

Laborator 1

Ce este un interpretor?

Un interpretor este un program simplu de linie de comandă, care permite utilizatorului să scrie o comandă. Comanda va fi evaluată și rezultatul evaluării va fi afișat utilizatorului.

Primele comenzi

După ce am [pornit interpretorul](#), putem scrie comenzi care vor fi executate de acesta. Ce mai simplă comandă pe care o putem scrie este o simplă valoare. Spre exemplu, putem scrie valoare `1` la linia interpretorului, urmată de caracterele `;;` (Vom discuta mai mult despre `;;` într-un paragraf viitor, important de înțeles este că `;;` va spune interpretorului că am terminat de scris comanda și dorim ca acesta să evalueze ce am scris și să ne dea rezultatul). Interpretorul va răspunde cu aceeași valoare (fiind doar o valoare simplă interpretorul nu are de făcut nici o procesare suplimentară) și va include și tipul valorii.

Pentru `1` tipul valorii va fi întreg (mulțimea \mathbb{Z} din matematică) sub denumirea de `int`.

```
# 1;;  
- : int = 1
```

Puțin mai complicat putem scrie o expresie, de exemplu `1+2`. De data aceasta, interpretorul va evalua expresia și ne va da rezultatul `3`. De asemenea, pe lângă valoare, va fi inclus și tipul valorii (tot tipul întreg).

```
# 1+2;;  
- : int = 3
```

Pentru numere întregi, avem următorii operatori pe care putem să îi folosim în expresii: `+`, `-`, `/`, `*`, `mod` (`mod` este restul împărțirii întregi).

Variabile

Acum că știm să folosim valori și expresii, să vedem cum putem reține rezultatele.

Putem reține un rezultat folosind o **declarație de variabilă**. Asemănător cu limbajului matematic, în care am spune „fie $x = 10$ ”, în OCaml declarăm o variabilă astfel:

```
let x = 10
```

(„let” în engleza înseamnă „fie”)

```
# let x = 10;;  
val x : int = 10
```

Dacă scriem declarația de mai sus în interpretor, acesta va răspunde cu `val x:int = 10`. La fel ca și o valoare, sau ca și rezultatul unei expresii, și o variabilă are un tip. Tipul variabilei este determinat de valoarea care îi este atribuită prin declarație. În exemplul anterior, tipul variabilei este tot întreg (`int`).

Putem atribui unei variabile și o expresie, acesta fiind evaluată și rezultatul va fi reținut în variabilă:

```
# let y = 10 + 12;;  
val y : int = 22
```

Unde putem folosi variabilele? Putem folosi o variabilă oriunde am putea folosi o valoare de același tip. Când expresia este evaluată, valoarea care fi folosită pentru o variabilă este valoarea pe care am atribuit-o la declarația variabilei:

```
# let y = 10 + 12;;  
val y : int = 22  
# let z = y + 10;;  
val z : int = 32
```

Nota: Cei care au mai folosit alte limbaje de programare ar putea să fie surprinși de faptul că **variabilele în OCaml nu pot să fie modificate, după ce le-am atribuit o valoare atunci când le-am definit**. Cu toate că acest lucru pare limitant la început, putem scrie programe fără capacitatea de a modifica variabilele, având totodată anumite beneficii (vom avea mai puține elemente în mișcare). Cu toate acestea, o variabilă în OCaml poate avea valori diferite de la o rulare la alta, dacă depinde de condiții externe (de exemplu, după cum vom vedea, o variabilă poate depinde de o comandă a utilizatorului).

Tipul Real

Până acum am folosit doar valori și variabile întregi. OCaml suportă și valori reale:

```
# let x = 1.2;;  
val x : float = 1.2
```

Tipul real în OCaml este denumit **float** (numele provine de la modul de reprezentare al valorii pentru numere reale, și anume reprezentarea în [virgulă flotantă](#)).

Operatorii pentru numere reale sunt diferiți față de cei pe numere întregi. Există variante reale ale operatorilor aritmetici. Acestea sunt **+. , -. , *. , /.** După cum observăm, operatorii reali se construiesc prin adăugarea unui **.** după operatorul pentru numere întregi (Excepție face operatorul **mod** , care nu își are sensul în contextul numerelor reale)

Când vrem să facem o operație aritmetică cu numere reale, trebuie să folosim variantele reale ale operatorilor:

```
# let r = 1.2 +. 1.3;;  
val r : float = 2.5
```

Nu putem să amestecăm valori întregi cu valori reale, sau operatori întregi cu valori reale. Vom analiza, în continuare, câteva exemple:

```
# let r = 1.2 + 1.3;;  
Characters 8-11:  
  let r = 1.2 + 1.3;;  
          ^^^  
Error: This expression has type float but an expression was expected of type  
       int
```

Codul de mai sus folosește două valori reale (**1.2** și **1.3**), împreună cu un operator pentru întregi (**+**). Interpretorul ne va avertiza că tipul găsit este **float** (din cauza valorii **1.2**), dar valoarea așteptată este de tip **int** (din cauza operatorului **+**).

```
# let r = 1.2 +. 1;;
Characters 15-16:
  let r = 1.2 +. 1;;
                  ^
Error: This expression has type int but an expression was expected of type
float
#
```

Codul de mai sus folosește un operator pentru numere reale (+.) cu o valoare întreagă (1). Interpretorul ne va spune că expresia are tipul `int` (din cauza valorii 1) dar tipul așteptat este `float` (din cauza operatorului +.)

Ultima eroare ne duce cu gândul la întrebarea „cum putem aduna o valoare întreagă la o valoare reală?”. Avem două posibilități. O posibilitate este să scriem valoarea întreagă ca valoare reală, prin adăugarea părții fracționare zero (ex: `1.0`) sau, mai simplu, adăugând un simplu `.` (ex: `1.`).

```
# let r = 1.2 +. 1.;;
val r : float = 2.2
```

A doua posibilitate este să apelăm o funcție care convertește o valoare întreagă într-o valoare reală, după cum vom vedea mai departe.

Citirea erorilor

După cum am văzut mai sus dacă interpretorul întâlnește o eroare în codul pe care l-a scris nu îl va putea executa și va raporta eroarea pe care a găsit-o. Elementele raportate în eroare, în ordinea în care apar sunt:

1. Numărul relativ la începutul comenzii curente al caracterul de început și de sfârșit unde a fost întâlnită eroarea
2. Textul liniei care cauzează eroarea
3. Caractere ^ care arată spre bucata de cod care conține eroarea din linia de mai sus
4. Mesajul de eroare.

```
# let a = 10
  let b = 1 +. 1;;
Characters 19-20: 1
  let b = 1 +. 1;; 2
                  ^ 3
Error: This expression has type int but an expression was expected of type 4
float
```

Folosirea Funcțiilor

După cum știm de la matematică, o funcție asociază unei valori din domeniul de definiție a funcției o valoare din codomeniu. Și în OCaml o funcție face același lucru.

Să luăm drept exemplu o funcție predefinită, funcția `float_of_int`. Această funcție va asocia unei valori de tip întreg o valoare egală, dar de tip real. Pentru a folosi funcția trebuie să scriem numele funcției, urmat de valoarea pe care dorim să o convertim.

```
# float_of_int 1;;
- : float = 1.
```

După cum vedem, când folosim funcția pentru valoarea 1, interpretorul va răspunde cu tipul rezultatului (`float`) și valoarea `1.`

Terminologie: Ceea ce am făcut mai sus se numește „apelarea” unei funcții. Mai puteți întâlni și termenul de „aplicarea” unei funcții. Valoarea `1` se numește „argumentul” funcției. Putem spune că am „pasat” funcției `float_of_int` argumentul `10`. Despre rezultat (valoare `1.`) spunem că a fost „returnată” de funcția `float_of_int`.

O funcție poate avea mai multe argumente separate prin spațiu. Spre exemplu, funcțiile predefinite `min` și `max` acceptă două argumente și returnează minimum respectiv maximum dintre cele două.

```
# min 10 11;;  
- : int = 10  
# max 10 11;;  
- : int = 11
```

Cum putem folosi valoarea returnată de o funcție?

O putem folosi în mai multe feluri. Cel mai simplu l-am văzut deja: este acela de a apela funcția în interpretor și de a vedea rezultatul.

O altă opțiune este să atribuim valoarea returnată de funcție la o variabilă, pe care putem mai departe să o utilizăm în alte calcule:

```
# let m = min 10 11;;  
val m : int = 10  
# m - 1;;  
- : int = 9
```

Putem să folosim funcția ca parte dintr-o expresie mai mare:

```
# 1 + min 0 2;;  
- : int = 1  
#
```

Până acum, argumentele funcțiilor pe care le-am folosit au fost doar valori. Putem folosi și variabile, definite anterior, pe post de argumente:

```
# let x = 10  
    let y = 11  
    let m = min x y;;  
val x : int = 10  
val y : int = 11  
val m : int = 10  
#
```

Un argument poate, de asemenea, să fie și o altă expresie, dar va trebui să punem în paranteze argumentul care este o expresie:

```
# let x = 10
  let y = 11;;
val x : int = 10
val y : int = 11
# let m = min (x - 5) (y + 2);;
val m : int = 5
```

Motivul pentru care avem nevoie de paranteze este dat de precedența operatorilor, pe care îi vom discuta în următorul capitol.

Prioritatea și asociativitatea operatorilor

[Prioritatea](#) și [asociativitatea](#) operatorilor determină cum vede limbajul o expresie pe care o scriem.

În continuare, avem un tabel cu prioritatea operatorilor (operatorii de la începutul tabelului au prioritate mai mare decât cei de mai jos) precum și cu asociativitatea acestora. Tabelul de mai jos include operatorii comuni pe care i-am văzut deja (sau pe care îi vom vedea) în decursul acestui curs.

Dacă doriți să vedeți un tabel complet, puteți să consultați [specificația limbajului](#) sau puteți să consultați cartea „[Real World OCaml](#)”.

Operator	Asociativitate
Aplicarea unei funcții	stânga
- -. (prefix, adică minusul unar, ex: -2 , -. 2.)	–
** (operatorul exponențial, $2 ** 3 = 2^3$)	dreapta
* * . / / . Mod	stânga
+ + . - -.	stânga
::	dreapta
= < <= > >= !=	stânga
&&	dreapta
 	dreapta
if	–
;	dreapta
let match fun function try	–

Tabel 1: Prioritatea și asociativitatea operatorilor

Impactul asociativității

Să vedem cum afectează asociativitatea modul de evaluare a unei expresii care conține mai mulți operatori **-**:

Ex: **1 - 2 - 3**

Există două moduri de a evalua această expresie, cu rezultate diferite:

Asociativ la stânga: **(1 - 2) - 3 // rezultat -4**

Asociativ la dreapta: **1 - (2 - 3) // rezultat 2**

Conform tabelului de mai sus, știm că OCaml tratează operatorul `-` ca fiind asociativ la stânga, deci vom ști că modul de evaluare este primul și rezultatul va fi `-4`.

```
# 1 - 2 - 3;;  
- : int = -4
```

Să vedem în continuare operatorul exponențial `**`:

Ex: `2. ** 2. ** 3.`

Din nou, există două moduri în care putem evalua această expresie, cu rezultate diferite:

Asociativ la stânga: `(2. ** 2.) ** 3. // rezultat 64` Sau matematic: $(2^2)^3$

Asociativ la dreapta: `2. ** (2. ** 3.) // rezultat 256` Sau matematic $2^{(2^3)}$

Din nou, consultând tabelul Tabel 1: Prioritatea și asociativitatea operatorilor”, vedem că asociativitatea operatorului exponențial este la dreapta, deci modul în care expresia este evaluată este al doilea:

```
# 2. ** 2. ** 3.;;  
- : float = 256.
```

Impactul priorității operatorilor

Dacă avem mai mulți operatori cu priorități diferite, într-o expresie, prioritatea dictează care operatori sunt evaluați mai întâi:

Ex: `1 + 2 * 3;`

Evident expresia de mai sus este evaluată (ca în matematică astfel): `1 + (2 * 3)`

Modul de evaluare este mai puțin evident în momentul în care avem o apelare de funcție:

Ex: `float_of_int 1 + 1`

Modul greșit de a vedea expresia: `float_of_int (1 + 1)`

Modul în care OCaml evaluează expresia `(float_of_int 1) + 1`

Motivul pentru care OCaml evaluează expresia în acest fel este deoarece aplicarea unei funcții are cea mai mare prioritate (este prima din tabel)

Acest mod de evaluare duce la o eroare în acest caz, deoarece rezultatul lui `(float_of_int 1)` este de tip `float`, dar acest rezultat este folosit împreună cu operatorul `+` care este un operator pentru numere întregi:

```
# float_of_int 1 + 1;;  
Characters 1-15:  
float_of_int 1 + 1;;  
^^^^^^^^^^^^^^^^  
Error: This expression has type float but an expression was expected of type  
      int
```

Funcții pentru numere reale / întregi

Acum că știm cum să folosim funcții să vedem câteva funcții folositoare pentru numere reale/întregi

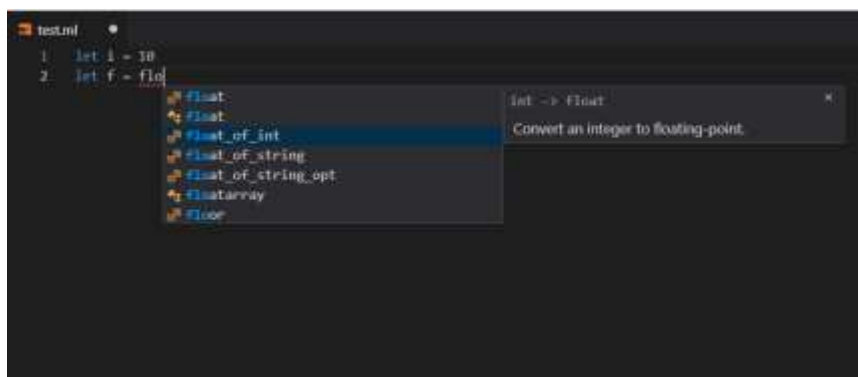
Folosirea fișierelor pentru cod

Până acum am scris cod direct în interpretor, ceea ce este util pentru a vedea rezultatul imediat, dar are și dezavantaje:

- Este dificil să scriem programe mai lungi
- Nu avem nici un ajutor din partea editorului când scriem cod
- Vedem erorile doar la execuție, nu în timp ce scriem codul
- Nu putem salva codul pentru a-l consulta/modifica mai târziu

Pentru început, trebuie să avem un director deschis (dacă nu știți cum vedeți [aici](#)) și să creăm un fișier cu extensia **.ml** (dacă nu știți cum, vedeți [aici](#)).

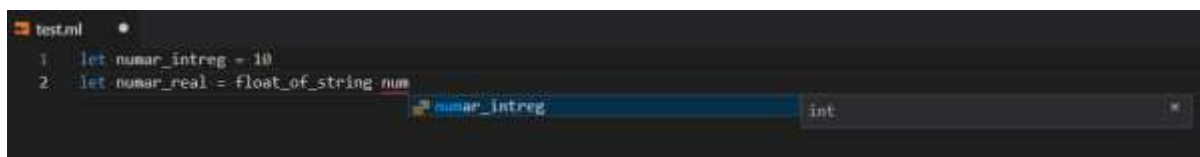
Când scriem în fișier, Visual Studio Code ne va ajuta cu o listă de posibile simboluri (nume de variabile, nume de funcții, nume de tipuri, nume de module) pe care le putem scrie mai departe. Dacă este disponibilă, Visual Studio Code ne va afișa și o documentație scurtă pentru funcțiile predefinite. Spre exemplu, mai jos am început să scriu **flo** și a apărut următoarea listă:



Pentru a selecta un element din listă, folosim săgețile sus/jos de pe tastatură. Pe măsură ce schimbăm selecția folosind săgețile, se va schimba și documentația în concordanță cu elementul selectat. În exemplul de mai sus, am selectat funcția **float_of_int** și a apărut textul de documentație „Convert an integer to floating-point”, precum și tipul funcției.

Dacă am selectat în listă numele pe care dorim să îl scriem, putem să apăsăm tasta „Tab”, sau tasta „Enter”, și numele va fi inserat în fișier fără să îl mai tastăm noi.

Lista poate conține atât nume de funcții și variabile predefinite, cât și nume de funcții și variabile definite de noi:

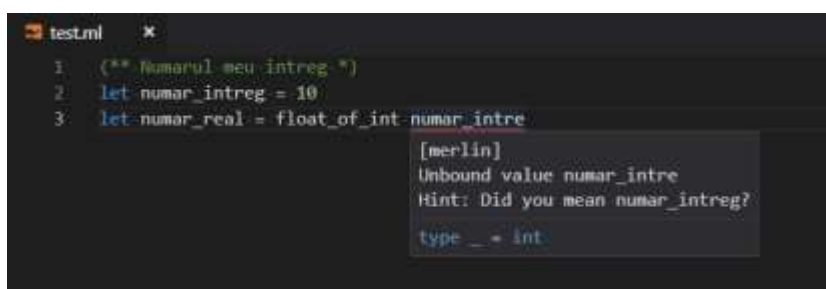


Vedem în imaginea de mai sus că lista conține numele „numar_intreg” pe care l-am definit mai sus. Editorul nu are ce documentație să afișeze, dar poate să ne spună tipul variabilei: în cazul acesta, `int`.

Notă: Observați în ultimul exemplu numele destul de lung al variabilei. Nu trebuie să vă fie frică când dați nume la variabile că dacă e prea lung va fi un inconvenient să îl tastați. Dați **nume sugestive** variabilelor și funcțiilor care le definiți. Editorul vă va ajuta să le scrieți, prin intermediul listei cu nume.

Notă: Dacă ați încercat să scrieți codul de mai sus și nu a apărut nici o listă, înseamnă că nu ați instalat extensia pentru OCaml (vedeți [aici](#) cum), sau nu ați instalat Merlin și ocp-indent (vedeți [aici](#) cum), sau nu ați configurat corect extensia (vedeți [aici](#) cum). Încercați să rezolvați problema și dacă nu reușiți consultați un asistent de laborator. **Nu subestimați importanța unui mediu de dezvoltare configurat corect pentru productivitate.**

Un alt mod în care ne poate ajuta editorul este să ne afișeze erori, în timp ce scriem cod:

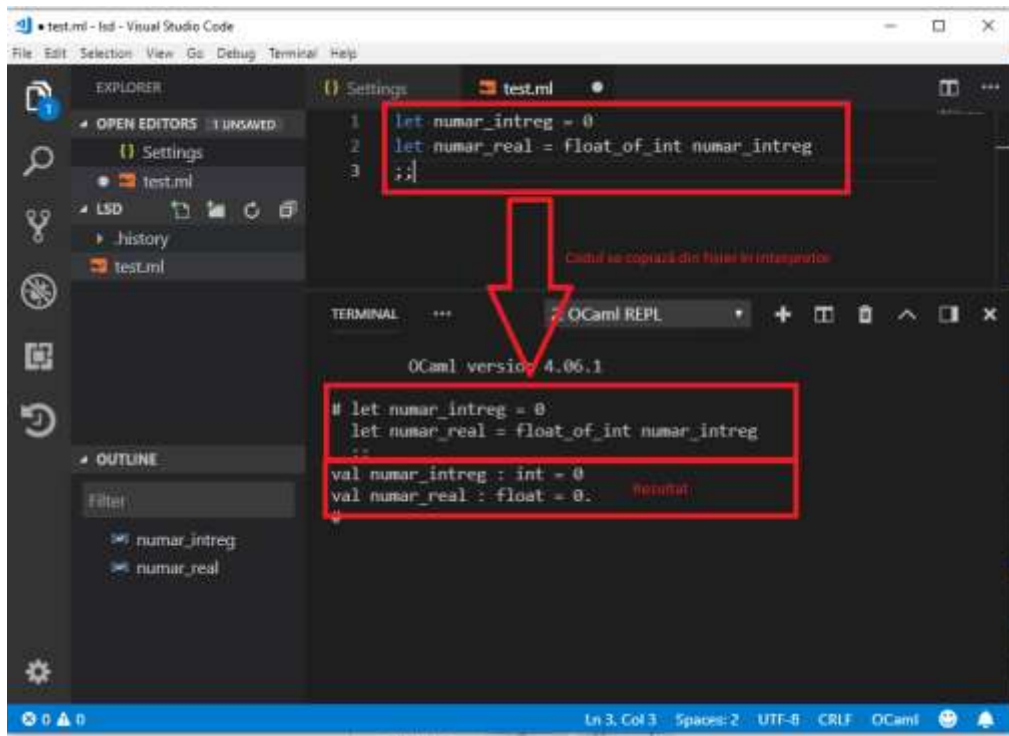


În imaginea de mai sus, vedem că textul `numar_intre` a fost subliniat cu o line roșie, marcând faptul că acest text cauzează o eroare. Dacă mergem cu mouse-ul peste textul subliniat, vom vedea textul erorii: „Unbound value numar_intre”.

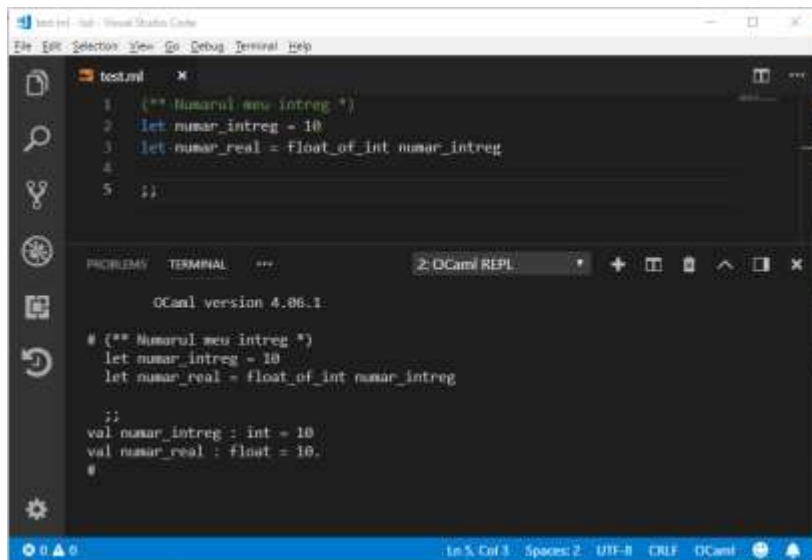
Pentru a executa fișierul prin trimiterea lui către interpretor, trebuie să selectăm comanda „Send File to REPL Session” din paleta de comenzi (vezi [aici](#) despre paleta de comenzi), sau, dacă am configurat corect scurtăturile, din tastatură, apăsând tasta **F6**



După ce execută comanda, codul pe care l-am scris în fișier se va copia în interpretor, va fi evaluat și se vor afișa rezultatele.



Notă: Deoarece vom executa fișierul în interpretor, fișierul va trebuie să conțină la sfârșitul său caracterele `;;` pentru a semnală interpretorului că poate începe execuția. Dacă vreți să citiți mai multe despre când trebuie să punem `;;` răspunsul este „aproape niciodată”, dar puteți consulta [această](#) pagină pentru o discuție mai detaliată.



Notă: Dacă uităm să punem `;;` la sfârșitul fișierului și îl trimitem interpretorului, interpretorul va nu va începe evaluarea ci va aștepta până scriem `;;`. Putem face acest lucru manual în panelul interpretorului.

Tipul caracter și tipul șir de caractere

De multe ori este nevoie de reprezentăm text în cadrul unui program. Fie dorim să citim un text de la utilizator, fie dorim să afișăm un mesaj text în consolă.

Tipul Caracter

Baza pentru reprezentarea textului în limbajele de programare este tipul caracter, denumit în OCaml (dar și în multe alte limbaje) `char`. O variabilă de tip caracter poate ține un singur caracter. Pentru a defini o valoare din acest tip, trebuie să punem caracterul pe care vrem să îl reprezentăm între `'`. Ex:

```
# let un_caracter = 'a';;  
val un_caracter : char = 'a'
```

După ce avem o astfel de valoare, putem să o folosim în expresii. Funcțiile care lucrează cu tipul caracter sunt puse în modulul `Char`. (Un modul este o colecție de funcții grupate sub un nume comun. Pentru a accesa funcțiile din modul, trebuie să prefixăm apelul cu numele modului, urmat de `.` și numele funcției din modul)

Câteva exemple de funcții care lucrează cu caractere:

```
# let un_caracter = 'a'  
  let caracterMare = Char.uppercase_ascii un_caracter  
  let caracterMic = Char.lowercase_ascii 'A';;  
val un_caracter : char = 'a'  
val caracterMare : char = 'A'  
val caracterMic : char = 'a'
```

Caractere speciale

Nu toate caracterele pot fi scrise direct între `'`. Spre exemplu dacă vrem să reprezentăm caracterul linie nouă, trebuie să scriem `'\n'`. Această secvență se numește [secvență de escape](#). O astfel de secvență e introdusă de caracterul `\` și este urmată de un caracter care reprezintă un cod pentru caracterul reprezentat (pentru linie nouă, caracterul este `n`). Aceste caractere speciale au de obicei semnificații speciale la afișare și vor fi afișate cu semnificația lor specială când sunt afișate cu [funcțiile de scriere](#) prezentate în capitolul următor.

Câteva secvențe de escape comune:

Caracter	Secvență de escape
Linie nouă	<code>\n</code>
Tab	<code>\t</code>
Ghilimele	<code>\"</code>
Apostrof	<code>\'</code>
Backspace	<code>\b</code>

Șir de caractere

După cum sugerează și numele acestui tip, tipul șir de caractere este o înșiruire de mai multe caractere. Șirul de caractere pe care dorim să îl reprezentăm, trebuie pus între ghilimele (`"`). Ex:

```
# let un_sir = "Ana are mere!";;  
val un_sir : string = "Ana are mere!"
```

Putem folosi secvențele de escape pentru a introduce caractere speciale în șirul pe care îl definim. De exemplu putem reprezenta textul de mai jos astfel:

Ana are mere!

Si vrea sa le imparta cu noi!

```
# let un_sir = "Ana are mere!\nSi vrea sa le imparta cu noi!";;  
val un_sir : string = "Ana are mere!\nSi vrea sa le imparta cu noi!"
```

Din nou avem un set de funcții definite sub modulul **String** care ne ajută să lucrăm cu șirurile de caractere

```
# let prima_litera_mare = String.capitalize_ascii "nato"  
let toate_literele_mari = String.uppercase_ascii "nato";;  
val prima_litera_mare : string = "Nato"  
val toate_literele_mari : string = "NATO"
```

Concatenarea de șiruri de caractere

De multe ori avem două șiruri de caractere pe care vrem să le „lipi” împreună. Acest proces de lipire, poartă numele de concatenare de șiruri. Pentru a face această operație, OCaml un operator special de concatenare de șiruri reprezentat de caracterul **^**.

```
# let sir1 = "Ana"  
let sir2 = " are mere!"  
let rezultat = sir1 ^ sir2;;  
val sir1 : string = "Ana"  
val sir2 : string = " are mere!"  
val rezultat : string = "Ana are mere!"
```

Funcții de citire / scriere

În biblioteca standard OCaml există și funcții care ne permit citirea-scrierea din/în [consolă](#). În mod convențional, funcțiile pentru citire încep cu **read_**, iar funcțiile pentru scriere încep cu **print_**. Câteva funcții utile sunt:

Nume funcție	Descriere	Exemplu de utilizare
print_int	Scrie un număr întreg în consolă	<code>let () = print_int 1</code>
print_float	Scrie un număr real în consolă	<code>let () = print_float 1.</code>
print_string	Scrie un șir de caractere în consolă	<code>let () = print_string "Suma este:"</code>
print_endline	Scrie un șir de caractere în consolă urmat de o linie nouă	<code>let () = print_endline "Done!"</code>
print_newline	Scrie o linie nouă în consolă	<code>let () = print_newline ()</code>
read_line	Citește o linie din consolă	<code>let linie = read_line ()</code>
read_int	Citește un număr întreg din consolă	<code>let a = read_int ()</code>
read_float	Citește un număr real din consolă	<code>let a = read_float ()</code>

Tipul unit

Observăm în utilizarea funcțiilor de mai sus secvența **()**. Aceasta este o valoare (la fel cum 1 este o valoare) din tipul **unit** care semnifică nimic.

Putem să folosim valoarea acestui tip în expresii. De exemplu putem să o atribuim la o variabilă:

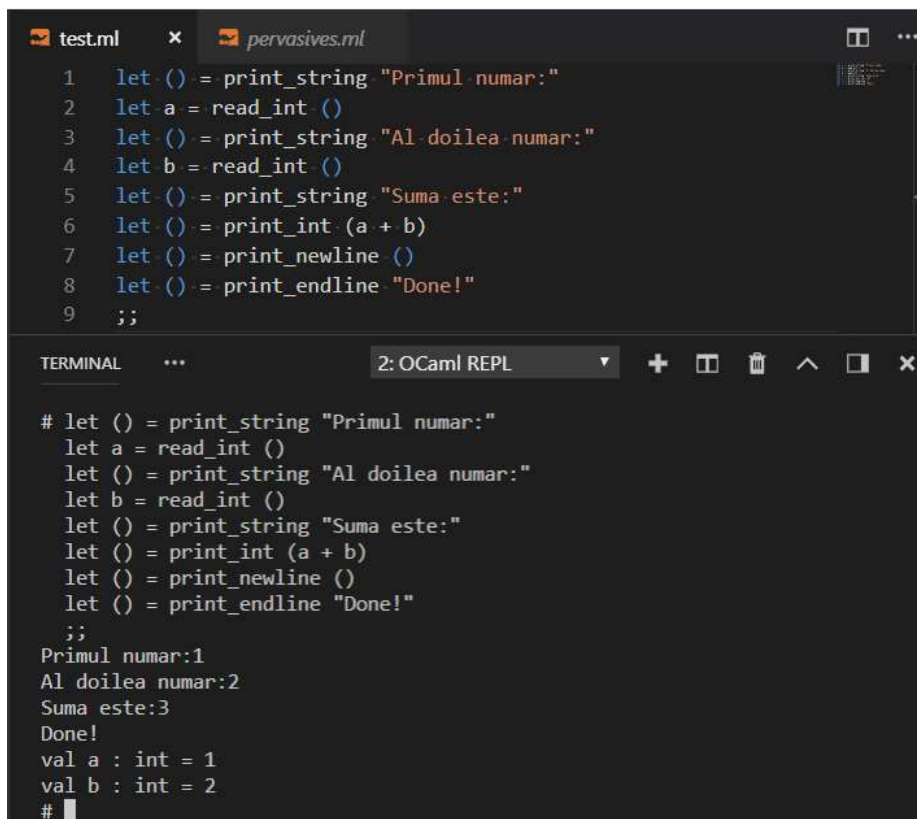
```
# let nimic = ();;  
val nimic : unit = ()
```

În OCaml, pentru a apela o funcție, trebuie să îi pasăm cel puțin un argument. Dacă am scrie `let a = read_int` nu s-ar întâmpla nici o citire, funcția nu ar fi apelată (codul nu ar da nicio eroare, deoarece codul este valid și are o semnificație pe care o vom vedea într-un [capitol](#) următor). Pentru a apela funcția `read_int` trebuie să îi pasăm un argument, iar funcția `read_int` acceptă un sigur parametru de tip `unit`, adică o valoare care nu are nicio semnificație, dar care trebuie să fie prezent, din motive sintactice.

Observăm de asemenea `let () = [apel de funcție]` în fața apelului unor funcții. Aceste funcții au doar efecte asupra consolei (afișează ceva), dar nu ne returnează nici o valoare utilă. Valoarea returnată de aceste funcții este tot `unit`, dar dat fiind faptul că nu avem ce face cu această valoare, nu o vom atribui unei variabile. Vom declara în loc faptul că așteptăm ca rezultatul funcției să fie valoarea nimic prin intermediul sintaxei `let () = [apel de funcție]`.

Notă: motivul pentru care este bine să scriem `let () =` în fața funcțiilor care ne așteptăm să returneze nimic, precum și motivul pentru care această sintaxă este validă vor fi explicate în capitole ulterioare ([aplicarea parțială a funcțiilor](#), respectiv potrivirea de tipare).

Exemplu de utilizare a funcțiilor de citire-scriere:



```
test.ml x pervasives.ml
1 let () = print_string "Primul numar:"
2 let a = read_int ()
3 let () = print_string "Al doilea numar:"
4 let b = read_int ()
5 let () = print_string "Suma este:"
6 let () = print_int (a + b)
7 let () = print_newline ()
8 let () = print_endline "Done!"
9 ;;

TERMINAL 2: OCaml REPL
# let () = print_string "Primul numar:"
  let a = read_int ()
  let () = print_string "Al doilea numar:"
  let b = read_int ()
  let () = print_string "Suma este:"
  let () = print_int (a + b)
  let () = print_newline ()
  let () = print_endline "Done!"
  ;;
Primul numar:1
Al doilea numar:2
Suma este:3
Done!
val a : int = 1
val b : int = 2
#
```

Printf.printf

De multe ori este necesar să afișăm un șir de caractere în care să inserăm anumite valori calculate. Funcția `Printf.printf` ne poate ajuta să facem acest lucru.

În exemplul de mai sus, avem următoarea secvență:

```
let () = print_string "Suma este:"
```

```
let () = print_int (a + b)
```

Am putea să înlocuim cele două linii cu una singură:

```
let () = Printf.printf "Suma este %d" (a + b)
```

În șirul de mai sus caracterele `%d` vor fi înlocuite cu rezultatul expresiei `(a + b)`.

Mai jos avem un tabel cu câteva secvențe de caractere speciale.

Secvență	Semnificație	Exemplu de utilizare
<code>%d</code>	Va fi înlocuit de un număr întreg	<code>let _ = Printf.printf "Intreg: %d" 1</code>
<code>%f</code>	Va fi înlocuit de un număr real	<code>let _ = Printf.printf "Real: %f" 1.</code>
<code>%s</code>	Va fi înlocuit cu un șir de caractere	<code>let _ = Printf.printf "Sir: %s" "sir"</code>

Primul parametru al funcției `printf` se numește **șirul de formatare** și determină ce va fi afișat, precum și câte argumente mai trebuie să pasăm la funcție, precum și care sunt tipurile argumentelor.

Exemple

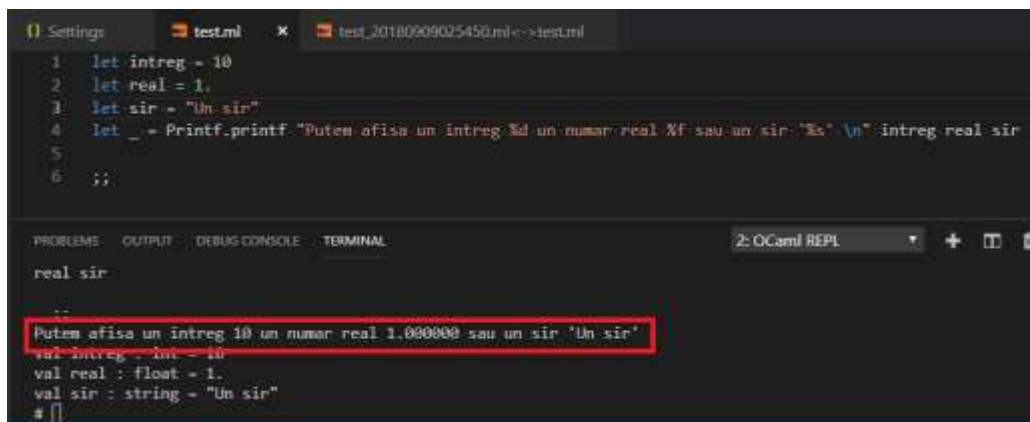
Următorul cod generează o eroare, deoarece șirul de formatare declară că are nevoie de un întreg (`%d`), dar argumentul pasat este real (`1.`)

```
let () = Printf.printf "Intreg: %d" 1.  
This expression has type float but an expression was expected of type
```

Putem să avem mai multe înlocuiri în șirul de formatare, dar trebuie să pasăm valori pentru toate pentru a face afișarea:

```
let () = Printf.printf "Intreg: %d Real: %f" 1 1.
```

Un exemplu mai complex de utilizare a lui `Printf.printf`:



```
(*) Settings test.ml x test_20180909025450.ml <-> test.ml  
1 let intreg = 10  
2 let real = 1.  
3 let sir = "Un sir"  
4 let _ = Printf.printf "Putem afisa un intreg %d un numar real %f sau un sir '%s' \n" intreg real sir  
5  
6 ;;  
  
real sir  
..  
Putem afisa un intreg 10 un numar real 1.000000 sau un sir 'Un sir'  
val intreg : int = 10  
val real : float = 1.  
val sir : string = "Un sir"  
# []
```

Tipul Boolean

Tipul boolean reprezintă cele două valori de logice, valoarea adevărat (reprezentată prin constanta `true`) și valoarea fals (reprezentată prin constanta `false`). (Numele tipului vine de la matematicianul [George Boole](#), care a definit primul un sistem de algebră logică în secolul al XIX-lea).

Să definim două variabile din acest tip:

```
# let adevarat = true
  let fals = false;;
val adevarat : bool = true
val fals : bool = false
```

Operatorii relaționali

De multe ori dorim să comparăm două valori, fie dorim să vedem dacă sunt egale sau diferite, sau vrem să vedem dacă o valoare este mai mică sau mai mare decât o altă valoare. Rezultatul unei astfel de comparații este o valoare de tip booleană. La urma urmei, răspunsul la întrebarea „Este x este egal cu y?” poate să fie „da” (adică valoarea booleană `true`) sau „nu” (adică valoarea booleană `false`).

Să vedem cum putem afla dacă alte două variabile au valori egale:

```
# let x = 10
  let y = 15
  let x_este_egal_cu_y = x > y;;
val x : int = 10
val y : int = 15
val x_este_egal_cu_y : bool = false
```

Variabila `x_este_egal_cu_y` va conține valoarea de adevăr a expresiei `x > y`. În acest caz variabila `x_este_egal_cu_y` va avea valoarea `false` deoarece `x` (cu valoarea `10`) nu este mai mare decât `y` (care are valoarea `15`).

Operatorii relaționali:

Denumire	Simbol	Exemplu
Egal	<code>=</code>	<pre># let x = 1 let y = 10 let x_este_egal_cu_y = x == y;; val x : int = 1 val y : int = 10 val x_este_egal_cu_y : bool = false</pre>
Diferit de	<code><></code>	<pre># let x = 1. let x_diferit_de_0 = x != 0.;; val x : float = 1. val x_diferit_de_0 : bool = true</pre>
Mai mare	<code>></code>	<pre># let x = 1. let x_mai_mare_decat_0 = x > 0.;; val x : float = 1. val x_mai_mare_decat_0 : bool = true</pre>
Mai mare sau egal	<code>>=</code>	<pre># let x = 1. let y = 10. let x_mai_mare_sau_egal_cu_y = x >= y;; val x : float = 1. val y : float = 10. val x_mai_mare_sau_egal_cu_y : bool = false</pre>
Mai mic	<code><</code>	<pre># let x = 1. let x_mai_mic_decat_0 = x < 0.;; val x : float = 1. val x_mai_mic_decat_0 : bool = false</pre>
Mai mic sau egal	<code><=</code>	<pre># let x = 1. let y = 10. let x_mai_mic_sau_egal_cu_y = x <= y;; val x : float = 1. val y : float = 10. val x_mai_mic_sau_egal_cu_y : bool = true</pre>

Operanzii unui operator relațional trebuie să fie de același tip dar pot să fie de orice tip: întreg, real, caracter, șir de caractere, boolean (**Exercițiu:** Testați operatorii pentru celelalte tipuri).

Notă: Dacă ați lucrat în alte limbaje, s-ar putea să fiți obișnuiți cu operatorii `==` și `!=` pentru egalitate, respectiv „diferit de”. OCaml are acești operatori, dar cu o semnificație puțin diferită. De cele mai multe ori, acești operatori (`==` și `!=`) vor produce aceleași rezultate ca operatorii recomandați mai sus (`=` și `<>`), dar nu întotdeauna. Motivul diferenței nu face subiectul acestui curs, dar, dacă doriți o explicație, o puteți găsi pe scurt [aici](#). Recomandarea este să folosiți `=` și `<>`, **nu** `==` și `!=`.

Operatorii logici

La fel cum avem operatori dedicație pentru alte tipuri, avem și operatori pentru tipul logic. Operatorii comuni sunt prezentați în tabelul de mai jos:

Operator	Denumire	Descriere	Exemple
not	Negare	Returnează adevărat dacă operandul este false, și false dacă operandul este adevărat	<pre># let nu_adevarat = not true (* va avea valoarea false *) let nu_fals = not false (* va avea valoarea true *);; val nu_adevarat : bool = false val nu_fals : bool = true</pre>
&&	Și logic	Returnează adevărat dacă ambii operanzi au valoare adevărat și fals în celelalte cazuri	<pre># let va_fi_adevarat = true && true let va_fi_false1 = false && true let va_fi_false2 = true && false let va_fi_false3 = false && false ;; val va_fi_adevarat : bool = true val va_fi_false1 : bool = false val va_fi_false2 : bool = false val va_fi_false3 : bool = false</pre>
 	Sau logic	Returnează adevărat dacă unul din operanzi are valoarea adevărat și fals în celelalte cazuri	<pre># let va_fi_adevarat1 = true true let va_fi_adevarat2 = false true let va_fi_adevarat3 = true false let va_fi_false = false false ;; val va_fi_adevarat1 : bool = true val va_fi_adevarat2 : bool = true val va_fi_adevarat3 : bool = true val va_fi_false : bool = false</pre>

Exemplu de utilizare a operatorilor logici și relaționali

Putem combina operatorii relaționali cu operatorii logice pentru a obține răspunsul la înrebări mai complicate. De exemplu să presupunem că avem două variabile `inceput` și `sfarsit` care reprezintă un interval, și o variabilă `x` care dorim să vedem dacă e în interval. Care este condiția ca `x` să fie în interval? Dacă `x` este **mai mare** decât `inceput` **și** `x` este **mai mic** decât `sfarsit`, `x` este în interval.

Folosind operatorii logici și relaționali această condiție poate fi scrisă:

```
# let inceput = 10
  let sfarsit = 20
  let x = 15
  let y = 0
  let z = 25
  let x_este_in_interval = inceput < x && x < sfarsit
  let y_este_in_interval = inceput < y && y < sfarsit
  let z_este_in_interval = inceput < z && z < sfarsit
  ;;
val inceput : int = 10
val sfarsit : int = 20
val x : int = 15
val y : int = 0
val z : int = 25
val x_este_in_interval : bool = true
val y_este_in_interval : bool = false
val z_este_in_interval : bool = false
```

Expresia condițională

Există situații în cod când dorim să luăm o decizie pe baza unei valori unei expresii booleene. De exemplu Dacă o variabilă (să o denumim `x`) este mai mare decât 0, dorim ca o altă variabilă (să o denumim `y`) să aibă valoare 1, altfel dorim să fie 1.

Pentru a obține acest comportament, putem folosi expresia `if`.

```
#let x = 10
  let y = if x > 0 then 1 else -1;;
val x : int = 10
val y : int = 1
```

Expresia `if` are următoarea structură :

```
if [expresie de tip boolean]
  then [expresie de tip 'a]
  else [expresie de tip 'a]
```

Expresia de tip boolean poate să fie orice expresie care are un rezultat boolean. În exemplul de mai sus am folosit un operator relațional, am fi putut să folosim și operatori logici, sau o funcție care returnează o valoare boolean.

Cele două expresii care dau rezultatul expresiei condiționale (cea de după `then` și cea de după `else`) pot avea orice tip, dar trebuie să aibă același tip. De exemplu următorul cod produce o eroare, deoarece o ramură produce un rezultat `float` pe când cealaltă produce un rezultat `int`.


```
# let x = 10
  let y = if x > 10 then 1. else 0;;
Characters 42-43:
  let y = if x > 10 then 1. else 0;;
                                ^
Error: This expression has type int but an expression was expected of type
float
```

Expresia condițională, este o expresie, și ca atare are un rezultat și poate fi folosită oriunde am putea folosi orice altă expresie. Mai jos vedem câteva exemple de utilizare în expresii mai complicate:

```
# let x = 10

(* Putem folosi if ca unul din operandi pentru un operator *)
let y = 2 * (if x > 0 then 1 else -1) (* y = 2 * 1 = 2 *)

(* Sau ca argument la o alta functie *)
let r = float_of_int (if x > 0 then 1 else -1) (* r = float_of_int 1 = 1. *)
;;
val x : int = 10
val y : int = 2
val r : float = 1.
```

Exercițiu rezolvat

Testați dacă valoarea unei variabile este în intervalul [0, 10]. Dacă da, afișați testul „Numărul **număr** este o notă validă”, dacă nu, afișați textul „Numărul **număr** nu este o notă validă!”

Pentru început trebuie să vedem care ar fi condiția care pentru ca un număr să fie în interval. Dacă numărul este în interval, va fi mai mare sau egal cu începutul intervalului ($0 \leq n$) și mai mic decât sfârșitul intervalului ($n \leq 10$). Punând cele două condiții împreună cu operatorul **&&** (cele două condiții trebuie să fie simultan adevărate) obținem ($0 \leq n \ \&\& \ n \leq 10$)

Folosind această condiție, putem să folosim expresia condițională pentru a afișa un mesaj folosind funcția **Printf.printf**:

```
# let n = 10
  let () = if 0 <= n && n <= 10
    then Printf.printf "Numărul %d este o notă validă\n" n
    else Printf.printf "Numărul %d NU este o notă validă\n" n
  ;;
Numarul 10 este o nota valida
val n : int = 10
```

Observăm că rezultatul expresiei **if** în acest caz este de tip **unit** (**()**) deoarece **Printf.printf** returnează o valoare de acest tip.

Exercițiu individual: Cum putem afișa mesaj doar dacă nota nu este validă. Soluția dorită, nu ar trebui să apeleze nici o funcție de scriere dacă nota este validă.

Expresia secvență de expresii

Uneori dorim să facem succesiv mai multe lucruri. De exemplu, am dori să afișăm o valoare, dar am dori ca expresia noastră să aibă un rezultat care nu este de tip `unit` (). Putem face acest lucru folosind o secvență de expresii:

```
# let x = 10
  let o = print_string "Initializăm pe o cu valoarea lui x\n"; x + 1;;
Initializam pe o cu valoarea lui x
val x : int = 10
val o : int = 11
```

În codul de mai sus, în inițializarea lui `o`, facem mai întâi o afișare cu expresia `print_string "..."`, a cărei rezultat este ignorat, după care avem expresia `x + 1` care va fi valoarea secvenței și care va fi atribuită lui `o`.

În general expresia o secvență de expresii are forma

```
expr1; expr2; expr3; ...exprN-1;exprN
```

Valorile expresiilor `expr1`, `expr2`, `expr3`, ...`exprN-1` vor fi ignorate (dar dacă au efecte secundare, spre exemplu afișarea pe ecran, aceste efecte se vor produce). Tipul secvenței de expresii, și valoarea ei este dată de ultima expresie adică `exprN`.

Deoarece valorile expresiilor `expr1`, `expr2`, `expr3`, ...`exprN-1` sunt ignorate, tipul acestor expresii este de obicei `unit` (adică nimic). În acest curs, singurele funcții care returnează nimic vor fi cele de afișare pe ecran). Dacă tipul acestor expresii nu este `unit`, compilatorul va semnala un avertisment. Codul se va executa, dar nu ignorați avertismentul. Cel mai probabil motiv pentru avertisment este o eroare în codul pe care l-ați scris. În codul pe care îl vom scrie, nu există nici un motiv bun pentru ca aceste expresii să aibă alt tip decât `unit`.

Ex de avertisment:

```
# let o = 1 + 2; 2;;
Characters 8-13:      Avertisment
  let o = 1 + 2; 2;;
      ^^^^^
Warning 10: this expression should have type unit.
val o : int = 2      Rezultat. Codul a fost executat.
```

În exemplul de mai sus, inițializăm variabila `o` cu secvența `1+2;2`. Rezultatul primei expresii `1+2` este ignorat, nu există nici un motiv pentru care să fi făcut calculul respectiv, nu are nici un rol, deci probabil că nu am vrut să facem o astfel de secvență de expresii, și probabil am vrut să scrie `+` spre exemplu în loc de `;`.

O secvență de expresii, este o expresie, și deci poate să apară în orice loc apare o expresie.

Să vedem cum se comportă o secvență de expresii ca operand:

```
# let o = (print_string "operand 1\n"; 1) + (print_string "operand 2\n"; 2);;
operand 2
operand 1
val o : int = 3
```

În inițializarea lui `o` de mai sus, adunăm `1` cu `2`, dar ambele valori sunt rezultatul unei secvențe de expresii care conține o afișare. Când expresiile sunt evaluate, se face afișarea mesajelor după care rezultatul fiecărei secvențe va fi folosită în adunare rezultând valoarea lui `o` care va fi `3`

Este interesant de observat că se afișează mai întâi șirul de caractere din operandul al doilea (`print_string "operand 2\n"; 2`), și apoi cel din primul operand (`print_string "operand 1\n"; 1`). De aici putem să ne dăm seama că ordinea în care se evaluează operația de adunare este de la dreapta la stânga.

Exercițiu: Aplicați aceeași strategie de a folosi o secvență de expresii ca operanzi împreună cu operatorii `&&` și `||`. Pentru `&&` alegeți valorile în așa fel încât primul operand să fie fals și al doilea adevărat (expresia pe care o scrieți să fie echivalentă cu `false && true`). Pentru `||` alegeți valorile în așa fel încât prima să fie adevărată și a doua fals (expresia să fie echivalentă cu `true || false`). Se fac scrierile asociate ambilor operanzi ? De ce ?

Scrierea, secvența de expresii și depanarea

Scrierea combinată cu secvența de expresii poate să fie un instrument foarte simplu dar util în depanarea programelor.

Prin afișare putem inspecta valorile variabilelor (și mai târziu a parametrilor la funcții) și putem să ne dăm seama care ramură a unei expresii `if` a fost evaluată (și mai târziu și a altor expresii care au mai multe ramuri ca `match` și `try` pe care le vom vedea în acest laborator).

Faptul că doar ultima expresie din secvență este semnificativă pentru rezultatul expresiei iar restul expresiilor din secvență sunt evaluate, dar rezultatul lor este ignorat, ne dă posibilitatea să introducem o scriere în absolut orice locație din cod.

Să luăm o problemă simplă: Se dă o variabilă `x` care are valoarea citită de la tastatură. Definiți o variabilă `n`, care are valoarea lui `x` dar limitată la intervalul `[0, 10]`. Dacă valoarea lui `x` este mai mare decât `10`, valoarea lui `n` va fi `10`, iar dacă valoarea lui `x` e mai mică decât `0`, valoarea lui `n` va fi `0`.

Să vedem o tentativă de rezolvare:

```
let () = print_string "Introduceți nota:"
let x = read_int ()
let n = if x < 0
  then 0
  else if x < 10
    then 10
    else x
;;
```

Să testăm acest program cu valorile `-1`, `6` și `14`. Observăm că ultima valoarea nu este conformă cu cerințele problemei.

```
Introduceti nota:-1
val x : int = -1
val n : int = 0
```

```
Introduceti nota:6
val x : int = 6
val n : int = 10
```

```
Introduceti nota:14
val x : int = 14
val n : int = 14
```

Putem să ne uităm la cod și să vedem eroare, dar e ora 2 dimineața în ziua dinaintea laboratorului și nu observăm că nu am făcut bine comparația cu 10. Ar fi de ajutor să vedem „pe unde merge programul”, adică care ramură a expresiilor if au fost evaluate. Să adăugăm câteva secvențe de expresii cu afișări:

```
let () = print_string "Introduceti nota:"
let x = read_int ()
let n = if x < 0
then (print_string "x a fost mai mic ca 0, folosim 0\n"; 0)
else (
  print_string "x a mai mare ca 0, mai vedem\n";
  if x < 10
  then (print_string "x a fost mare ca 10, folosim 10\n"; 10)
  else (print_string "x a fost in inetrvalul 0, 10 e ok\n"; x)
)
;;
```

Notă: Sunt necesare și câteva paranteze în plus din cauza precedenței lui `;` față de `if`.

Rulând codul de mai sus obținem:

```
Introduceti nota:14
x a mai mare ca 0, mai vedem
x a fost in inetrvalul 0, 10 e ok
val x : int = 14
val n : int = 14
```

Observăm că, spre surpriza noastră, ramura care este evaluată este cea care zice că valoarea este în intervalul [0,10], deci examinăm mai atent condiția care duce la această ramură.

Definirea Funcțiilor

Până acum am folosit doar funcții predefinite. Putem să definim și funcții noi, pe care să le folosim. Există mai multe motive pentru care am dori să scriem funcții. Un motiv major este faptul că o funcție trebuie definită o singură dată, dar poate fi utilizată de mai multe ori.

Să luăm o funcție matematică simplă:

$$f: \mathbb{Z} \rightarrow \mathbb{Z}, f(x) = x + 3$$

Transcrisă în OCaml această funcție ar fi:

```
let f x = x + 3
```

La modul mai general, definirea unei funcții are următoarele componente:

```
let [nume funcție] [listă de parametri] = [expresie de definiție]
```

Numele funcției trebuie să înceapă cu un o literă mică (**numele cu literă mare au altă semnificație**) sau caracterul `_`, poate să conțină litere mici, litere mari, numere și caracterele `-` și `'`. Numele funcției poate să fie orice (respectând restricțiile pe care le-am specificat), dar ar fi util să alegem un nume sugestiv.

Lista de parametri trebuie să conțină cel puțin un parametru (dacă nu avem nici un parametru, înseamnă că nu mai definim o funcție, ci o variabilă). Lista de parametri poate să conțină mai mulți parametri, separați prin spațiu. Numele parametrului respectă aceleași reguli cu privire la nume ca numele funcției. Parametrii definiți pot să fie utilizați mai departe în expresia care definește funcția.

Expresia de definiție a funcției poate să fie orice expresie validă OCaml. În expresie putem să folosim parametrii definiți în lista de parametri.

După ce definim funcția, putem să o folosim ca orice altă funcție pe care am utilizat-o până acum.

Terminologie: Definiția funcției are parametri (de exemplu în `let f x = x + 3`, `x` este un parametru). Când apelăm funcția, spunem că „pasăm argumente funcției” (de exemplu, în expresia `f 0`, `0` este argument al funcției `f`)

Când apelăm funcția, argumentele care sunt pasate funcției vor fi valorile parametrilor pentru apelul respectiv al funcției, în expresia de definiție.

```
# let f x = x + 3 (* Definiție *)
  let r = f 0 (* Apel f 0 = 0 + 3 = 3 *)
  ;;
val f : int -> int = <fun>
val r : int = 3 Rezultat
```

Legătura dintre argumentele pasate la apel și parametrii în care sunt puse valorile este pozițională. Adică argumentul de pe prima poziție va fi valoarea primului parametru, valoarea argumentului de pe a doua poziție va fi valoarea celui de al doilea parametru al funcției.

Să luăm un exemplu de funcție cu 3 parametri:

$g : \mathbb{Z} \rightarrow \mathbb{Z}, g(x, y, z) = x * y + z$

În acest caz, lista de parametri conține 3 parametri: `x`, `y` și `z`. Apelul funcției `g` pasează 3 argumente (`2 5 3`). Pentru acest apel, când se calculează valoarea lui `g`, `x` va avea valoarea `2`, `y` valoarea `5` iar `z` valoarea `3`.

```
# let g x y z = x * y + z
  let r = g 2 5 3
  ;;
val g : int -> int -> int -> int = <fun>
val r : int = 13
```

Tipul unei funcții

Când interpretorul evaluează o declarație de variabilă, ne va afișa numele variabilei, tipul ei, precum și valoarea acesteia, sub următoarea formă :

```
val [nume variabilă] : [tipul variabilei] = [valoare variabilă]
```

Spre exemplu:

```
# let x = 10;;  
val x : int = 10
```

Când definim o funcție, observăm că interpretorul ne va răspunde în mod similar:

```
# let f x = (float_of_int x) +. 1.;;  
val f : int -> float = <fun>
```

Unde înainte aveam valoarea variabilei, acum avem textul `<fun>`, cu semnificația că valoarea lui `f` este o funcție (interpretorul nu ne va spune exact care este expresia care definește funcția, doar faptul că valoarea este o funcție).

Mai interesant este ce avem acolo unde înainte aveam tipul variabilei și acum avem textul `int -> float`. Acest text reprezintă tipul funcției și el ne spune despre o funcție ce tip de argumente acceptă această funcție, precum și care este tipul valorii returnate de funcției. Funcția de mai sus acceptă un argument de tip întreg (determinat de `int` din tipul `int -> float`) și returnează un rezultat real (determinat de `float` din tipul `int -> float`).

La modul general, pentru o funcție cu un parametru avem `[tip parametru] -> [tip rezultat]`

Pentru funcții cu mai mulți parametri, lista cu tipurile de parametri acceptați crește, pentru fiecare adăugându-se o expresie de forma `[tip parametru] ->`.

De exemplu:

```
# let f x y = (float_of_int x) +. y;;  
val f : int -> float -> float = <fun>
```

Din tipul funcției de mai sus, ne dăm seama că primul parametru trebuie să fie un întreg (`int -> float -> float`), al doilea parametru este un număr real (`int -> float -> float`), iar valoarea returnată este de tip real (`int -> float -> float`).

Pentru a afla tipul oricărei funcții (predefinite sau definite de noi), putem să scriem numele funcției în interpretor (pentru funcțiile definite de noi, codul care face definirea trebuie să fi fost rulat), sau putem să mergem cu mouse-ul în editor peste numele oricărei funcții.

```
let f x y = (float_of_int x) +. y;;  
type _ = int -> float -> float  
let r = f 0 1.
```

```
# let f x y = (float_of_int x) +. y  
let r = f 0 1.  
;;  
val f : int -> float -> float = <fun>  
val r : float = 1.  
# f;;  
- : int -> float -> float = <fun>
```

Să explorăm câteva funcții predefinite:

```
# float_of_int;;
- : int -> float = <fun>
```

`float_of_int` acceptă un parametru de tip întreg și returnează o valoare reală

```
# int_of_float;;
- : float -> int = <fun>
```

`int_of_float` acceptă un parametru de tip real și returnează o valoare întreagă

Tipuri generice de funcții

Uneori o funcție nu acceptă un singur tip de parametru, ci orice tip de parametru (uneori cu anumite restricții, dar nu vom intra în acest subiect în acest laborator). Faptul că argumentul poate să fie de orice tip, nu înseamnă că nu există nici o relație între tipul argumentelor și valoarea returnată, sau că nu există nici o relație între tipurile argumentelor (dacă există mai multe argumente). Să luăm ca exemplu funcția `min` (care returnează minimum a două valori) și să o apelăm cu diverse valori:

```
# min 1 2;;
- : int = 1
# min 1. 2.;;
- : float = 1.
# min 1 2.;;
Characters 6-8:
  min 1 2.;;
    ^^
Error: This expression has type float but an expression was expected of type
      int
```

Mai sus avem 3 apeluri ale funcției `min`:

- În primul apel pasăm două numere întregi, iar rezultatul este tot un număr întreg.
- În al doilea apel pasăm funcției două numere reale, iar rezultatul este tot un număr real
- În al treilea caz pasăm un număr întreg și un număr real, iar interpretorul ne răspunde cu o eroare

Ce putem deduce de aici despre comportamentul funcției `min`:

- Acceptă două argumente, neapărat de același tip
- Tipul celor două argumente poate să fie de orice tip (am testat pentru `int` și `float` dar putem încerca și pentru alte tipuri)
- Tipul rezultatului funcției `min` este același cu tipul celor două argumente

Cum arată tipul unei astfel de funcții? Putem să întrebăm interpretorul:

```
# min;;
- : 'a -> 'a -> 'a = <fun>
```

Observăm că tipul parametrilor și tipul returnat este tipul `'a`. Tipul `'a` nu este un tip concret. Este un tip generic, care va fi înlocuit în momentul apelului cu un tip concret. **Oricând vedem un tip care începe cu `'`, acesta este un tip generic.**

```
# min 1 2;;
- : int = 1
# min 1. 2.;;
- : float = 1.
# min 1 2.;;
Characters 6-8:
  min 1 2.;;
    ^
Error: This expression has type float but an expression was expected of type
      int
```

În exemplul de mai sus, pentru primul apel 'a va fi `int`, pentru al doilea 'a va fi `float`, în ambele cazuri acest lucru este determinat de tipul argumentelor.

Pentru al treilea apel 'a ar putea să fie `int` (după primul argument) dar ar putea să fie și `float` (după al doilea argument). Acest conflict între posibilele înlocuiri ale lui 'a duce la eroarea pe care o vedem. Modul în care OCaml determină 'a duce de fapt la eroarea că al doilea parametru este așteptat să fie `int`. Acest lucru se întâmplă deoarece interpretorul întâlnește mai întâi primul argument cu o valoare întreagă `1` și decide în consecință că 'a nu poate fi decât `int` și face înlocuirea lui 'a cu `int`. Mai târziu când întâlnește al doilea argument 'a este deja considerat `int` și eroarea care o primim este că valoarea `2.` nu este un întreg, așa cum ar fi de așteptat.

Notă: Tipul unei funcții s-ar putea să pară un subiect destul de esoteric și rar utilizat în practică. Tipul unei funcții este însă foarte important atât în înțelegerea mesajelor de eroare (când primim o eroare este important să ne uităm să vedem care este tipul funcției și care sunt tipurile argumentelor pe care le-am pasat), precum și în scrierea codului, când întâlnim funcții pentru prima oară (sau pentru a doua oară, dar nu mai știm exact ce argumente acceptă). Numele funcției, împreună cu tipul acesteia, ne dau indicii despre ce face funcția și ce argumente trebuie să îi pasăm.

Cum se determină tipul unei funcții în OCaml

Până acum am văzut că o funcție are un tip, dar de unde provine acest tip ? Nu l-am specificat niciunde. OCaml folosește ceea ce se numește **inferență de tipuri** pentru a determina tipul funcțiilor (de fapt, acest mecanism este și modul în care OCaml a determinat tipul variabilelor). Acest mecanism presupune:

- **Pentru parametri:** Tipul parametrilor se determină pe baza modului în care parametrii sunt utilizați în expresia de definiție.
- **Pentru rezultat:** Tipul se determină pe baza tipului expresiei care definește funcția.

Să luăm drept exemplu o funcție simplă:

```
# let f x = x + 1;;
val f : int -> int = <fun>
```

În exemplul de mai sus, `x` este folosit împreună cu un operator pentru numere întregi, deci tipul lui `x` nu poate fi decât `int`. Pentru tipul valorii returnate, din nou operatorul `+` este decisiv, rezultatul acestui operator este întotdeauna întreg, deci rezultatul lui `f` va fi întotdeauna întreg (`int`).

Nu doar operatorii pot da indicii limbajului cu privire la tipul unui parametru. Utilizarea parametrului ca argument la o altă funcție poate determina și ea tipul unui parametru:


```
# let f x = int_of_float x;;
val f : float -> int = <fun>
```

În exemplul de mai sus, folosim funcția `int_of_float` care are tipul `float -> int` deci acceptă un argument real (`float`) și returnează o valoare întreagă (`int`). Deoarece `int_of_float` acceptă un argument de tip `float`, tipul lui `x` trebuie să fie și el `float`. Deoarece `f` returnează rezultatul lui `int_of_float` tipul rezultatului lui `f` va fi și el `int`.

Dacă apare o inconsecvență în modul în care folosim un parametru, interpretorul va semnaliza o eroare:

```
# let f x = x + int_of_float x;;
Characters 27-28:
  let f x = x + int_of_float x;;
                  ^
Error: This expression has type int but an expression was expected of type
float
```

În exemplul de mai sus, prima utilizare a lui `x` ar sugera că acest parametru este de tip întreg (deoarece e folosit împreună cu operatorul pentru întregi `+`). OCaml va reține acest lucru. Când va întâlni a doua utilizare a lui `x` ca argument pentru `int_of_float`, care așteaptă un parametru real (`float`), va semnaliza o eroare, deoarece `x` a fost deja determinat ca fiind de tip `int`.

Ce se întâmplă în momentul în care nu există suficientă informație pentru a determina tipul unui parametru? Să luăm ca exemplu funcția identitate `f(x) = x`

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

Vedem în tipul lui `f` un parametru generic (`'a`). Acest tip a fost determinat pentru parametru și pentru valoarea returnată, deoarece nu există mai multă informație cu privire la un tip concret pentru parametru și pentru valoarea returnată. Tot ce poate OCaml să spună este că valoarea returnată are același tip cu tipul parametrului. Când vom utiliza funcția, tipul generic `'a` va fi înlocuit cu un tip concret, bazat pe modul în care utilizăm funcția.

Confuzii comune

Parametrii și variabilele

O sursă comună de confuzie este momentul în care avem parametrii și variabile cu același nume. Să luăm un exemplu simplu:

```
let x = 0
let f x = x
let r = f x
```

Care este legătura dintre variabila `x` (marcată cu roșu) și parametrul `x` (marcat cu albastru)?

Răspuns: Niciuna! Nu există nici o legătură întru cele două declarații. E doar o coincidență de nume.

Când definim un parametru nou, dacă acesta are același nume cu o altă variabilă/parametru care ar fi vizibilă din funcție, acesta va ascunde declarația inițială.

Concluzia este că în interiorul expresiei de definiție a funcției, când scriem `x` ne referim la parametru, iar în afara funcției ne referim la variabila declarată anterior :

```
let x = 0
let f x = x
let r = f x
```

Care va fi tipul lui `f` în codul de mai sus ? `int -> int` sau `'a -> 'a`.

Dat fiind faptul că în expresia lui `f`, nu avem mai multe informații despre `x`, lui `x` i se va atribui un tip generic. Faptul că avem o variabilă `x`, de tip întreg, cu același nume **NU ARE NICI O RELEVANȚĂ PENTRU PARAMETRUL** `x`. Putem supune că în acest caz compilatorul vede codul ca și cum am fi denumit variabilele diferit. Următorul cod ar fi echivalent:

```
let x_0 = 0
let f x_1 = x_1
let r = f x_0
```

Execuția unei funcții vs. Declarația unei funcții

De multe ori în probleme se cere „scrieți o funcție care ...”. Un lucru important de reținut este faptul că **declarația unei funcții nu va executa codul funcției** și în consecință nu vom ști dacă funcția funcționează corect decât în momentul în care o utilizăm (cineva o apelează cu niște parametri).

Să presupunem următoarea problemă: Scrieți o funcție care calculează distanța unui punct din plan (specificat prin coordonate (x, y)) până la origine.

Pentru a rezolva această problemă, scriem următoarea funcție:

```
let distanta x y = sqrt ((x *. x) +. (y *. y))
```

Este aceasta o rezolvare corectă ? Fără nici un apel de funcție, nu avem de unde să știm. Desigur, putem să ne uităm cu mare atenție la cod și să vedem că în a doua paranteză am folosit `x` în loc de `y`, dar metoda „ne uităm la cod până suntem convinși că merge” are aplicabilitate limitată. Cu cât crește dimensiunea funcției, cu atât este mai dificil să vedem o eroare.

Următoare funcție funcționează corect ?

```
let rec parseInt ic =
  let rec parseNumbers ic n =
    match input_char ic with
    | '0' .. '9' as c -> parseNumbers ic (n * 10 + (int_of_char c) - (int_of_char '0'))
    | _ -> input ic; n
  in
  let whitespace ic =
    let sign = match input_char ic with
    | '-' -> -1
    | _ -> input ic; 1
    in
    let result = ~ (sign * (parseNumbers ic 0))
    in
    eatWhitespace ic; result
  in
  parseNumbers ic 0
```

Fără să știm exact ce trebuie să facă funcția, deja vedem că metoda „ne uităm la cod până suntem convinși că merge” devine dificilă, iar funcția de mai sus nici măcar nu este una foarte lungă.

Cea mai bună metodă de a determina dacă o funcție funcționează conform așteptărilor este de a apela funcția pentru un set de parametri (reprezentativ) pentru care știm rezultatul așteptat. Dacă rezultatul așteptat este în concordanță cu rezultatul returnat de funcția pe care am scris-o, atunci înseamnă că rezolvarea este corectă (sau, mai bine zis, nu este incorectă, din ce am testat până atunci). Succesul acestei metode

în a produce cod corect depinde foarte mult și de valorile luate pe post de teste ale funcției (cât de reprezentative sunt aceste valori, cât de bine acoperă cazurile posibile).

Să vedem cum ar arăta câteva teste pentru funcția `distance` definită mai devreme:

```
# let distance x y = sqrt ((x *. x) +. (y *. y))
let d1 = distance 0. 0. (* Rezultat așteptat 0. *)
let d2 = distance 1. 0. (* Rezultat așteptat 1. *)
let d3 = distance 0. 1. (* Rezultat așteptat 1. *)
let d4 = distance 1. 1. (* Rezultat așteptat 1.41... *)
let d5 = distance 1. 2. (* Rezultat așteptat 2.23... *)
let d6 = distance 2. 1. (* Rezultat așteptat 2.23... *)

**
val distance : float -> float -> float = <fun>
val d1 : float = 0. ok
val d2 : float = 1. ok
val d3 : float = 0. greșit
val d4 : float = 1.4142135623730951 ok
val d5 : float = 1.7320508075688772 greșit
val d6 : float = 2.4494897427831779 greșit
```

Având aceste teste, putem să ne uităm la implementare, să vedem unde am greșit (testele care au dat rezultatul greșit ne dau un indiciu cu privire la unde ar putea fi eroarea).

Notă: Dacă în rezolvările pe care le predați la laborator (fie ca temă, fie în timpul laboratorului) NU includeți teste pentru funcțiile pe care le scrieți, asistenții își rezervă dreptul să **depunțeze** acest lucru.

Notă: În programare este în general indicat să includem teste (puteți să citiți mai multe despre acest subiect dacă veți căuta „unit test” – „teste unitare”). Există chiar și o metodologie de a scrie cod care ne obligă să scriem teste înainte să scriem implementarea (denumită „test driven development”).

Returnarea unei valori vs. Afișarea unei valori

Când scriem o funcție, este important să returnăm valoarea care reprezintă rezultatul funcției, nu doar să o afișăm.

Să luăm aceeași problemă ca mai devreme: Scrieți o funcție care calculează distanța unui punct din plan (specificat prin coordonate (x, y)) până la origine.

Să luăm și două implementări posibile:

```
# let distanta1 x y = sqrt (x *. x +. y *. y)
let () = Printf.printf "Distanța este %f\n" (distanta1 1. 2.)

let distanta2 x y = Printf.printf "Distanța este %f\n" (sqrt (x *. x +. y *. y))
let () = distanta2 1. 2.

;;
Distanța este 2.236068
Distanța este 2.236068
val distanta1 : float -> float -> float = <fun>
val distanta2 : float -> float -> unit = <fun>
```

Funcția `distanta1` returnează valoarea calculată, pe când `distanta2` face direct afișarea (și nu returnează nimic).

Dacă problema nu specifică explicit ca funcția să facă afișarea, este preferabil să returnăm o valoare calculată în loc să o afișăm în funcție. De ce? Dacă returnăm o valoare putem folosi această valoare în

alte feluri decât doar să o afișăm. Returnarea valorii ne dă o flexibilitate mult mai mare în utilizarea ulterioară a funcției.

Spre exemplu, să presupunem că dorim să folosim funcția definită anterior pentru calcula suma a două distanțe. Funcția `distanța1` poate fi folosită pentru a rezolva această nouă cerință. Funcția `distanța2` nu poate să facă acest lucru, face doar afișarea distanței:

```
let distanța1 x y = sqrt (x *. x +. y *. y)
let distanța2 x y = Printf.printf "Distanța este %f \n" (sqrt (x *. x +. y *. y))

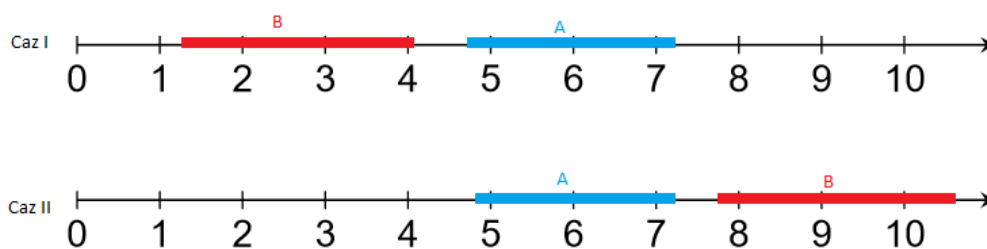
let sum1 = (distanța1 1. 2.) +. (distanța1 2. 1.)
(* Eroare: This expression has type unit but an expression was expected of type float *)
let sum2 = (distanța2 1. 2.) +. (distanța2 2. 1.)
;;
```

Exercițiu rezolvat

Scrieți o funcție care primește ca argument două intervale, și returnează dacă cele două intervale se intersectează. Citiți de la tastatură cele două intervale, și folosiți funcția scrisă anterior pentru a afișa mesajul „Intervalul `[primulIntervalStart, primulIntervalSfarsit]` se intersectează cu intervalul `[alDoileaIntervalStart, alDoileaIntervalSfarsit]`” sau „Intervalul `[primulIntervalStart, primulIntervalSfarsit]` NU se intersectează cu intervalul `[alDoileaIntervalStart, alDoileaIntervalSfarsit]`”

Rezolvare

Pentru început trebuie să ne gândim ce înseamnă că două intervale se intersectează. Puteți găsi o explicație mai detaliată [aici](#). Dar un sumar ar fi: Dacă cele două intervale A și B **nu** se intersectează (sunt disjuncte), atunci fie B este complet în stânga lui A fie este complet în dreapta lui B pe axa numerelor:



Dacă B este stânga lui A, atunci B se termină înainte să înceapă A. Deci începutul lui A este mai mare decât sfârșitul lui B. (`endB < startA`)

Dacă B este în dreapta lui A, atunci B începe după ce A se termină. Deci sfârșitul lui A este mai mic decât începutul lui B. (`endA < startB`)

Acum că știm când cele două intervale sunt disjuncte (`endB < startA || endA < startB`), putem nega această condiție pentru a afla dacă cele două intervale se intersectează. Dacă nu sunt disjuncte, atunci ele se intersectează, rezultând expresia (`not (endB < startA || endA < startB)`).

Să scrie funcția care va determina dacă cele două intervale se intersectează. Care vor fi parametri funcției ? Funcția va trebui să aibă ca parametri începutul și sfârșitul primului interval și începutul și sfârșitul celui de al doilea interval, deci 4 parametri.

```
let intervalIntersect startA endA startB endB = not (endB < startA || endA < startB)
```

Să testăm această funcție cu câteva valori:

```
# let intervalIntersect startA endA startB endB = not (endB < startA || endA < startB)
let t0 = intervalIntersect 0 10 15 20 (* B e la dreapta lui A *)
let t1 = intervalIntersect 0 10 (-10) (-5) (* B e la stanga lui A *)
let t2 = intervalIntersect 0 10 (-10) 20 (* B include pe A *)
let t3 = intervalIntersect 0 10 2 5 (* A include pe B *)
let t4 = intervalIntersect 0 10 2 15 (* A intersecteaza B *)
;;
val intervalIntersect : 'a -> 'b -> 'b -> 'a -> bool = <fun>
val t0 : bool = false
val t1 : bool = false
val t2 : bool = true
val t3 : bool = true
val t4 : bool = true
```

Pare să funcționeze corect.

Să trecem mai departe partea a doua a problemei, care cere să citim de la tastatură începutul și sfârșitul intervalelor și să apelăm funcția cu valorile citite. Pentru citire putem folosi funcția `read_float`, și să reținem valorile în variabile. Apoi trebuie doar să apelăm funcția cu variabilele ca argumente:

```
# let intervalIntersect startA endA startB endB = not (endB < startA || endA < startB)

let a0 = read_float ()
let a1 = read_float ()
let b0 = read_float ()
let b1 = read_float ()
let r = intervalIntersect a0 a1 b0 b1
;;
10
11      Numere introduse de
15      utilizator
20
val intervalIntersect : 'a -> 'b -> 'b -> 'a -> bool = <fun>
val a0 : float = 10.
val a1 : float = 11.
val b0 : float = 15.
val b1 : float = 20.
val r : bool = false
```

Observăm mai sus că numerele citite nu au nici un mesaj înaintea lor ceea ce face dificilă introducerea lor (Utilizator: Trebuie să introduc un număr? Ce număr trebuie să introduc acum?). Pentru a face experiența utilizatorului mai plăcută, am putea să afișăm un mesaj înaintea fiecărei citiri:

```

let () = print_string "Start A:"
let a0 = read_float ()
let () = print_string "End A:"
let a1 = read_float ()
let () = print_string "Start B:"
let b0 = read_float ()
let () = print_string "End B:"
let b1 = read_float ()
let r = intervalIntersect a0 a1 b0 b1

```

Codul de mai sus este funcțional, dar ne obligă să repetăm secvența `let () = print_string mesaj` `let variabilă = read_float ()`. În general dacă observăm o secvență care se repetă este bine să o punem într-o funcție și folosim funcția (Motivele sunt multiple, dacă dorim să modificăm modul în care facem afisarea/citirea avem un singur loc unde trebuie să modificăm, dacă descoperim o problemă trebuie să modificăm într-un singur loc, avem mai puțin de scris, dacă copiem secvența există șanse mai mari să greșim la un moment dat, nu faceți [programare copy-paste](#))

Cum va arăta funcția care să ne ajute cu această citire și afișare ? Funcția va trebui să declare un parametru care să reprezinte mesajul pe care îl afișăm, și să returneze valoarea citită. Funcția va trebui mai întâi să facă afișarea mesajului, după care să facă citirea (folosind secvențierea)

```

let citire_cu_mesaj msg= print_string msg; read_float ()

let a0 = citire_cu_mesaj "Start A:"
let a1 = citire_cu_mesaj "End A:"
let b0 = citire_cu_mesaj "Start B:"
let b1 = citire_cu_mesaj "End B:"

```

Singura cerință rămasă din problemă este să afișăm un mesaj mai prietenos pentru utilizator. Putem face acest lucru folosind expresia condițională (`if`) și funcția `Printf.printf` pentru a formata mesajul.

Punând cap la cap ce am făcut până acum obținem o rezolvare completă:

```

# let intervalIntersect startA endA startB endB = not (endB < startA || endA < startB)
  let citire_cu_mesaj msg= print_string msg; read_float ()

  let a0 = citire_cu_mesaj "Start A:"
  let a1 = citire_cu_mesaj "End A:"
  let b0 = citire_cu_mesaj "Start B:"
  let b1 = citire_cu_mesaj "End B:"

  let () =
    if intervalIntersect a0 a1 b0 b1
    then Printf.printf "Intervalul [%f, %f] se intersectează cu intervalul [%f, %f] \n" a0 a1 b0 b1
    else Printf.printf "Intervalul [%f, %f] NU se intersectează cu intervalul [%f, %f] \n" a0 a1 b0 b1
  ;;

Start A:10
End A:15
Start B:20
End B:25
Intervalul [10.000000, 15.000000] NU se intersectează cu intervalul [20.000000, 25.000000] Afisare
val intervalIntersect : 'a -> 'b -> 'b -> 'a -> bool = <fun>
val citire_cu_mesaj : string -> float = <fun>
val a0 : float = 10.
val a1 : float = 15.
val b0 : float = 20.
val b1 : float = 25.

```

Exercițiu individual: Ce modificare trebuie să facem pentru a citi numere întregi în loc de numere reale. Ce tip are funcția `intervalIntersect` și de ce ?

Exercițiu individual 2: Momentan dacă începutul intervalului este mai mic decât sfârșitul intervalului, codul de mai sus nu va funcționa corect. Modificați codul ca să meargă și dacă începutul și sfârșitul intervalului sunt inversate.

Exerciții

Exercițiul 1: minim/maxim (discutat la curs) Scrieți o funcție care returnează minimul/maximul a trei valori date ca parametri. Folosiți funcțiile predefinite `min` respectiv `max` care funcționează cu orice valori de același tip. Remarcați tipul funcției scrise și verificați că funcționează și cu întregi și cu reali (și chiar cu șiruri), însă nu cu un amestec.

Exercițiul 2: ecuația de gradul 2. Scrieți o funcție care ia ca parametri trei întregi a, b, c și tipărește soluțiile ecuației de gradul doi $ax^2+bx+c=0$, sau un mesaj dacă nu există soluții reale. Folosiți funcția predefinită `sqrt : float -> float` pentru rădăcina pătrată și nu uitați conversiile de la întreg la real unde sunt necesare. Folosiți secvențierea când trebuie tipărite două soluții.

Exercițiul 3: an bisect Scrieți o funcție care determină dacă un an (întreg) dat ca parametru e bisect, returnând un boolean. Dacă un an e bisect sau nu se poate determina după următoarele reguli (va trebui să le reformulați sau reordonați pentru a scrie funcția):

- a) un an divizibil la 4 e bisect, altfel nu
- b) prin excepție de la a), anii divizibili cu 100 nu sunt biseți
- c) prin excepție de la b), anii divizibili cu 400 sunt biseți

Funcții ca valori, argumente și parametri

Un atribut important al unui limbaj funcțional (iar OCaml este un limbaj funcțional) este faptul că funcțiile sunt tratate ca orice alt tip în limbaj. Ce înseamnă asta în mod practic? Înseamnă că putem să pasăm o funcție ca argument la altă funcție, putem să le returnăm ca rezultate din alte funcții, și le putem stoca în variabile și în structuri de date.

Să vedem cum putem stoca o funcție într-o variabilă:

```
# let f x = x + 1
  let g = f
  let rez_f = f 0
  let rez_g = g 0
;;
val f : int -> int = <fun>
val g : int -> int = <fun>
val rez_f : int = 1
val rez_g : int = 1
```

Observăm în codul de mai sus că după ce definim variabila `g`, aceasta are același tip ca și `f`. De asemenea `g` poate fi utilizat la fel ca `f`. În fapt, după ce am definit pe `g`, nu există nici o diferență între folosirea lui `f` sau a lui `g`, cele două simboluri sunt echivalente. De asemenea este important de observat că numele unei funcții pe care îl dăm când o definim nu are nici o semnificație specială.

Funcții ca argumente la alte funcții

După cum am zis în capitolul anterior, o funcție poate fi argument la altă funcție. Să presupunem că dorim să scriem o funcție care returnează valoarea funcției trimisă ca argument pentru valoarea `10` și adună `10` la acest rezultat.

```
# let pt10 h = 10 + h 10;;  
val pt10 : (int -> int) -> int = <fun>
```

În definiția funcției `pt10` avem parametru `h`.

`h` trebuie să fie o funcție care acceptă un întreg și returnează un întreg.

Cum deduce OCaml tipul parametrului `h`?

`h 10` este un apel de funcție, o funcție care acceptă un argument întreg (din cauza valorii `10`). Deoarece valoarea returnată de `h` este folosită cu operatorul pentru întregi `+`, rezultatul lui `h` trebuie să fie de tip întreg. De aici compilatorul OCaml deduce că tipul lui `h` trebuie să fie `int -> int` (adică o funcție care acceptă un întreg și returnează un întreg).

Tipul pentru întreaga funcție `pt10` este `(int -> int) -> int` adică o funcție care acceptă un parametru de tip `(int -> int)` și returnează un `int`.

Cum putem să folosim funcția `pt10`? Argumentul pentru un apel a lui `pt10` poate să fie orice funcție de tipul `int -> int`.

Să definim 3 funcții:

```
# let f x = x + 1  
    let g x = x * x  
    let u x = x *. x;;  
val f : int -> int = <fun>  
val g : int -> int = <fun>  
val u : float -> float = <fun>
```

Bazat pe tipul lor, care dintre cele 3 funcții poate fi parametru pentru `pt10`?

Evident doar `f` și `g` pot fi argumente pentru `pt10` deoarece `u` este de tip `float -> float`.

Să vedem cum arată funcția `pt10` apelată cu `f`, respectiv `g`:

```
# let rf = pt10 f;; (* pt10(f) = 10 + f(10) = 10 + (10 + 1) = 21 *)  
val rf : int = 21  
# let rg = pt10 g;; (* pt10(g) = 10 + g(10) = 10 + (10 * 10) = 110 *)  
val rg : int = 110
```

Merită evidențiat faptul că pasarea unei funcții ca argument nu implică nimic special, sintactic este identică cu apelul unei funcții cu orice altă variabilă (`pt10 f`)

Funcții anonime

De multe ori, când avem o funcție ca argument, nu vom dori să definim funcția înainte trimisă ca argument. Dacă putem să pasăm o valoare ca argument la o funcție pentru alte tipuri (pentru `int` și `float` de exemplu) fără să o punem înainte într-o variabilă înainte, de ce nu ar fi acest lucru posibil pentru

funcții (de exemplu, putem scrie direct `let r = f 10` și nu e necesar să scriem `(let x = 10 let r = f x)`?

Putem defini o funcție oriunde o funcție definită anterior ar fi validă folosind o funcție anonimă. Apelul `pt10 f`, folosind o funcție anonimă, ar arăta astfel:

```
let r = pt10 (fun x -> x + 1);; (* pt10(f) = 10 + (10 + 1) = 21*)
```

Definiția unei funcții anonime este introdusă prin cuvântul cheie `fun` urmată de lista de parametri (separate cu spațiu) urmată de `->`, după care urmează expresia care definește funcția.

```
fun [listă parametri] -> [expresie]
```

Dat fiind faptul că putem folosi o funcție anonimă oriunde am putea folosi o funcție, putem să definim o variabilă care să ia valoarea unei funcții anonime:

```
let fa = fun x -> x + 1
let f x = x + 1
```

Funcțiile `fa` și `f` sunt echivalente, diferă doar prin modul în care le definim, `fa` este definită ca o variabilă căreia îi este atribuită o funcție anonimă, pe când `f` este definită ca funcție. După definiție însă nu va exista nici o diferență între ele, nici în rezultatele pe care le furnizează, nici în modul în care le-am putea folosi ulterior.

Compunerea funcțiilor

Una din cele mai simple și larg folosite operații cu funcții este compunerea lor. În acest fel, putem obține ușor funcții (prelucrări) complexe, pornind de la funcții simple.

În matematică, dacă $f : A \rightarrow B$ și $g : C \rightarrow A$, compunerea $f \circ g : C \rightarrow B$ e definită prin relația $(f \circ g)(x) = f(g(x))$. Deci, pornind de la o valoare $x \in C$ se obține o valoare $g(x) \in A$, și apoi prin aplicarea lui f valoarea $f(g(x)) \in B$.

Cum am defini în OCaml o funcție care să facă compunerea a două funcții?

```
let comp f g x = f (g x);;
```

Să vedem care va fi tipul acestei funcții.

Expresia `g x` este apelul funcției `g` cu argumentul `x`.

Dat fiind faptul că nu există alte constrângeri, compilatorul va atribui un tip generic pentru tipul lui `x` (să denumim acest tip `'c`) și un tip generic pentru tipul rezultatului lui `g` (să denumim acest tip `'a`).

Date fiind aceste tipuri, tipul lui `g` va fi `'c -> 'a`.

Expresia `f (g x)` este din nou un apel de funcție, de data aceasta, al funcției `f` cu rezultatul expresie `g x` ca argument.

Dat fiind faptul că `g` returnează o valoare de tip `'a`, tipul parametrului lui `f` trebuie să fie tot `'a`.

Tipul rezultatului funcției `f` nu are nici o altă constrângere asupra lui, deci i se va atribui un nou tip generic (să îl denumim `'b`).

Deci tipul lui `f` va fi `'a -> 'b`.

Rezultatul funcției `comp` va fi dat de valoarea returnată de `f`, deci va fi `'b`.

Cu aceste informații, putem să construim acum tipul funcției `comp` care va fi `[tipul lui f] -> [tipul lui g] -> [tipul lui x] -> [tipul rezultatului]`.

Făcând înlocuirile, obținem `('a -> 'b) -> ('c -> 'a) -> 'b`.

Să întrebăm interpretorul care este tipul lui `comp` ca să vedem dacă am dedus corect:

```
# let comp f g x = f (g x);;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Să vedem cum am putea utiliza această funcție pentru a compune două alte două funcții:

```
# let comp f g x = f (g x)
  let f x = x + 1
  let g x = x * 2

  let r = comp f g 1 (* comp f g 1 = f (g 1) = f (1 * 2) = f 2 = 2 + 1 = 3 *)
  ;;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val f : int -> int = <fun>
val g : int -> int = <fun>
val r : int = 3
#
```

Mai sus definim două funcții simple `f` și `g` și le pasăm funcției `comp`, împreună cu valoarea pentru care calculăm `(f ∘ g)`.

Deci `r` va fi egal cu `(f ∘ g)(0)`

Exercițiu: Definiți compunerea funcțiilor `f(x) = x + 1` și `g(x) = x * 2` fără să definiți explicit aceste funcții. Definiți `f` și `g` ca funcții anonime când faceți apelul funcției `comp`.

Aplicarea parțială a unei funcții

Aplicarea parțială a unei funcții înseamnă trimiterea

Să analizăm tipul unei funcții cu mai mulți parametri:

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
```

Putem să privim tipul acestei funcții ca o funcție care acceptă mai mulți parametri, după cum am făcut-o până acum. Am putea însă, cu ajutorul unor paranteze să schimbăm perspectiva asupra acestui tip:

```
val f : int -> (int -> int) = <fun>
```

Privită așa, `f` are un parametru `int` și returnează o funcție nouă cu tipul `int -> int`. Această perspectivă nu este un întâmplătoare. OCaml chiar privește orice definiție de funcție cu mai multe argumente, ca pe o funcție cu un argument (primul argument), care returnează o funcție care acceptă restul argumentelor și returnează rezultatul final.

Să vedem această proprietate a funcțiilor cu mai mulți parametri în acțiune:

```
# let f x y = x + y
let aduna10 = f 10 (* echivalent cu aduna10 y = 10 + y (x a fost specificat) *)
let r0 = aduna10 0 (* r0 = f 10 0 = 10 + 0 = 10 *)
let r5 = aduna10 5 (* r5 = f 10 5 = 10 + 5 = 15 *)
;;
val f : int -> int -> int = <fun>
val aduna10 : int -> int = <fun>
val r0 : int = 10
val r5 : int = 15
```

Variabila `aduna10` este rezultatul lui `f 10`. După cum am discutat mai sus, dacă pasăm unei funcții care acceptă mai multe argumente doar primul argument, se va comporta ca o funcție care acceptă un parametru și returnează o funcție care acceptă restul de argumente. În acest caz, funcția `f` (cu tipul `int -> int -> int`) ar accepta două argumente întregi. Specificând doar primul parametru, va rezulta o nouă funcție, care o atribuim variabilei `aduna10`, cu tipul `int -> int`, care va fi tipul lui `aduna10`.

Mai departe, putem folosi funcția `aduna10` ca orice altă funcție care acceptă un argument de tip întreg și returnează un alt întreg.

Confuzii comune

Nu există eroare dacă pasăm prea puține argumente

O consecință a aplicării parțiale a unei funcții este faptul că OCaml nu ne va spune niciodată că nu am pasat suficiente argumente la o funcție (cum s-ar întâmpla în alte limbaje). Acest lucru poate duce la erori care la început sunt surprinzătoare. Să luăm un exemplu simplu:

```
# let f x y = x + y
let r = 10 + f 10;;
Characters 31-35:
let r = 10 + f 10;;
      ^^^^^
Error: This expression has type int -> int
but an expression was expected of type int
#
```

În exemplul de mai sus, vrem să adunăm `10` la rezultatul funcției `f`, dar am pasat prea puține argumente funcției `f` (`f 10`).

Expresia `f 10` va fi o funcție cu tipul `int -> int`.

În consecință, modul în care vede OCaml această expresie este tentativa de a aduna un întreg (`10`) cu o funcție de tip `int->int` (`f 10`).

Eroarea pe care o va raporta interpretorul este că a găsit tipul `int->int` într-un loc unde s-ar fi așteptat să găsească `int`.

Soluția în acest caz este simplă, să pasăm ambele argumente necesare ca apelul lui `f` să rezulte într-un întreg:

```
# let f x y = x + y
let r = 10 + f 10 0;;
val f : int -> int -> int = <fun>
val r : int = 20
```

Erorile pot să apară și în timpul rulării programului. Să presupunem următorul cod:

```
# let f x y = x + y
  let r = min (f 10) (f 5);;
Exception: Invalid_argument "compare: functional value".
```

Codul de mai sus încearcă să calculeze minimum între rezultatele apelului lui `f` cu două valori, dar, din nou, nu specifică suficiente argumente pentru ca `f` să returneze un întreg.

Funcția `min` acceptă orice tip argument, dar va afișa o eroare la rulare (`Exception` denotă o eroare la rulare, vom discuta mai mult despre excepții într-un alt laborator), deoarece compararea a două funcții nu are nici o semnificație.

Funcțiile de scriere

O altă potențială problemă cu aplicarea parțială a funcțiilor apare în cazul funcțiilor de scriere (și în general în cazul funcțiilor care returnează nimic (adică returnează tipul unit)). Să luăm un exemplu simplu:

```
# let o = Printf.printf "Valorile sunt %d si %d \n" 10
;;
val o : int -> unit = <fun>
```

Observăm că am apelat funcția `Printf.printf`, dar nu s-a produs nici o scriere în consolă. Funcția a fost apelată cu un format care necesită două argumente întregi, dar am pasat un singur argument. OCaml nu a considerat acest lucru o eroare, a considerat că am încercat să aplicăm parțial funcția `Printf.printf` și a pus în `0` rezultatul acestei aplicării parțiale. Pentru a evita erori de acest tip, este bine să nu folosim un nume de variabilă pentru rezultatul afișării, ci să declarăm că ne-am aștepta ca rezultatul să fie valoarea `()` (adică valoare tipului unit care semnifică nimic).

```
val o : int -> unit = <fun>
# let () = Printf.printf "Valorile sunt %d si %d \n" 10
;;
Characters 9-53:
    let () = Printf.printf "Valorile sunt %d si %d \n" 10
    ~~~~~
Error: This expression has type int -> unit
      but an expression was expected of type unit
```

Acum avem o eroare de compilare dacă am uitat să specificăm toate argumentele necesare afișării. După ce corectăm eroarea afișarea se produce conform așteptărilor:

```
# let () = Printf.printf "Valorile sunt %d si %d \n" 10 15;;
Valorile sunt 10 si 15
```

Aplicarea parțială și compunerea funcțiilor

Să revizităm subiectul compunerii funcțiilor și să vedem cum am putea să îmbunătățim utilizarea funcției `comp` folosind aplicarea parțială a funcțiilor. Să presupunem că avem 2 funcții `f` și `g` pe care dorim să le compunem și am dori să știm valoarea funcției compuse pentru mai multe valori ale argumentului.

```
let comp f g x = f (g x)
let f x = x + 1
let g x = x * x
let r0 = comp f g 0
let r5 = comp f g 5
let r10 = comp f g 10
;;
```

Observam ca a trebuit sa repetam de multe ori `comp f g`, singurul lucru diferit între liniile care calculează valoarea funcției compuse este valoarea ultimului argument .

Am putea oare să definim, folosind aplicarea parțială a funcțiilor, o funcție care să reprezinte compunerea lui `f` cu `g` și care să accepte doar valoarea pentru care dorim să facem calculul ca argument?

Să analizăm tipul funcției `comp`:

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Dacă am pasa primii doi parametrii la funcția `comp` (parametrii care reprezintă cele două funcții care trebuie compuse) am rămâne cu o nouă funcție care ar accepta cel de-al treilea parametru (care reprezintă valoarea pentru care se face calculul funcției compuse) și ar returna valoarea funcției compuse, adică exact funcția pe care o căutăm.

```
# let comp f g x = f (g x)
let f x = x + 1
let g x = x * x
let fg = comp f g
;;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val f : int -> int = <fun>
val g : int -> int = <fun>
val fg : int -> int = <fun>
```

Am putea fi surprinși la început că funcția `fg` nu are tipul `'c -> 'b`. Dacă eliminăm primii doi parametrii ai funcției `comp` am rămâne cu acest tip. Motivul pentru care `fg` are tipul `int -> int` este că specificarea unui argument pentru o funcție generică (cum este `comp`) poate să „fixeze” un tip generic. Funcția `f` are tipul `int -> int`, această funcție e pasată ca prim argument pentru `comp`. Primul parametru al lui `comp` are tipul `'a -> 'b`, deci tipul `'a` este fixat ca `int` și `'b` este fixat tot ca `int`. Același raționament este aplicabil și pentru al doilea argument `g`. Rezultatul este că funcția `fg` va avea tipul `int -> int` deoarece atât `'c` cât și `'b` au fost fixate la `int` de celelalte două argumente.

Înlocuind `fg` în locurile unde înainte aveam `comp f g` obținem rezultatul final:

```
# let comp f g x = f (g x)
let f x = x + 1
let g x = x * x
let fg = comp f g
let r0 = fg 0
let r5 = fg 5
let r10 = fg 10
;;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val f : int -> int = <fun>
val g : int -> int = <fun>
val fg : int -> int = <fun>
val r0 : int = 1
val r5 : int = 26
val r10 : int = 101
```

Exerciții

Exercițiu: Încercați să apelați `comp` cu o singură funcție. De exemplu:

```
let comp f g x = f (g x)
let f x = x + 1
let h = comp f
;;
```

Ce face funcția `h`? Care este tipul ei? De ce are acest tip? Cum puteți să folosiți funcția `h`?

Capturarea de variabile/parametrii

Până acum funcțiile noastre au folosit în expresiile care le definesc doar funcții predefinite și parametrii. Putem însă folosi orice variabilă sau funcție definită anterior. Spre exemplu:

```
# let add10 x = x + 10
  let factor = 5
  let newFunc x = factor * add10 x
;;
val add10 : int -> int = <fun>
val factor : int = 5
val newFunc : int -> int = <fun>
```

În exemplul de mai sus funcția `newFunc` folosește funcția `add10` definită anterior precum și variabila `factor`. Spunem că `newFunc` a capturat `add10` și `factor`.

Notă: În OCaml putem redefini funcții și variabile. Acest lucru nu influențează funcțiile care au capturat deja funcțiile și variabilele pe care le redefinim. Ex:

```
# let add10 x = x + 10
  let factor = 5
  let newFunc x = factor * add10 x
  (* Apelul inițial a funcției newFunc *)
  let r0 = newFunc 1
  (* Redefinim factor și add10 *)
  let factor = 10
  let add10 x = x + 15
  let newAdd10 = add10 0 (* Afișează 15, varianta nouă de add10 *)
  let newFactor = factor (* factor e 10 acum *)
  let r1 = newFunc 1 (* Rezultatul nu e modificat *)
;;
val newFunc : int -> int = <fun>
val r0 : int = 55
val factor : int = 10
val add10 : int -> int = <fun>
val newAdd10 : int = 15
val newFactor : int = 10
val r1 : int = 55
```

În exemplul de mai sus `r0` și `r1` au aceeași valoare în ciuda faptului că atât `factor` cât și `add10` au fost redefinite. Deoarece `newFunc` a capturat valorile acestor variabile înainte de a fi redefinite, `newFunc` va folosi variantele originale pentru `factor` și `add10`.

Operatorii ca funcții

Din punct de vedere matematic operatorul „+” (spre exemplu, dar este aplicabil pentru toți operatorii) este și el o funcție care acceptă două argumente întregi, și returnează o valoare întreagă :

$+: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}, +(x, y) = x + y$

Și OCaml vede operatorii tot ca funcții, doar cu proprietatea specială că argumentele lor nu trebuie să apară după numele funcției, ci primul argument apare în fața funcției.

Dacă vrem să ne referim la operator ca o funcție oarecare, trebuie să punem numele operatorului între paranteze.

Să vedem tipurile câtorva operatori:

```
# (+);;  
- : int -> int -> int = <fun>  
# (+.);;  
- : float -> float -> float = <fun>  
# ( * );;  
- : int -> int -> int = <fun>
```

Notă: Deoarece `(*)` e începutul unui comentariu și `*)` este sfârșitul unui comentariu, operatorul `*` trebuie pus între paranteze, dar cu cel puțin un spațiu în plus între paranteze și `*` rezultând `(*)`.

Când sunt între paranteze, operatorii se comportă ca orice altă funcție. Ceea ce înseamnă că, printre altele:

- Putem să apelăm operatorul cu ambii parametri la final:

```
# let r = (+) 1 2;;  
val r : int = 3
```

- Putem să aplicăm parțial operatorul:

```
# let add2 = (+) 2  
  let r = add2 3;;  
val add2 : int -> int = <fun>  
val r : int = 5
```

- Putem să pasăm operatorul ca parametru la altă funcție:

```
# let aplica_un_operator_la_patrate op x y = op (x*x) (y*y)  
  let r = aplica_un_operator_la_patrate (+) 2 1;;  
val aplica_un_operator_la_patrate : (int -> int -> 'a) -> int -> int -> 'a = <fun>  
val r : int = 5
```

Exerciții

Exercițiul 4: Valori distincte

Scrieți o funcție cu trei parametri (de același tip oarecare), care returnează câte valori distincte există între argumentele primite (unul, două sau trei) și tipărește, după caz, un mesaj: "toate argumentele sunt distincte/egale" sau "argumentele 1 și 2 (resp. 2 și 3, sau 1 și 3) sunt egale". Evitați pe cât posibil duplicarea de cod: pentru porțiuni de cod similare, creați (și apelați) o funcție care conține partea comună și are ca parametri valorile care diferă.

Exercițiul 5: Mediana

Scrieți o funcție care calculează mediana a trei valori (valoarea aflată între celelalte două).

Încercați să scrieți cod cât mai simplu, și să nu-l repetați. Puteți folosi o funcție auxiliară care calculează mediana a trei numere, pentru care știm că primul e mai mic sau egal decât al doilea. Sau puteți încerca să compuneți doar funcțiile standard `max/min` de două elemente (expresia trebuie să fie oarecum simetrică). Care din variante necesită mai puține comparații?

Exercițiul 6: Operații cu funcții

În matematică, am extins uneori operatorul + de la numere la funcții, definind funcția $f + g$ prin relația $(f + g)(x) = f(x) + g(x)$

a) Definiți în ML o funcție care ia ca parametru două funcții f și g și returnează funcția definită ca suma lor prin relația de mai sus.

b) Scrieți o funcție mai generală, care primește ca parametru și operatorul binar (o funcție de două argumente) care e aplicată (valorilor) celor două funcții. Verificați că o puteți folosi cu operatorul (+) și (*) pentru a calcula suma și produsul.

Exercițiu 7: Tarife cu bonus

În New York, o călătorie cu transportul în comun costă \$2.75. La încărcarea cardului de transport se acceptă doar sume în multipli de 5 cenți. Pentru încărcarea cu valoarea a cel puțin două călătorii se oferă un bonus de 5%, rotunjit la cent. Scrieți o funcție care calculează și returnează suma minimă care trebuie încărcată pentru N călătorii, și afișează restul care rămâne pe card. Puteți verifica rezultatele [aici](#).

