```

O código OCaml subjacente é o seguinte.

```
let rec sublists = function
    []    -> [[]]
  | x::xs ->
      let subl = (sublists xs) in
         subl @  (map (fun l -> x::l) subl)
```

Pretendemos agora calcular a lista de todas as listas possíveis que resultam da inserção de um elemento `e` numa lista `l`.

A função por implementar é: `val insertions e l : 'a -> 'a list -> 'a list list`

Como no exemplo anterior, podemos tentar descobrir o padrão subjacente pela análise dos casos iniciais.

```
insertions e []      = [[e]]
insertions e [c]     = [[e;c];[c;e]]
insertions e [b;c]   = [[e;b;c];[b;e;c];[b;c;e]]
insertions e [a;b;c] = [[e;a;b;c];[a;e;b;c];[a;b;e;c];[a;b;c;e]]
```

logo

```
insertions e el::li =
juntar e::el::li a (el::(todos os elementos de (insertion e li)))
```

Assim o código OCaml é o seguinte:

```
let rec insertions e  =  function
   []    -> [[e]]
 | x::xs -> (e::l) :: (map (fun li -> x::li) (insertions e xs))
```

8.4 Polinómios, e o Método de Horner

Podemos representar um polinómio P de grau n por uma sequência p de reais em que uma célula de índice i representa o coeficiente associado à potência de grau i.

Podemos então calcular $P(x)$ usando o método de Horner, i.e.

$$P_n(x) = (\cdots((p_n x + p_{n-1})x + p_{n-2})x + \cdots + p_1)x + a_0$$

```
open Printf;;

(*
horner r gr x n (pi,pr)=
r: valor do índice de Pr desejado
gr: grau do polinómio
x: valor da variável x
n: grau actualmente por explorar (de 0 a gr)
pi: valor do Pn por calcular
pr: valor de Pr por calcular (se n<=r) ou já calculado (se r<n)

se n > grau
 então devolver  (pi,pr)
 senão seja v um valor real obtido da entrada standard (do teclado)
       seja p o valor  x*pi + v
       se n=r então horner r gr x (n+1) (p,p)
              senão horner r gr x (n+1) (p,pr)
*)

let rec horner r gr x n (pi,pr) =
    if  n> gr then (pi,pr)
    else
      let v = read_float () in
      let p =  x*.pi +. v in
      let new_pr =  if n = r then p else pr in
horner r gr x (n+1) (p,new_pr);;


let grau = read_int () in
let x = read_float () in
let r = read_int () in
let (p_x,pr_x) = horner r grau x 0 (0.0,0.0) in
  (printf "%.3f\n%.3f\n" p_x pr_x);;
```

8.5 Matrizes e ficheiros

Para começar, ilustremos de forma muito simples a redirecção de um canal de leitura para um buffer de leitura, muito prático quando queremos evitar

as habituais dificuldades de leitura via `scanf`.

Assim o objectivo deste pequeno exemplo é ler inteiros dum ficheiro, coloca-los numa lista e calcular para todos eles o factorial.

Sendo a função `leitura` a função mais adequada para tratar dos erros de argumentos que podem surgir (porque esta função é responsável pela extracção dos valores fornecidos à função `fact`), esta tratará da excepção `Minha mensagem` eventualmente lançada pela função `fact`.

```
let leitura name =
  let fich = open_in name in
  let sb = Scanf.Scanning.from_channel fich in
  let l = ref [] in
    try
      while (true) do
let x = Scanf.bscanf sb " %d" (fun a -> a) in
  l:= x::!l
      done; []
    with End_of_file ->
      try
List.map fact List.rev(!l)
      with Minha st ->(
prerr_string ("Problema com dados inválidos: "^st^"\n");[])
;;
```

Vamos neste exemplo introduzir uma utilização básica da leitura e escrita em ficheiro. Pretendemos assim ler uma sequência de inteiros presentes num determinado ficheiro (pedido ao utilizador) e escrever a soma destes inteiros num segundo ficheiro (também previamente escolhido pelo utilizador).

```
  let iname = print_endline "Nome de ficheiro de input"; read_line() in
  let oname = print_endline "Nome de ficheiro de output"; read_line() in

  (** Função auxiliar de leitura.
      Cada inteiro lido é somado a um acumulador.
      O processo é recursivo, e pela definição ingénua do padrão
      recursiva, parece sugerir um padrão recursivo sem fim.
      Tal não acontece visto que os ficheiros são finitos e que
```

```
      necessariamente o processo de leitura irá debater-se contra o
      fim de ficheiro (que ocasionará a excepção End_of_file).
      Caso a leitura corra mal (leitura de um valor que não é inteiro)
      outra excepção será lançada.
      A recuperação destas duas situações excepcionais devolverá o
      valor do acumulador actual.
  *)
  let rec leitura fich acc =
    begin
      try
        let n  = int_of_string (input_line fich) in
        leitura fich (n+acc)
      with _ -> acc
    end
  in
    (** testamos aqui se os identificadores de ficheiros
        introduzidos representam ficheiros válidos*)
    if Sys.file_exists iname && Sys.file_exists oname
    then
      (** Neste caso abrimos os ficheiros e damos inicio ao processo
          de leitura (com o acumulador inicializado a 0)*)
      let ic = open_in iname in
      let oc = open_out oname in
      let v = leitura ic 0 in
begin
          (** Escrevemos a soma no ficheiro destino e fechamos
              a seguir os respectivos canais*)
  output_string oc (string_of_int v);
  close_in ic;
  close_out oc
end
    else failwith "File error"
```

No exemplo seguinte introduzimos um pequeno exercício e ilustramos o uso dos buffers de leitura (*scanning buffer*).

O objectivo é ler uma matriz $n \times n$ de inteiros de um ficheiro (primeiro lê-se o n e depois o conteúdo da matriz) e calcular para cada linha e cada coluna a sua soma. No final calcula-se a soma global dos valores da matriz.

Todos estes valores são colocados numa matriz $(n+1) \times (n+1)$. Por fim, visualiza-se a matriz final.

O estilo imperativo, propositado, segue naturalmente da escolha da matriz como suporte aos cálculos.

```
open Scanf;;
open Printf;;
open Array;;


(** um conjunto de funções que inicializam uma matriz n x n com valores
lidos dum ficheiro e devolve a soma das linhas e colunas **)

let calcula name =
  let fich = open_in name in
  (**  criação dum buffer para scanf (melhor no caso do uso repetido
       de scanf, como vai ser o caso)**)
  let sb = Scanning.from_channel fich in

  (** leitura do tamanho da matriz *)
  let n = int_of_string (input_line fich) in

  (** função auxiliar que lê uma linha de n valores e devolve um vector
      de n+1 posição com os valores lidos mais o 0 final **)
  let fill i =
    if i<n
    then
      Array.init (n+1)
        (fun j -> if j < n then bscanf sb " %d" (fun a -> a) else 0)
    else Array.make (n+1) 0
  in

  (** Inicializar a grelha **)
  let grelha = Array.init (n+1) fill in

  (** cálculo da coluna da direita  **)
  let  _ =
    for i = 0 to n-1 do
```

```
    let acc = ref 0 in
      for j = 0 to n-1 do
        acc := !acc + grelha.(i).(j)
      done;
      grelha.(i).(n) <- !acc
    done in

  (** cálculo da linha final **)
  let  _ =
    for j = 0 to n-1 do
    let acc = ref 0 in
      for i = 0 to n-1 do
        acc := !acc + grelha.(i).(j)
      done;
      grelha.(n).(j) <- !acc
    done in

  (** cálculo do valor extremo grelha.(n).(n) **)
  let _ =
    let i = ref 0 in
    let acc = ref 0 in
      while (!i <= n) do
        acc := !acc + grelha.(!i).(n);
        i:= !i + 1
      done; grelha.(n).(n)<- !acc
  in

    (** visualização**)
    for i = 0 to n do
      for j = 0 to n do
        printf "   %3d"  grelha.(i).(j)
      done;
      print_newline ()
    done
;;

(**** test.txt=2
4
```

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
*****)

calcula "test.txt";;

(**** Resultado
     1     2     3     4    10
     5     6     7     8    26
     9    10    11    12    42
    13    14    15    16    58
    28    32    36    40   136
*****)
```

8.6 Memoização automática

Vamos aqui exemplificar de forma simples a definição ad-hoc de funções recursivas optimizadas com memoização[6].

Relembremos sucintamente que a memoização, para uma função f, é uma técnica de optimização, simples mas eficaz, que consiste no registo numa estrutura adequada de chamadas prévias à função f.

Por exemplo, se pretendemos calcular $f(x)$, o princípio é ver se o resultado de $f(x)$ já não foi previamente guardado na estrutura. No caso positivo, devolvemos directamente o valor (porque esse já foi calculado e arquivado). No caso negativo então procedemos ao cálculo de forma tradicional. Quando este for finalizado, arquivamos o par $(x, f(x))$ na estrutura e devolvemos o resultado $f(x)$. Nesta configuração, se $f(x)$ for novamente requisitado, a estrutura poderá devolvê-lo imediatamente sem necessidade de cálculo.

Vamos exemplificar aqui duas formas de definir funções com memoização. A primeira ilustrará o uso do módulo `Map`.

A estrutura de dados fornecida pelo módulo `Map` é funcional (não é mutável). Fornece assim um mecanismo puramente funcional para a definição de tabelas de associações $(chave, valor)$ implementado com base em árvores binárias de pesquisa equilibradas. É neste tipo de tabela que vamos guardar

[6]Consulte Wikipedia: Memoization para mais detalhes sobre a técnica da memoização.

os valores calculados da sequência de Fibonacci.

As funcionalidade do módulo `Map` é acessível via o functor `Make` que é instanciado por um módulo que estabelece a estrutura do tipo das chaves.

```
(** o functor Make do módulo Map precisa de um conjunto
    ordenado para as suas chaves. Usemos os inteiros. *)
module OrderedInt =
   struct
     type t = int
     let compare n1 n2 = compare n1 n2
   end ;;

(** Tabela de associações com chaves inteiras *)
module AssocInt = Map.Make (OrderedInt) ;;

(** Fibonacci, com memoização - Usamos aqui a técnica baseada no fecho.
    [fib_memo] é a função local [f_m] que tem referência a uma tabela
    de associação.

*)
let fib_memo =
let table = ref AssocInt.empty in
let rec f_m n =
 (** primeiro: ver se [n] já está na tabela.
     No caso positivo devolver a associação encontrada*)
 try AssocInt.find n !table
 with Not_found ->
   (** no caso positivo, calcular o valor de forma clássica,
       recursivamente, e actualizar a tabela de acordo *)
   let ret = if n <= 1 then 1 else f_m (n-1) + f_m (n-2) in
   table := AssocInt.add n ret !table; ret in
f_m;;

(* instantâneo...
# fib_memo 5;;
- : int = 8
# fib_memo 1234;;
```

```
- : int = 424361
*)
```

A desvantagem do método apresentado é a sua ligação forte ao tipo de função definida. Cada caso a sua definição. Obviamente podemos fazer bem melhor e mais genérico.

Vamos agora mostrar uma forma genérica de obter uma função com memoização. Este método foi-nos sugerido por Jean-Christophe Filliâtre.

No lugar de uma tabela de associação, vamos nesta versão preferir tabelas de hash, alteráveis *in-place*.

A função chave nesta definição é a função `memo` que aceita a função alvo da memoização e que devolve assim uma função provida deste mecanismo.

A parte interessante desta versão é a definição da função alvo, se essa tem de ser ligeiramente alterada (para ser alvo da função `memo`), ficar muito próxima da versão que qualquer programador teria escrito de forma natural. Outra vantagem é a sua generalidade. A chave da associação guardada é o tuplo formado pelos parâmetros da função alvo, quaisquer que sejam esses.

```
let memo func =
  let table = Hashtbl.create 999 in
  let rec f v =
    try Hashtbl.find table v
    with Not_found ->
     let ret = func f v in
       Hashtbl.add table v ret; ret
  in
  f;;

let fibm = memo (fun  fib2 n ->
if n <=1 then 1 else (fib2 (n-1) + fib2 (n-2)));;

(* instantâneo igualmente ... Mas muito estranho ;)
# fibm 5000;;
- : int = -126665694
*)

let ackm = memo (fun  ack (m,n) ->
if m <=0
```

```
  then n+1
  else if n <= 0
         then ack ((m-1),1)
         else ack ((m-1),(ack (m,(n-1))))
) ;;

(*** instantâneo, mas nunca estamos longe da explosão....
# ackm (1,7);;
- : int = 9
# ackm (2,8);;
- : int = 19
# ackm (3,6);;
- : int = 509
# ackm (4,6);;
Stack overflow during evaluation (looping recursion?).
*)
```

8.7 Tipo Soma e Indução Estrutural

Vamos neste exemplo ilustrar a proximidade do estilo funcional da programação com a formalização e o raciocínio matemático (indutivo).

Não pretendemos aqui expor um tratado completo da relação entre tipos soma, a indução e a recursão estrutural. Para um tratamento mais formal e completo, queira consultar os manuais "habituais"de matemática discreta, lógica e teoria da computação.

Um exemplo habitual de conjunto definido por indução estrutural é o conjunto das fórmulas da lógica proposicional.

Classicamente tal conjunto assenta na existência de um conjunto numerável (potencialmente infinito) $\mathcal{V}$ de variáveis proposicionais (i.e. $\mathcal{V} = \{A, B, C, \ldots, P, Q, R, \ldots\}$), de duas constantes particulares $\top$ (o valor *verdade*) e $\bot$ (o valor $falso$) e finalmente de um conjunto de conectivas lógicas (de forma *razoável*, $\{\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg\}$).

A definição indutiva do conjunto $\mathcal{P}rop$ das fórmulas proposicionais pode ser dada classicamente da seguinte forma. O conjunto $\mathcal{P}rop$ das fórmulas proposicionais é o menor sub-conjunto X de $(\mathcal{V} \cup \{(,), \bot, \top, \wedge, \vee, \Rightarrow, \Leftrightarrow, \neg\})^*$ tal que

- Caso de Base

- $\mathcal{V} \subseteq X$
- $\{\bot, \top\} \subseteq X$

- Construtores

  - $\forall F \in X, \neg F \in X$
  - $\forall F, G \in X, (F \wedge G) \in X$
  - $\forall F, G \in X, (F \vee G) \in X$
  - $\forall F, G \in X, (F \Rightarrow G) \in X$
  - $\forall F, G \in X, (F \Leftrightarrow G) \in X$

Dois benefícios imediatos da definição por indução estrutural das formulas proposicionais são a existência de um principio de prova por indução (estrutural) e um princípio de recursão (estrutural) útil para a programação sobre estas estruturas.

O *princípio de indução* para as fórmulas proposicionais pode ser descrito resumidamente da seguinte forma.

Seja P uma propriedade sobre os elementos de $\mathcal{P}rop$. Para provar

$$\forall x \in \mathcal{P}rop, P(x)$$

basta provar:

- Casos de Base.

  - $\forall v \in \mathcal{V}, P(v)$
  - $P(\bot)$
  - $P(\top)$

- Construtores

  - $\forall F \in \mathcal{P}rop, se\ P(F)\ então\ \ P(\neg F)$
  - $\forall F, G \in \mathcal{P}rop, se\ P(F)\ e\ P(G)\ então\ \ P((F \wedge G))$
  - $\forall F, G \in \mathcal{P}rop, se\ P(F)\ e\ P(G)\ então\ \ P((F \vee G))$
  - $\forall F, G \in \mathcal{P}rop, se\ P(F)\ e\ P(G)\ então\ \ P((F \Rightarrow G))$
  - $\forall F, G \in \mathcal{P}rop, se\ P(F)\ e\ P(G)\ então\ \ P((F \Leftrightarrow G))$

O mecanismo de definição de funções (estruturalmente) recursivas, designado de *princípio de recorrência* para as fórmulas proposicionais, expõe-se da seguinte forma.

Para definir uma função $f : Prop \to' a$ (total) sobre fórmulas proposicionais, basta saber (fornecer):

- Casos de Base.
  - a função fv que para todo o $v \in \mathcal{V}$ devolve o valor pretendido para a variável v. (ou seja $f(v) = fv(v)$).
  - o valor vb da função em $\bot$ (ou seja $f(\bot) = vb$)
  - o valor vt da função em $\top$ (ou seja $f(\top) = vt$)
- Construtores
  - a função fn que para cada formula F e valor v, se $f(F) = v$ então $f(\neg F) = fn(v)$
  - a função fa que para cada formulas F e G e valores $v1$ e $v2$, se $f(F) = v1$ e $f(G) = v2$ então $f((F \wedge G)) = fa(v1, v2)$
  - a função fo que para cada formulas F e G e valores $v1$ e $v2$, se $f(F) = v1$ e $f(G) = v2$ então $f((F \vee G)) = fo(v1, v2)$
  - a função fi que para cada formulas F e G e valores $v1$ e $v2$, se $f(F) = v1$ e $f(G) = v2$ então $f((F \Rightarrow G)) = fi(v1, v2)$
  - a função fe que para cada formulas F e G e valores $v1$ e $v2$, se $f(F) = v1$ e $f(G) = v2$ então $f((F \Leftrightarrow G)) = fe(v1, v2)$

Este princípio merece uma pequena explicação. Vamos olhar para dois casos particulares, o caso $\top$ e o caso $F \wedge G$.

O que este princípio diz é que

- Se é sabido o valor particular vt da função f no caso de base $\top$ e
- Se, sabendo o valor de f para duas fórmulas quaisquer F e G (digamos que $f(F) = v1$ e $f(G) = v2$), é sabido que o valor de $f((F \wedge G))$ é dado pela função fa, isto é $f((F \wedge G)) = fa(v1, v2)$

(para além dos outros casos, obviamente) então sabe-se calcular o valor desta função f para todas as fórmulas possíveis. Ou seja f fica totalmente definida. Mais, esta termina comprovadamente desde que os seus parâmetros (vt, vb, *etc.*) terminam. De facto, a classe das funções estruturalmente recursivas termina!

Vamos agora apresentar o código OCaml que corresponde ao conjunto das fórmulas proposicionais assim como ilustrar estes conceitos.

```
(*************************************)
(**** Definições Indutivas e OCaml *****)
(*************************************)

type variavel = string;;


type formula =
  | Implica of formula*formula
  | Equivale of formula*formula
  | Ou of formula*formula
  | E of formula*formula
  | Nao of formula
  | Var of variavel
  | Verdade
  | Falso;;

(*exemplo de elemento do conjunto das formulas proposicionais
  (A <-> ~ ( B -> C))
*)

(*exemplo de elemento do conjunto das formulas proposicionais*)
let exemplo = (Equivale ((Var "A"), Nao ( Implica (Var "B", Var "C"))));;


(* Princípio de recorrência para as fórmulas proposicionais *)
let rec formularec
    (fv : variavel -> 'a)
    (vt : 'a)
    (vb : 'a)
```

```
(fn : 'a -> 'a)
(fa : 'a -> 'a -> 'a)
(fo : 'a -> 'a -> 'a)
(fi : 'a -> 'a -> 'a)
(fe : 'a -> 'a -> 'a)
(f: formula)
= match f with
  | Var (v)       -> fv v
  | Verdade       -> vt
  | Falso         -> vb
  | Nao     (p)   -> fn (formularec fv vt vb fn fa fo fi fe p)
  | Ou      (p,q) -> fo (formularec fv vt vb fn fa fo fi fe p)
                        (formularec fv vt vb fn fa fo fi fe q)
  | E       (p,q) -> fa (formularec fv vt vb fn fa fo fi fe p)
                        (formularec fv vt vb fn fa fo fi fe q)
  | Implica (p,q) -> fi (formularec fv vt vb fn fa fo fi fe p)
                        (formularec fv vt vb fn fa fo fi fe q)
  | Equivale(p,q) -> fe (formularec fv vt vb fn fa fo fi fe p)
                        (formularec fv vt vb fn fa fo fi fe q)
  ;;
```

```
(* exemplo de função estruturalmente recursiva sobre as fórmulas
   proposicionais

   Esta função recursiva é estruturalmente recursiva. Porquê? porque
   os argumentos das chamadas recursivas são "explicitamente"
   sub-fórmulas do argumento.

   Outra forma de ver isso: as fórmulas (as definições indutivas) são
   árvores (mais precisamente: termos). Se as chamadas recursivas tem
   sempre lugar sobre uma sub-árvore do argumento inicial, então temos
   uma definição estruturalmente recursiva (i.e. a recursividade
   permitiu descer explicitamente na estrutura do termo em
   parâmetro. Sendo os termos árvores finitas, de certeza que este
   processo chegará a uma folha---> as funções estruturalmente
   recursivas terminam sempre).
```

```
   O seguinte exemplo introduz uma função que conta os número de
   variáveis que ocorrem numa fórmula
*)


(* Comecemos por usar o princípio de recorrência definido
   previamente. Para tal vamos definir as funções e constantes
   necessárias. Estas são óbvias e não merecem uma explicação alongada.
   A contagem para uma variável é 1. Para os outros casos de base, é
   0. A contagem para a conjunção, por exemplo, é a soma da contagem
   de variáveis das suas duas sub-fórmulas, etc.
*)

let  fv = fun v -> 1
let  vt = 0
let  vb = 0
let  fn = fun v -> v
let  fa = fun v1 v2 -> v1 + v2
let  fo = fun v1 v2 -> v1 + v2
let  fi = fun v1 v2 -> v1 + v2
let  fe = fun v1 v2 -> v1 + v2

let rec conta_var form =  formularec fv vt vb fn fa fo fi fe form;;

conta_var exemplo;;
(* devolve 3*)


(* Versão "user-friendly", usando directamente os mecanismos clássicos
   de OCaml*)
let rec conta_var f =
  match f with
      Var x -> 1
    | Verdade | Falso  -> 0
    | Nao g -> conta_var g
    | E (g,h) -> conta_var g + conta_var h
    | Ou (g,h) -> conta_var g + conta_var h
```

```
    | Equivale (g,h) -> conta_var g + conta_var h
    | Implica (g,h) -> conta_var g + conta_var h;;


conta_var exemplo;;
(* devolve 3*)
```

Outras ilustrações possíveis são os seguintes exemplos.

```
(*Outro exemplo de tipo/conjunto indutivo: a listas*)
type 'a lista =
    Vazia | Cons of ('a* 'a lista);;


(*Outro exemplo de tipo indutivo: as árvores binárias*)
type 'a arvore =
   AVazia | Nodo of ('a arvore*'a*'a arvore);;



(*As 5 funções seguintes são estruturalmente recursiva sobre listas
(as 2 primeiras) e sobre árvores (as restantes)*)

let rec length (l:'a lista) =
  match l with
    | Vazia -> 0
    | Cons (x,li) -> 1 + length li;;

let rec concat (l1: 'a lista) (l2: 'a lista) =
  match l1 with
    | Vazia -> l2
    | Cons (x,l) -> Cons (x,concat l l2);;

let rec nodos (a: 'a arvore) =
  match a with
    | AVazia -> 0
    | Nodo (e,x,d) -> 1 + nodos e + nodos d
;;
```

```
(*função auxiliar...*)
let max a b = if a>b then a else b

let rec altura (a : 'a arvore)=
  match a with
    | AVazia -> 0
    | Nodo (e,x,d) -> 1 + (max (altura e) (altura d))
;;


let  rec percurso_infixo (a:'a arvore) =
match a with
    | AVazia -> []
    | Nodo (e,x,d) -> (percurso_infixo e) @ (x::(percurso_infixo d))
;;
```

Após termos visto como programar sobre tipos indutivos, debrucemo-nos agora sobre as demonstrações e sobre os princípios de indução. Além de poder programar, podemos demonstrar que programamos bem e sem bug!

O princípio de indução para as listas pode ser resumido da seguinte forma. Para demonstrar que "qualquer que seja a lista l, $P(l)$" basta:

- (Base) Provar que $P(Vazia)$.
- (Indutivo) Provar que para todo o l lista e x elemento, se $P(l)$ então $P(Cons(x, l))$.

De forma semelhante, o princípio de indução para as árvores é o seguinte. Para demonstrar que "qualquer que seja a árvore ar, $P(ar)$" basta:

- (Base) $P(AVazia)$
- (Indutivo) Provar que para todo os $ar1$ e $ar2$ árvores e x elemento, se $P(ar1)$ e $P(ar2)$ então $P(Nodo(ar1, x, ar2))$.

Ilustremos assim estes dois princípios.

1. Para todas as listas $l1$ e $l2$, demonstrar que $length\ l1\ +\ length\ l2\ =\ length\ (concat\ l1\ l2)$.

   Vamos proceder por demonstração indutiva sobre a lista $l1$.

- Caso de base:
  $length\ Vazia\ +\ length\ l2\ =\ length\ (concat\ Vazia\ l2)$.
  Ora, $length\ Vazia\ =\ 0$
  e, $concat\ Vazia\ l2\ =\ l2$.
  logo $length\ (concat\ Vazia\ l2)\ =\ length\ l2$. Done.
- Passo Indutivo:
  Hipótese de indução:
  Dado uma lista l, $length\ l\ +\ length\ l2\ =\ length\ (concat\ l\ l2)$.
  Verifiquemos agora que para todo o x elemento temos então necessariamente $length\ (Cons\ (x, l))\ +\ length\ l2\ =\ length\ (concat\ (Cons\ (x, l))\ l2)$.
  Vejamos:
  $length\ (Cons\ (x, l))\ +\ length\ l2\ =\ length\ l\ +\ length\ l2\ +\ 1$ (por definição de length)
  $length\ (concat\ (Cons\ (x, l))\ l2)\ =\ length\ Cons\ (x, (concat\ l\ l2))$ (por definição de concat)
  ou seja $=\ 1\ +\ length\ (concat\ l\ l2)\ =\ length\ l\ +\ length\ l2\ +\ 1$ (por hipótese de indução). Done.

Prova concluída ... QED.

2. Para toda a árvore ar, $nodos\ ar \geq altura\ ar$

   Demonstração por indução sobre a árvore ar.

   - (Caso de Base)
     $nodos\ AVazia\ =\ 0\ =\ altura\ AVazia$. Done.
   - (Passo Indutivo)
     Sejam $ar1$ e $ar2$ duas árvores, x um elemento.
     Hipóteses de Indução :
     (a) $nodos\ ar1\ \geq\ altura\ ar1$
     (b) $nodos\ ar2\ \geq\ altura\ ar2$
     Será que temos:
     $nodos\ (Nodo(ar1, x, ar2)) \geq altura\ (Nodo(ar1, x, ar2))$?
     Vejamos.

$nodos\ (Nodo\ (ar1, x, ar2)) = nodos\ ar1 + nodos\ ar2 + 1$ (por definição de nodos)

$altura\ (Nodo\ (ar1, x, ar2)) = 1 + max(altura\ ar1,\ altura\ ar2)$ (por definição de altura)

Dois casos possíveis. a) $altura\ ar1 > altura\ ar2$ ou

b) $altura\ ar1 \leq altura\ ar2$

No caso a) $altura\ (Nodo(ar1, x, ar2)) = 1 + altura\ ar1$ assim sendo, porque $nodos\ ar1 \geq altura\ ar1$ (por hipótese 1) temos $nodos\ (Nodo\ (ar1, x, ar2)) \geq altura\ (Nodo\ (ar1, x, ar2))$. Done.

o caso b) é idêntico.

Assim, a demonstração está concluída.

8.8 Execução de autómatos

Para ilustrar a programação sobre grafos de uma forma simples com base em listas de adjacências, podemos nos debruçar sobre a execução de autómatos de estados finitos deterministas. Um bom exercício pode ser transformar este exemplo por forma a utilizar estruturas de dados mais escaláveis e eficientes.

O leitor poderá reparar no uso da biblioteca `Str` que fornece funções de manipulação de strings e expressões regulares muito convenientes e aqui utilizadas para facilitar a leituras dos dados. A compilação deverá incluir na linha de comando a referência a `str.cma` ou `str.cmxa` consoante o compilador utilizado.

```
open List
open Str
open Scanf
open Printf

(**

Execução de autómatos deterministas

*)

(**
```

```
   O tipo [simbolo] representa o tipo das letras (o alfabeto -
   presentes nas fitas mas também nas transições). Aqui fixamos o tipo
   char como sendo o alfabeto usado nestes autómatos.

*)
type simbolo = char


(**
    a fita de entrada é simplesmente uma lista de símbolos
*)
type fita =  simbolo list

(**

   Escolhemos representar os estados por inteiros.  Em particular
   tentaremos respeitar o invariante seguinte sobre a representação dos
   estados: Se houver [n] estados então estes são [0 1 2 3 .. n-1].

*)
type estado = int

(**

   As transições [q1 --a--> q2] são representadas como [((q1,a),q2)] ao
   detrimento da representação mais natural [(q1,a,q2)].

   Esta pequena "nuance" permite uma pesquisa melhorada de uma transição
   no conjunto das possíveis transições (de tipo hash table, em que a chave é
   [(q1,a)] e o conteúdo é estado destino [q2])

*)
type transicao =  ((estado*simbolo)*estado)

(**

   Neste cenário, um autómato (ou máquina) é dado pela relação de
```

```
   transição (a listas de adjacência, ou seja a lista das transições),
   *o* estado inicial e o conjunto dos estados finais. Sendo que o
   alfabeto e conjunto dos estados se deduz automaticamente dos dados
   anteriores.
*)
type maquina =  (transicao list * estado * estado list)

(**

   As configurações da máquina (determinista), aqui designada de
   memória, é simplesmente o par *do* estado actualmente activo (onde
   a máquina se encontra no momento a execução) e o buffer que resta
   ainda por processar.

*)
type memoria = (estado * fita)

(**
uma excepção para assinalar o fim de uma execução
*)
exception FIM of memoria



(**

    [next] calcula o próximo estado, ou seja o estado [q] destino da
    transição [esta---simb--->q], se esta transição existir. Senão
    (i.e. a excepção [Not_found] foi lançada) esta situação configura
    o fim da execução.

*)
let next simb maquina memo =
  let (esta, restante) = memo in
  try
    let transicoes,b,c = maquina in
    (assoc (esta,simb) transicoes)
  with Not_found -> raise (FIM memo)
```

```
(**
   [step] realiza um passo de execução do autómato [maq] a partir da
   configuração [memo].
*)
let step memo maq =
  let (aqui, restante) = memo in
  (** se o buffer de entrada for vazio, então acabou, senão tratamos
      do primeiro caracter do buffer. Ou seja, vamos ver que novo
      estado atingimos com este caracter a partir do estado
      actualmente activo (onde a execução actualmente se encontra, o
      estado [aqui]). Chamamos aqui a função [next] que trata deste
      cálculo.  *)
  match restante with
      [] ->  raise (FIM memo)
    | el::li ->  (next el maq memo,li)

(** [is_accepted] é um predicado que detecta se uma configuração
    [memo] da execução do autómato [maq] prefigura a aceitação. Ou seja o
    buffer de entrada encontra-se vazio e há pelo menos um estado final na
    configuração
*)
let is_accepted memo maq =
  let (aqui, restante) = memo in
  let (trans,init,accept)= maq in
  (mem aqui accept)&&(restante=[])

(** funções utilitárias simples*)
let em_par a b c =  ((a, b),c)

let char_of_string s = s.[0]

(**
Lê no formato texto a palavra por reconhecer e o autómato por executar.
A leitura devolve as estruturas correspondentes.
```

```
*)
let leitura () =
  let dados = map (fun x -> char_of_string x)
    (Str.split (Str.regexp "[ \t]+")  (read_line ())) in
  let initl = read_int () in
  let finl = map (fun x -> int_of_string x)
    (Str.split (Str.regexp "[ \t]+")  (read_line ()))  in
  let input = ref [] in
    try
      while (true) do
input:= (scanf " %d %c %d " em_par)::!input
      done; (dados,(!input,initl,finl))
    with _ -> (dados,(!input,initl,finl))

(** a função [print_output] analisa a configuração final e imprime na
    saída standard o veredicto.
*)
let print_output memo maq=
    if (is_accepted memo maq)
    then printf "YES\n"
    else printf"NO\n"


let main () =
  let dados,maquina = leitura () in
  let (a,b,c) = maquina in
    try
      let memor = ref (b,dados)  in
(** Enquanto houver passos de execução por realizar, executar.  A
    excepção [FIM] é lançada para assinalar o fim da
    execução.  *)
      while true do
memor := (step !memor maquina)
      done
    with
FIM x -> print_output x maquina
;;
```

```
main ();;

(**   exemplo de entrada:
a a b a
0
1 2
0 a 0
0 b 1
0 a 3
1 a 2
2 a 3
3 a 1
3 a 2
*)
```

8.9 Parsing com fluxos e computação simbólica

Neste exemplo, ilustramos as facilidades que a linguagem OCaml, à semelhança das linguagens do mesmo paradigma, nos oferece para a programação simbólica. Um exemplo clássico é assim a unificação de termos de primeira ordem. Utilizaremos o formato clássico dos concursos de programação ACM para a apresentação do enunciado (do exercício).

O objectivo deste exercício é a implementação de um dos mais famosos e importantes algoritmo em informática: o algoritmo de unificação de primeira ordem. Historicamente o primeiro estudo formal e algoritmo para este problema foram definidos e apresentados por Robinson em 1965 (*Robinson J. A. [1965], 'A machine oriented logic based on the resolution principle', J. of the ACM 12(1), 23-41*) e permitiu avanços fundamentais em Ciência da Computação, Lógica, Inteligência Artificial, Demonstração por Computador e até mesmo em Compiladores. O problema é no entanto relativamente simples de enunciar e trata essencialmente de encontrar formas de tornar dois termos, digamos u e v, iguais via uma instanciação adequada, designada de *substituição*, das variáveis presentes em cada termo. Mais formalmente esta substituição, digamos ρ, permite identificar os dois termos, i.e. $\rho(u) = \rho(v)$. Tais substituições, quando existam, são chamadas de $unificadores$.

Para clarificar as ideias, imagine que se pretenda unificar $f(f(X, a), g(f(b, Y)))$ com $f(Y, g(Z))$. (note que os identificadores em maiúsculas são variáveis (i.e. podem ser instanciadas) e os outros identifica-

dores são funções ou constantes (designados de *átomos* neste contexto)). A notação utilizada é:

$$f(f(X,a),g(f(b,Y))) \stackrel{?}{=} f(Y,g(Z))$$

A resposta é sim. De facto se se escolher a substituição $\rho = \{Y := f(X,a); Z := f(b,f(X,a))\}$ então $\rho(f(f(X,a),g(f(b,Y)))) = f(f(X,a),g(f(b,f(X,a))))$ (o Y foi substituído pelo valor indicado na substituição) e $\rho(f(Y,g(Z))) = f(f(X,a),g(f(b,f(X,a))))$ (o Y e o Z foram substituídos pelos valores indicados em ρ). Logo $\rho(f(f(X,a),g(f(b,Y)))) = \rho(f(Y,g(Z)))$.

No contexto deste exercício estamos interessado no que chamamos os **unificadores principais** (*most general unifier*, em inglês). Por exemplo os dois termos X e $f(Y)$ tem vários unificadores: $\{X := f(Y)\}$, mas ainda $\{X := f(a); Y := a\}$ ou $\{X := f(f(b)); Y := f(b)\}$. No entanto só $\{X := f(Y)\}$ nos interessa aqui por esse unificador ser o mais geral de todos.

Apresentamos aqui um algoritmo para a unificação (uma versão melhorada do algoritmo original apresentado pelo próprio Robinson): o algoritmo de *Martelli-Montanari*.

Este algoritmo processa conjuntos finitos de equações

$$E = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$$

e uma substituição θ. Inicialmente $E = \{s \stackrel{?}{=} t\}$, onde s e t são os dois termos por unificar, e $\theta = \emptyset$. então o algoritmo escolhe arbitrariamente uma equação de E (por exemplo se este conjunto for codificado por uma lista, a escolha pode ser o primeiro elemento da lista). Esta equação é processada da seguinte forma:

	Equação retirada de E	Acção por realizar
1	$f(s_1,\ldots,s_n) \stackrel{?}{=} f(t_1,\ldots,t_n)$	substituir em E pelas equações $s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n$
2	$f(s_1,\ldots,s_n) \stackrel{?}{=} g(t_1,\ldots,t_n)$	falhar
3	$X \stackrel{?}{=} X$	retirar a equação de E
4	$X \stackrel{?}{=} t$ ou $t \stackrel{?}{=} X$ onde X não occorre em t	juntar $X := t$ a θ, aplicar a substituição $\{X := t\}$ a E e ao termos de θ e finalmente retirar esta equação de E
5	$X \stackrel{?}{=} t$ ou $t \stackrel{?}{=} X$ onde X occorre em t e $X \neq t$	falhar (o chamado occur check)

O procedimento deve ser repetido até E ficar vazio ou este falhar.

Vários algoritmos para a unificação (além do algoritmo apresentado) podem ser encontrados em `http://lat.inf.tu-dresden.de/research/papers/2001/BaaderSnyderHandbook.ps.gz` ou ainda `http://www.dcs.kcl.ac.uk/staff/endriss/cs3aur-02/pdf/unification.pdf` (google é aqui e mais uma vez o vosso amigo....).

Aqui vão alguns exemplos (retirados do *wikipedia*) :

- $A \stackrel{?}{=} A$: unificação bem sucedida (a substituição é vazia, já que os termos são iguais)
- $A \stackrel{?}{=} B, B \stackrel{?}{=} abc$: unificação bem sucedida, $\rho = \{A := abc; B := abc\}$
- $xyz \stackrel{?}{=} C, C \stackrel{?}{=} D$: a unificação é simétrica, ou seja $\rho = \{C := xyz; D := xyz\}$
- $abc \stackrel{?}{=} abc$: unificação bem sucedida (a substituição é vazia, já que os termos são iguais)
- $abc \stackrel{?}{=} xyz$: a unificação falha já que os átomos são diferentes
- $f(A) \stackrel{?}{=} f(B)$: unificação bem sucedida, $\rho = \{A := B\}$
- $f(A) \stackrel{?}{=} g(B)$: a unificação falha visto que $f \neq g$
- $f(A) \stackrel{?}{=} f(B, C)$: a unificação falha já que f apresenta aridades diferentes
- $f(g(A)) \stackrel{?}{=} f(B)$: unificação bem sucedida, $\rho = \{B := g(A)\}$
- $f(g(A), A) \stackrel{?}{=} f(B, xyz)$: unificação bem sucedida, $\rho = \{A := xyz; B := g(xyz)\}$
- $A \stackrel{?}{=} f(A)$: Unificação infinita. A é unificada com $f(f(f(f(\cdots))))$. Em lógica de primeira ordem esta situação é proibida, e neste exercício também. Este caso é detectado verificando que a substituição $A := f(A)$ tem A a ocorrer em ambos os lados. Este teste é designado de *occurs check*.
- $A \stackrel{?}{=} abc, xyz \stackrel{?}{=} X, A \stackrel{?}{=} X$: a unificação falha porque de facto $abc \neq xyz$.

Input

O input organizado em duas linhas. Na primeira linha é apresentada o primeiro termo, na segunda linha o segundo termo. Por convenção as funções e as constantes são introduzidas por identificadores sem maiúsculas e as variáveis são introduzidas por identificadores em letras maiúsculas.

Output

A palavra "NO" se os dois termos não se unificam. A palavra "YES" caso contrário. Neste caso a lista das variáveis instanciadas é apresentada uma por linha, no formato `nome da variável = termo"`. Esta lista é ordenada pelo nome das variáveis.

Sample Input 1

```
f(a,g(Y))
f(Y,X)
```

Sample Output 1

```
YES
X = g(a)
Y = a
```

Sample Input 2

```
f(X,g(b,h(X)))
f(X,g(b,h(X)))
```

Sample Output 2

```
YES
```

Sample Input 3

```
f(X,g(b,h(X)))
f(a,g(X,h(X)))
```

Sample Output 3

```
NO
```

Sample Input 4

```
f(X,g(b,h(X)))
f(b,g(X,t(X)))
```

Sample Output 4

```
NO
```

Uma solução pode ser a seguinte. O leitor reparará no uso da biblioteca `Genlex` e no uso dos fluxos. Para essa ultima o leitor deverá incluir no comando de compilação as devidas opções de compilação.

```
(** Unificação de termos de primeira ordem -
Inspirada de "Term Rewriting and All That -
Franz Baader and Tobias Nipkow -
http://www4.in.tum.de/~nipkow/TRaAT/**)

open List
open Genlex
open Str
open String
open Char
open Printf

type vname = string

type term = V of vname | T of (string * (term list))

type subst = (vname * term) list

(* indom: vname -> subst -> bool *)
(* Testa se a variável x está na substituição s *)
let indom x s = exists (fun (y,_) -> x = y) s
```

```
(* app: subst -> vname -> term *)
(* Aplica uma substituição a uma variável. *)
(* Isto é: se existe "x:= t" em substit
   então (app substit x) devolve t  *)
let rec app substit  x  =
match substit with
    [] -> V x
  | ((y,t)::s) ->   if x=y then t else app s x

(* lift: subst -> term -> term *)
(* A função lift generaliza a função app. *)
(* Ou seja: lift s t devolve s(t). i.e.
 devolve t em que todas as variáveis que estão
em s foram substituídas pelo valor indicado em s   *)
let rec lift s = function
    (V x)   -> if indom x s then app s x else V x
  | (T(f,ts)) -> T(f, map (lift s) ts)

(* occurs: vname -> term -> bool *)
(* (occurs x t) testa se a variável x occorre em t *)
let rec  occurs x = function
    (V y)     -> x=y
  | (T(_,ts)) -> exists (occurs x) ts



exception UNIFY

(* solve: (term * term) list * subst -> subst *)
(* Função principal.
   solve E,s -->  E é o conjunto de equações por explorar *)
(*           -->  s é a substituição actualmente calculada *)
(* Algoritmo utilizado: o exposto no enunciado            *)
(* Olhamos para a primeira equação de E,                  *)
(* 4 casos gerais  ----->                                 *)
(* 1) E está vazio                                        *)
(* 2) equação da forma X = termo                          *)
(* 3) equação da forma termo = X                          *)
```

```
(* 4) equação da forma termo1 = termo2                    *)
(* No final: *)
let rec  solve =  function
    ([], s) -> s
  | ((V x, t) :: li, s) ->
      if V x = t then solve(li,s) else elim(x,t,li,s)
  | ((t, V x) :: li, s) ->  if V x = t then solve(li,s) else elim(x,t,li,s)
  | ((T(f,ts),T(g,us)) :: li, s) ->
      if f = g
      then
if List.length ts = List.length us
then
  solve((combine ts us) @ li, s)
else raise UNIFY
      else raise UNIFY

(* elim: vname * term * (term * term) list * subst -> subst *)
and elim (x,t,su,s) =
      if occurs x t then raise UNIFY
      else let  xt = lift [(x,t)]
           in solve(map (fun (t1,t2) -> (xt t1, xt t2)) su,
                    (x,t) :: (map (fun (y,u) -> (y, xt u)) s))


(* unify: term * term -> subst *)
(* invoca a função principal de unificação: a função solve*)
(* Inicialmente E={t1=t2} *)
let unify(t1,t2) = solve([(t1,t2)], [])

let maiuscula s = let v = s.[0] in
      ((code v >= code 'A') && (code v <= code 'Z'))

let trata s e =
  match e with
      [] ->
if maiuscula s then V s
else T (s,e)
    | _ -> T (s,e)
```

```
(**
Vamos construir um lexer simples, com base no lexer genérico
fornecido pela biblioteca Genlex.
A função make_lexer precisa, para construir tal lexer, que lhe
seja fornecido o conjunto de palavras chaves (que serão distinguidas
dos identificadores) *)
let lexer = make_lexer ["("; ")"; ","]


(*  Construção do parser com base nos fluxos e num parser descendente  LL(1)

A gramática LL(1)  é:

E   ::= id F
F   ::= \epsilon
F   ::= (E EL)
EL  ::= ,E EL
EL  ::= \epsilon
*)

let rec parse_expr = parser (* corresponde a entrada E da gramática *)
[< 'Ident s; e = parse_F>] -> trata s e
and parse_F = parser (* corresponde a entrada E' da gramática *)
[< 'Kwd "("; e = parse_expr; el = parse_EL; 'Kwd ")" >] -> e::el
| [< >] -> []
and parse_EL = parser (* corresponde a entrada T' da gramática *)
[< 'Kwd ","; e = parse_expr; el = parse_EL >] -> e::el
| [< >] -> []



let parse_expression = parser [< e = parse_expr; _ = Stream.empty >] -> e;;

(* função principal de leitura usando streams*)
let expression_of_string s = parse_expression(lexer(Stream.of_string s));;

let rec string_of_expression e =
```

```
match e with
    V s -> s
  | T (e,el) -> (e^(if el = [] then "" else "("^(aux el)^")"))
and   aux l =
    match l with
[] -> ""
      | [e] ->  (string_of_expression e)
      | e::li ->
  ( (string_of_expression e)^","^aux li)


let main () =
  let e1 = (expression_of_string (read_line ())) in
  let e2 = (expression_of_string (read_line ())) in
    try
      let  unific =(List.sort
      (fun (a,b) (c,d) -> String.compare a c)
                      (unify (e1,e2))) in
print_string "YES\n";
(List.iter (fun x -> printf "%s\n" x)
   (map (fun (x,e) -> x^" = "^(string_of_expression e)) unific))
    with UNIFY -> printf "NO\n";;


main ()
```

Referências

[1] D. Bagley. The great computer language shootout. `http://www.bagley.org/~doug/shootout`

[2] E. Chailloux, P. Manoury, and B. Pagano. Developing applications with objective caml. `http://caml.inria.fr/oreilly-book`, 2003.

[3] G. Cousineau and M. Mauny. *The functional approach to programming.* Cambridge University Press, 1998.

[4] J.-C. Filliâtre. Initiation à la programmation fonctionnelle. Master Informatique M1, Université Paris-Sud. Available at `http://www.lri.fr/~filliatr/m1/cours-ocaml.en.html`, 2006.

[5] J. Hickey. Introduction to objective caml. Under publication, available at `http://files.metaprl.org/doc/ocaml-book.pd`, 2008.

[6] X. Leroy. Functional programming and type systems. Master Parisien de Recherche en Informatique (MPRI), Paris, available at `http://gallium.inria.fr/~xleroy/mpri/2-4/`, 2011-2012.

[7] L. Pequet. Programmation fonctionnelle - introduction illustrée en objective caml. `http://www-rocq.inria.fr/~pecquet/pro/teach/teach.html`, 2002.

[8] D. Rémy. *Understanding, and Unraveling the OCaml Language*, volume LNCS 2395 of *Applied Semantics. Advanced Lectures*, chapter ?, pages 413–537. Springer-Verlag, 2002. `http://pauillac.inria.fr/~remy/cours/appsem/ocaml.pdf`

[9] OCaml Development Team. *The Objective Caml system:Documentation and user's manual*, 2002. `http://caml.inria.fr/ocaml/htmlman/index.html`

[10] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993. Available at `caml.inria.fr/pub/distrib/books/llc.pdf`