Laborator 2

Variabile locale

Până acum am folosit doar variabile globale, variabile care sunt vizibile din întregul program pe care îl scriem. Variabilele globale se definesc folosind cuvântul cheie `let` , urmat de numele variabilei și expresia de inițializare și sunt accesibile din toate expresiile definite ulterior.

Avem însă posibilitatea de a declara variabile locale, care să nu fie accesibile din tot programul, ci doar într-o anumită expresie, folosind expresia `let..in` ,care are următoarea formă:

```
let variabila = expresie_initializare in expresie_care_folosește_variablia
```

Valoarea și tipul expresiei `let..in` sunt date de expresia de după `in`

Să luăm un exemplu, să definim următoarea funcție :

$$f(x) = \begin{cases} (x^2 + x\ + 1)^2 + x^2,\ x \leq 0 \\ \sqrt{x^2 + x + 1}, x > 0 \end{cases}$$

```
#let f x = if x <= 0.
   then (x *. x +. x +. 1.) *. (x *. x +. x +. 1.) +. x *. x
   else sqrt (x *. x +. x +. 1.)
 let r = f 10.
 let r2 = f (- 10.);;
val f : float -> float = <fun>
val r : float = 10.535653752852738
val r2 : float = 8281.
```

Observăm că expresia `x *. x +. x +. 1.` se repetă de 3 ori. Un programator bun trebuie să fie comod la scris. Așadar, putem să punem rezultatul acestei expresii într-o variabilă locală:

```
# let f x =
    let p  = x *. x +. x +. 1. in  A
    if x <= 0.                     B
     then p *. p +. x *. x
     else sqrt p
  let r = f 10.                    C
  let r2 = f (- 10.);;
val f : float -> float = <fun>
val r : float = 10.535653752852738
val r2 : float = 8281.
```

Blocul **A** definește variabila locală `p`. Această variabilă poate fi folosită doar în blocul **B**. Dacă încercăm să folosim variabila `p` în blocul **C** vom primi o eroare de la interpretor. Valoarea returnată de funcția `f` este dată de expresia de după `in`, adică în acest caz de expresia `if`.

Putem înlănțui mai multe expresii `let..in` pentru a declara mai multe variabile locale, spre exemplu putem folosi o variabilă locală și pentru x^2 (`x*x`):

```
let f x =
    let xx = x *. x in
    let p = xx +. x +. 1. in
    if x <= 0.
      then p *. p +. xx
      else sqrt p
```

Putem folosi xx aici

Putem folosi p aici

Unde putem să folosim o expresie `let..in`?

În exemplele anterioare, expresia `let..in` a fost folosită în expresia de definiție a unei funcții, dar, asemănător cu expresia `if` și expresia secvență de expresii, putem folosi expresia `let..in` oriunde putem să folosim orice altă expresie.

Spre exemplu, să citim două numere de la tastatură și să afișăm suma pătratelor. Putem să folosim faptul că `let..in` este o expresie pentru a face citirea numerelor chiar în cadrul adunării:

```
# let r = (let x = read_int () in x * x) + (let y = read_int() in y * y);;
3
4
val r : int = 25
```

O variabilă locală poate să fie inclusiv o definiție de funcție.

Să luăm un exemplu modificat al funcției definite anterior:

$$f(x) = \begin{cases} ((x+1)^2 + x + 1)^2 + x^2, & x \leq 0 \\ \sqrt{x^2 + x}, & x > 0 \end{cases}$$ b

Putem observa că expresia care se repetă nu mai este `x *. x +. x +. 1.`, ci `y *. y +. y`, dar într-un caz pentru valoarea y = x +. 1. și în celălalt caz pentru y = x.

Putem defini o funcție care să facă acest calcul, în cadrul definiției funcției `f`:

```
let f x =
  let p y  = y *. y +. y  in
  if x <= 0.
    then p (x +. 1.) *. p (x +. 1.) +. x *. x
    else sqrt (p x)
```

Putem evita să apelăm de două ori `p (x +. 1.)` în exemplul de mai sus, folosind o nouă variabilă locală, pe ramura `then` a expresiei `if`.

```
let f x =
  let p y  = y *. y +. y  in
  if x <= 0.
    then let pr = p (x +. 1.) in pr *. pr +. x *. x
    else sqrt (p x)
```

După cum vedem, avem o flexibilitate foarte mare în ceea ce privește locul în care putem introduce o variabilă locală. Este bine, în general, să introducem o variabilă locală când avem nevoie de o valoare în mai multe locuri, sau când dorim să evidențiem că un anumit rezultat are o semnificație aparte.

Vizibilitatea unei variabile

Să facem un rezumat al regulilor de vizibilitate a variabilelor pe care le-am văzut până acum.

Variabilele globale sunt vizibile din orice expresie definită ulterior. Spre exemplu, mai jos folosim variabila globală `r` în definiția variabilei globale `rr`.

```
let r = 0
let rr = r * r
```

Variabilele globale pot să fie redefinite, iar expresiile ulterioare redefinirii vor vedea valoarea redefinită a variabilei:

```
let r = 0
let rr = r * r (* rr va avea valoarea 0 *)
let r = 1 (* r este redefinit *)
let rr2 = r * r (* rr2 va avea valoarea 1 *)
```

Dacă o funcție capturează valoarea unei variabile globale, valoarea folosită va fi cea definită în momentul capturii, nu valoarea redefinită:

```
let r = 0
let f x = x + r (* capturam r *)
let r = 1 (* r este redefinit *)
let rf = f 0 (* valoarea lui rf este 0, deoarece f foloseste valoarea valabila inmomentul definitiei *)
```

O variabilă locală poate să fie folosită doar în expresia de după `in` a expresiei `let..in` care o introduce.

```
let f x = let t = x * x
  in t*t (* t este accesibil aici *)
let error = t (* t nu este acesibil aici *)
```

Putem redefini o variabilă locală:

```
let f x =
  let t = x * x in (* Definim t *)
  let p = t + 1 in (* Folosim t in expresia lui p *)
  let t = p + 1 in (* Redefinim t *)
  t;; (* Folosim varianta redefinit a lui t *)
```

Asemănător cu variabilele globale, variabilele locale pot să fie capturate de funcții, și redefinirea unei variabile locale nu are impact asupra valorii capturate anterior:

```
let f x =
  let t = x * x in (* Definim t *)
  let p x = t + x in (* Folosim t in expresia de definitie a functiei p *)
  let t = 0 in (* Redefinim t *)
  p 0;; (* Folosim p, care a capturat valoarea originala a lui t, nu cea redefinita *)
let r = f 1 ;;
val r : int = 1
```

Exerciţiu: Scrieţi o funcţie care ia ca parametri trei reali a, b, c şi tipăreşte soluţiile ecuaţiei de gradul doi $ax^2+bx+c=0$, sau un mesaj dacă nu există soluţii reale. Folosiţi funcţia predefinită `sqrt : float -> float` pentru rădăcina pătrată şi nu uitaţi conversiile de la întreg la real unde sunt necesare. Note de rezolvare: Folosiţi secvenţierea când trebuie tipărite două soluţii. Folosiţi o variabilă locală pentru delta.

Potrivire de tipare/Pattern matching

OCaml dispune de sintaxă foarte puternică pentru potrivirea de tipare.

În acest laborator, vom introduce sintaxa pentru potrivirea de tipare şi o vom vedea în acţiune pentru numere. În laboratoarele următoare, vom vedea aceeaşi sintaxă folosită pentru liste, tuple şi alte tipuri de date.

Problemă motivatoare:

Vrem să scriem o funcţie care returnează textul în română pentru toate cifrele :

```
# let cifra_to_string c =
    if c = 0 then "zero"
    else if c = 1 then "unu"
    else if c = 2 then "doi"
    else if c = 3 then "trei"
    else if c = 4 then "patru"
    else if c = 5 then "cinci"
    else if c = 6 then "sase"
    else if c = 7 then "sapte"
    else if c = 8 then "opt"
    else if c = 9 then "noua"
    else "Nu e o cifra"
  let r = cifra_to_string 2;;
val cifra_to_string : int -> string = <fun>
val r : string = "doi"
```

Funcţia de mai sus funcţionează, dar conţine foarte multă „ceremonie”. Pentru fiecare caz, trebuie să scriem o nouă expresie `if` şi trebuie să scriem explicit că vrem să comparăm o constantă cu parametrul `c`.

Putem să folosim o expresie `match` pentru a simplifica această funcţie. Funcţia de mai jos va produce aceleaşi rezultate pentru numerele de la 0 la 9:

```
# let cifra_to_string c = match c with
    | 0 -> "zero"
    | 1 -> "unu"
    | 2 -> "doi"
    | 3 -> "trei"
    | 4 -> "patru"
    | 5 -> "cinci"
    | 6 -> "sase"
    | 7 -> "sapte"
    | 8 -> "opt"
    | 9 -> "noua"
  let r = cifra_to_string 2;;
val cifra_to_string : int -> string = <fun>
val r : string = "doi"
```

Expresia `match` are următoarea formă:

```
match [expresie_cu_care_se_compara] with
| [tipar1] -> [expresie_rezultat1]
| [tipar2] -> [expresie_rezultat2]
....
| [tiparn] -> [expresie_rezultatn]
```

expresie_cu_care_se_compara poate să fie orice expresie validă în OCaml. În exemplul cu cifrele, expresia cu care se face comparația este parametrul `c`, dar putem să folosim orice expresie. Spre exemplu, am putea să folosim o variabilă, sau rezultatul unei funcții cu care să facem comparația.

$tipar_1$ ... $tipar_n$ trebuie să fie un tipar compatibil cu tipul expresiei cu care se compară. În exemplul de mai sus, am folosit valori simple pentru tipare.

Dacă valoarea cu care se face comparația se potrivește cu $tipar_x$, atunci valoarea expresiei `match` va fi dată de $expresie\_rezultat_1$. Pentru tipare care sunt valori (cum sunt în exemplul cu cifrele), valoarea care se compară trebuie să fie egală cu valoarea din tipar.

Asemănător cu alte tipuri de expresii compuse pe care le-am văzut până acum (expresia `if`, expresia secvență de expresii și expresia `let..in`), expresia `match` poate și ea să fie folosită în orice loc unde am putea folosi o expresie simplă.

Tiparul „orice”

Dacă scriem funcția `cifra_to_string` în interpretor în forma pe care am definit-o mai sus, vom vedea următorul avertisment:

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
10
```

Dacă folosim potrivirea de tipare, OCaml va verifica faptul că tratăm toate cazurile posibile. Este de dorit să nu ignorăm aceste avertismente. De obicei, sunt un semn că nu am făcut ceva bine.

Evident, ar dura destul de mult să tratăm toate cazurile tipului întreg, folosind doar valori (valoarea maximă pentru `int` este `4611686018427387903` în OCaml).

Din fericire, nu trebuie să specificăm în tipar toate valorile posibile pentru `c`. Putem specifica un tipar care se potrivește cu orice valoare. Acest tipar este definit cu ajutorul caracterului `_`.

```
# let cifra_to_string c = match c with
    | 0 -> "zero"
    | 1 -> "unu"
    | 2 -> "doi"
    | 3 -> "trei"
    | 4 -> "patru"
    | 5 -> "cinci"
    | 6 -> "sase"
    | 7 -> "sapte"
    | 8 -> "opt"
    | 9 -> "noua"
    | _ -> "Nu e o cifra"
  let r = cifra_to_string 2;;
val cifra_to_string : int -> string = <fun>
val r : string = "doi"
```

Ordinea tiparelor este importantă. Primul tipar întâlnit, care se potrivește cu valoarea expresiei cu care se face comparația, va da rezultatul expresiei `match`. Spre exemplu, următoarea funcție va returna întotdeauna textul `"Nu e o cifra"` deoarece tiparul orice apare primul.

```
let cifra_to_string c = match c with
  | _ -> "Nu e o cifra"
  | 0 -> "zero"
  | 1 -> "unu"
  | 2 -> "doi"
  | 3 -> "trei"
  | 4 -> "patru"
  | 5 -> "cinci"
  | 6 -> "sase"
  | 7 -> "sapte"
  | 8 -> "opt"
  | 9 -> "noua"
let r = cifra_to_string 2;;
```

OCaml ne avertizează chiar că tiparele de sub tiparul „orice” nu vor fi folosite (deoarece tiparul „orice” se va potrivi cu orice număr, iar tiparele aflate sub el nu vor fi niciodată evaluate, pentru că a fost găsită deja o potrivire, și anume potrivirea cu „orice”).

```
Characters 68-69:
    | 0 -> "zero"
      ^
Warning 11: this match case is unused.
```

Funcții cu potrivire de tipar

Uneori o funcție nu face altceva decât să potrivească valoarea unui parametru cu o listă de tipare. OCaml are o sintaxă specială pentru o astfel de funcție, introdusă de cuvântul cheie `function`.

Cele două funcții de mai jos sunt echivalente:

```
# let cifra_to_string = function
    | 0 -> "zero"
    | 1 -> "unu"
    | 2 -> "doi"
    | 3 -> "trei"
    | 4 -> "patru"
    | 5 -> "cinci"
    | 6 -> "sase"
    | 7 -> "sapte"
    | 8 -> "opt"
    | 9 -> "noua"
    | _ -> "Nu e o cifra"
  let r = cifra_to_string 2;;
val cifra_to_string : int -> string = <fun>
val r : string = "doi"
```

```
# let cifra_to_string c = match c with
    | 0 -> "zero"
    | 1 -> "unu"
    | 2 -> "doi"
    | 3 -> "trei"
    | 4 -> "patru"
    | 5 -> "cinci"
    | 6 -> "sase"
    | 7 -> "sapte"
    | 8 -> "opt"
    | 9 -> "noua"
    | _ -> "Nu e o cifra"
  let r = cifra_to_string 2;;
val cifra_to_string : int -> string = <fun>
val r : string = "doi"
```

Cuvântul cheie `function` definește o funcție cu un singur parametru (care nu va avea un nume explicit), care va fi potrivit cu lista de tipare care urmează după cuvântul cheie `function`.

Tipar multiplu

Sa definim funcția următoare :

$$f(x) = \begin{cases} 0, & x = 0,1 \\ x^2, & altfel \end{cases}$$

```
# let f x = match x with
    | 0 -> 0
    | 1 -> 0
    | _ -> x
  let r0 = f 0
  let r1 = f 1
  let r2 = f 2;;
val f : int -> int = <fun>
val r0 : int = 0
val r1 : int = 0
val r2 : int = 2
```

Observăm că repetăm expresia `-> 0` atât pentru tiparul `0` cât și pentru tiparul `1`. Sintaxa pentru potrivirea de tipare permite să specificăm mai multe tipare care să returneze aceeași expresie rezultat:

```
let f x = match x with
  | 0 | 1 -> 0
  | _ -> x
let r0 = f 0
let r1 = f 1
let r2 = f 2;;
```

Luând în considerare această nouă posibilitate, expresia `match`, la modul general, poate să arate în felul următor:

```
match [expresie_cu_care_se_compara] with
| [tipar1,0] ... | [tipar1,m1] -> [expresie_rezultat1]
| [tipar2,0] ... | [tipar2,m2] -> [expresie_rezultat2]
....
| [tipar_n,0] ... | [tipar_n,mn] -> [expresie_rezultatn]
```

Definirea de noi variabile în tipar

Putem să salvăm valoarea care a fost potrivită de un tipar într-o valoare folosind sintaxa *`tipar as nume_variabila`*. Această opțiune este utilă în special la tipare mai complexe (pe care le vom vedea mai târziu), la tipare multiple, sau când folosim o funcție cu potrivire de tipar (`function`), unde nu avem un nume care ne permite să accesăm valoarea care este potrivită.

Să definim următoarea funcție:

$$f(x) = \begin{cases} x * 10, & x = -1, 1 \\ x^2, & altfel \end{cases}$$

```
let f = function
  | -1 | 1 as u -> u * 10
  | _ as x -> x * x
```

În exemplul de mai sus, am definit două variabile `u` și `x`.

Variabila `u` va avea valorile care se potrivesc cu tiparele `1` și `-1`, adică valorile `1` și `-1`.

Variabila `x` va avea valorile care se potrivesc cu tiparul „orice” (`_`), deci va avea orice valoare, în afară de valorile care se potrivesc cu tiparele anterioare (adică orice valoare în afară de `1` și `-1`).

Pentru tiparul `_ as x` putem simplifica expresia folosind direct `x` fără a specifica explicit tiparul „orice” (`_`).

Folosirea lui `x` în tipar va avea ca efect declararea acestei variabile locale.

Dat fiind faptul că nu avem nici o altă restricție asupra lui `x`, orice număr se a potrivi cu un tipar care este o declarație de variabilă. Folosind o variabilă pe post de tipar putem simplifica funcția anterioară:

```
let f = function
  | -1 | 1 as u -> u * 10
  | x -> x * x
```

Condiții suplimentare pentru un tipar

Pe lângă tipar, putem defini și o condiție suplimentară, care să fie evaluată, dacă tiparul se potrivește. Condiția poate să folosească atât variabile definite în tipar, cât și variabile și parametri definiți anterior. Sintaxa pentru condiție suplimentară este: `tipar when expresie_booleana`.

Să definim în OCaml următoarea funcție:

$$f(x,c) = \begin{cases} \frac{x}{2c}, & x \text{ par și } c <> 0 \\ \frac{x}{2}, & x \text{ par și } c = 0 \\ \frac{x}{2c}, & x \text{ impar și } c <> 0 \\ \frac{x}{2c}, & x \text{ impar și } c = 0 \end{cases}$$

```
let f c x = match x with
  | par when c = 0 && par mod 2= 0 -> par / 2
  | par when par mod 2 = 0 ->  par / c / 2 (* cazul c = 0 e tratat de cazul anterior, deci nu trebuie conditia c <> 0 *)
  | impar when c = 0 -> (impar +1) /2 (* cazul x este par e deja tratat de  cazurile anterioare *)
 (* ultimul caz, c <> 0, din cauza tiparului anterior, iar cazul în care x e par e deja exclus de primele cazuri *)
  | impar -> (impar +1) / c /2
```

Exercițiu:

Se dau următoarele două funcții în OCaml:

```
let f c x = match x with
  | par when c = 0 && par mod 2= 0 -> par / 2
  | par when par mod 2 = 0 ->  par / c / 2
  | impar when c = 0 -> (impar +1) /2
  | impar -> (impar +1) / c /2


let f2 c = function
  | par when c = 0 && par mod 2= 0 -> par / 2
  | par when par mod 2 = 0 ->  par / c / 2
  | impar when c = 0 -> (impar +1) /2
  | impar -> (impar +1) /c /2
```

Care sunt tipurile celor două funcții? Câte argumente acceptă cele două funcții? Există vreo diferență între cele două funcții din punct de vedere al modului în care le putem utiliza și a valorilor pe care le returnează ? Sunt ambele rezolvări valide pentru problema de la paragraful anterior ? (Consultați laboratorul 1, Tipul unei funcții)

Funcții recursive

Ce este o păpușă Matrioșka ? O păpușă goală pe interior care conține în ea o altă păpușă Matrioșka, mai mică.

Definiția de mai sus este o definiție recursivă. Definiția unei păpuși Matrioșka conține însuși termenul pe care îl definim.

Desigur, definiția de mai sus nu este completă. La un moment dat, păpușa va deveni prea mică să permită existența unei alte păpuși înăuntru.

O definiție mai aproape de realitate ar fi:

O păpușă Matrioșka este o păpușă goală pe interior, care conține în ea fie o altă păpușă Matrioșka mai mică, fie, dacă dimensiunea păpușii superioare este mai mică decât x cm, va conține o păpușă plină.

De aici putem deducem câteva proprietăți necesare pentru o definiție recursivă corectă:

1. Un caz de bază (Păpușa plină)
2. O condiție pentru care se aplică cazul de bază (dimensiunea e mai mică decât x cm)
3. O expresie recursivă (o păpușă Matrioșka conține o păpușă Matrioșka)

Asemănător cu această definiție, multe probleme matematice pot fi privite în mod recursiv.

Care este suma primelor `n` numere naturale ?

Este fie suma primelor `n-1` numere naturale la care adăugăm `n` , sau 0, dacă `n` este 0

$$\sum_{k=1}^{n} k = \begin{cases} n + \sum_{k=1}^{n-1} k, & n > 0 \\ 0, & n = 0 \end{cases}$$

Sau, scris mai simplu, $s(n) = \begin{cases} n + s(n-1), n > 0 \\ 0, n = 0 \end{cases}$

OCaml ne permite să definim o astfel de funcție recursivă, dar trebuie să o introducem cu cuvintele cheie `let rec`:

```
# let rec sum n = if n = 0
      then  0
      else n + sum (n-1)
  let r = sum 3;;
val sum : int -> int = <fun>
val r : int = 6
```

Funcția `sum` este of funcție recursivă deoarece în expresia de definiție a lui `sum` apare un apel la funcția `sum` (aceeași funcție pe care o definim).

Cuvântul cheie `rec` schimbă vizibilitatea funcției care se declară, pentru ca acesta să fie vizibilă în cadrul propriei expresii de definiție. Dacă omitem acest cuvânt cheie, funcția pe care o declarăm nu va fi accesibilă în expresia de definiție și vom avea o eroare:

```
# let sum n = if n = 0
      then  0
      else n + sum (n-1);;
Characters 47-50:
      else n + sum (n-1);;
               ^^^
Error: Unbound value sum
```

Practic, interpretorul nu va ști ce reprezintă `sum` în expresia `sum (n-1)`, întrucât `sum` nu a fost încă definit. Folosirea lui `let rec` va rezolva această eroare.

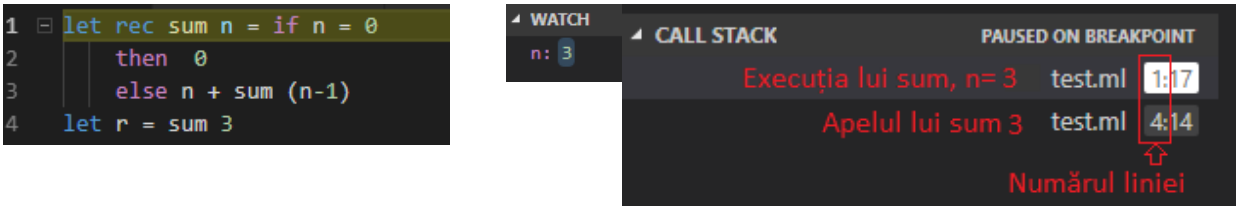
Rulare pas cu pas

Să vedem cum se execută această funcție:

Pas 1: Execuția programului ajunge la apelul `sum 3`

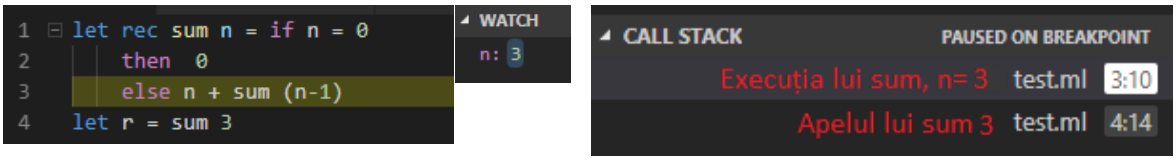
```
1  let rec sum n = if n = 0
2      then  0
3      else n + sum (n-1)
4  let r = sum 3
```

Pas 2: Pentru a calcula valoarea lui `sum 3`, programul va începe evaluarea expresiei de definiție a lui `sum` pentru valoarea lui `n = 3` (valoarea pe care o vedem în poza în secțiunea Watch). De asemenea, programul va ține minte, în ceea ce se numește stiva de apeluri (call stack), unde trebuie să returneze valoarea (care este locația în program unde trebuie să se întoarcă, după ce a calculat valoarea cerută).



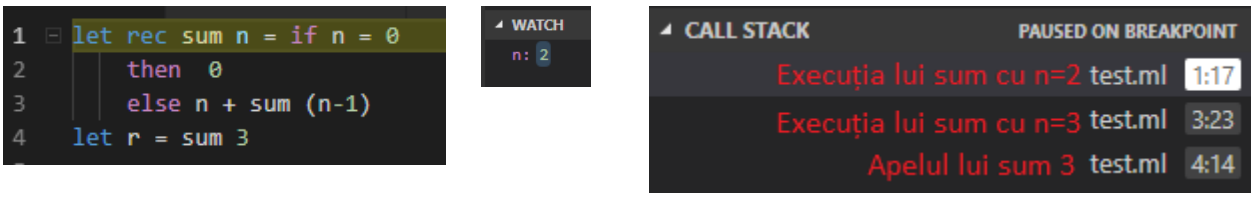
În exemplul de mai sus, stiva de apeluri va avea două intrări: una pentru locația unde valoarea trebuie returnată (locația lui `sum 3` de la linia 4), și o intrare pentru instrucțiunea care este în execuție în acest moment (linia 1, la începutul expresiei `if`).

Pas 3: Programul va evalua condiția din expresia `if` și va decide că trebuie să meargă pe ramura `else` (deoarece `n <> 0`). Observăm că locația curentă din stivă se mută la linia `3`.

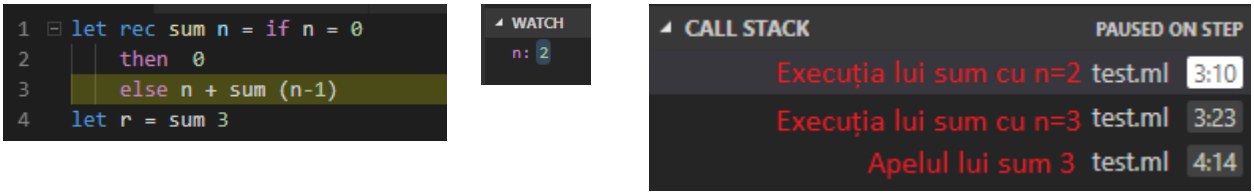


Pas 4: În evaluarea expresiei `n + sum (n-1)`, programul va avea nevoie de valoarea pentru `sum (n-1)`, deci va începe execuția lui `sum` pentru `n = 3 - 1 = 2`.

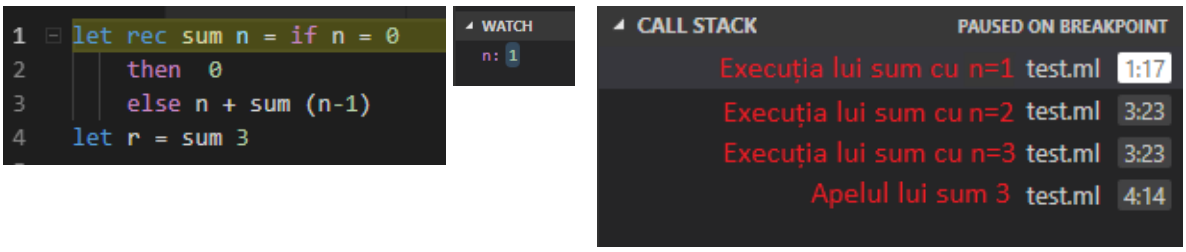
Observăm că, în stiva de apeluri, a fost adăugată o nouă intrare pentru noul apel al lui `sum` cu argumentul `2`. Execuția lui `sum` pentru `n=3` a fost oprită și aceasta va aștepta valoarea lui `sum 2`.



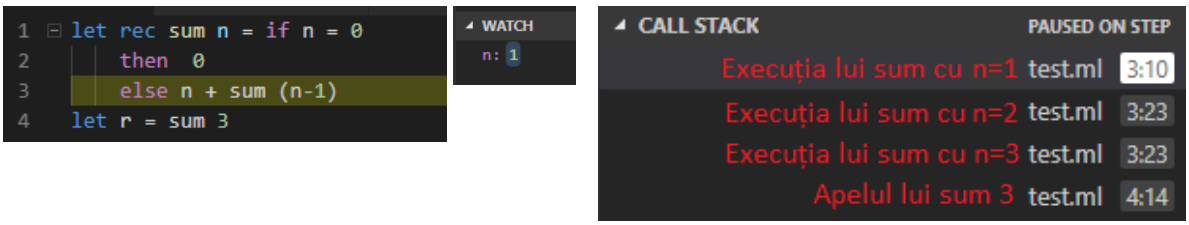
Pas 5: În execuția lui `sum 2`, din nou se va evalua condiția din `if` și se va alege ramura `else`, deoarece `n <> 0`.



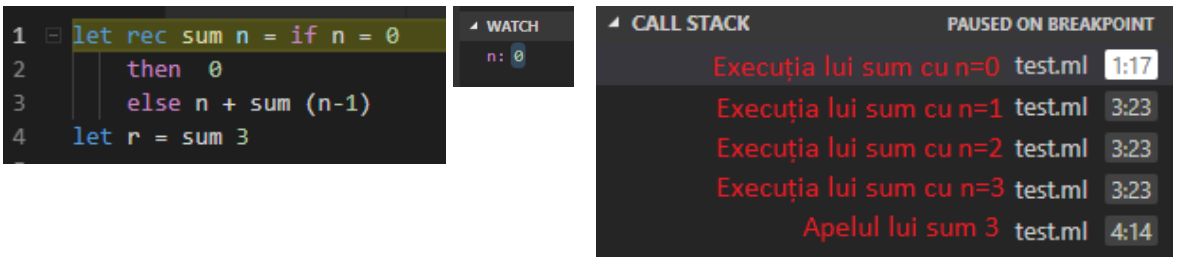
Pas 6: Asemănător cu pasul 4, în evaluarea expresiei `n + sum (n-1)`, programul va avea nevoie de valoarea pentru `sum (n-1)`, deci va începe execuția lui `sum` pentru `n = 2 - 1 = 1`. Observăm că, în stiva de apeluri, a fost adăugată o nouă intrare pentru noul apel al lui `sum` cu argumentul `1`. Execuția lui `sum` pentru `n=2` a fost oprită și așteaptă valoarea lui `sum 1`.



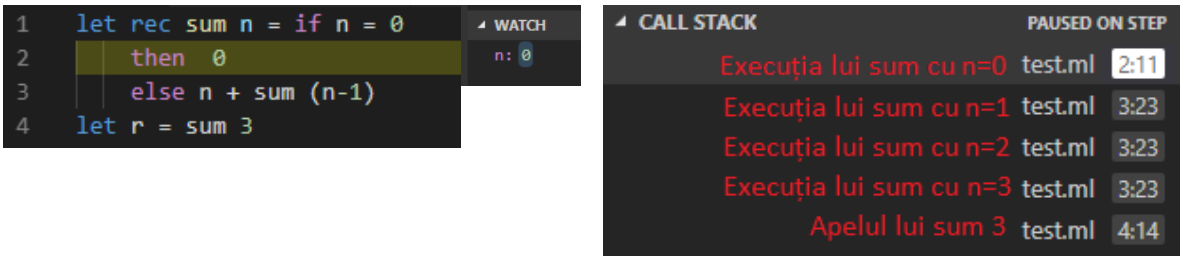
Pas 7: În execuția lui `sum 1`, din nou, se va evalua condiția din `if` și se va alege ramura `else`, deoarece `n <> 0`.



Pas 8: Asemănător cu pasul 4 și 6, în evaluarea expresiei `n + sum (n-1)`, programul va avea nevoie de valoarea pentru `sum (n-1)`, deci va începe execuția lui `sum` pentru `n = 1 - 0 = 0`. Observăm că, în stiva de apeluri, a fost adăugată o nouă intrare pentru noul apel al lui `sum` cu argumentul `0`. Execuția lui `sum` pentru `n=1` a fost oprită și așteaptă valoarea lui `sum 0`.



Pas 9: În execuția lui `sum 0`, din nou, se va evalua condiția din `if`. De data aceasta, se va alege ramura `then`, deoarece `n = 0`. Valoarea returnată pentru apelul `sum 0` va fi astfel `0`.



Deoarece nu a fost nevoie de un nou apel recursiv al funcției `sum`, programul poate să înceapă să calculeze valorile pentru toate expresiile care sunt în așteptare (expresiile care și-au oprit execuția în pașii 4, 6 și 8 pot acum să termine de calculat valoarea).

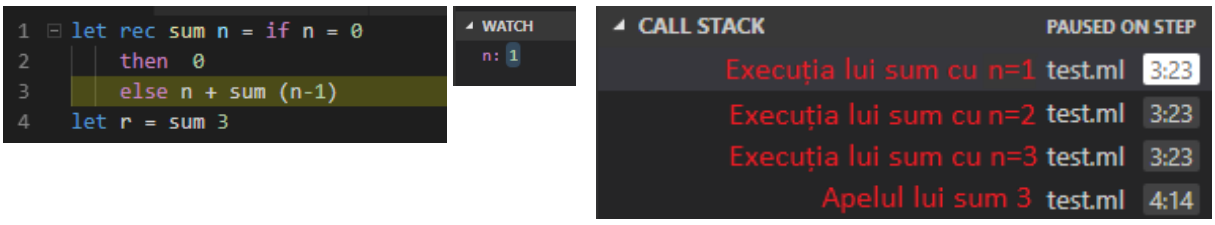
Pas 10: Expresia care s-a oprit din execuție la pasul 8 în cadrul apelului lui `sum` cu `n = 1`, poate acum să fie evaluată.

Expresia `n + sum (n-1)`, pt `n = 1`, va fi `1 + sum 0`.

`sum 0` a fost calculat anterior, deci rezultatul apelului `sum 1` va fi `1`.

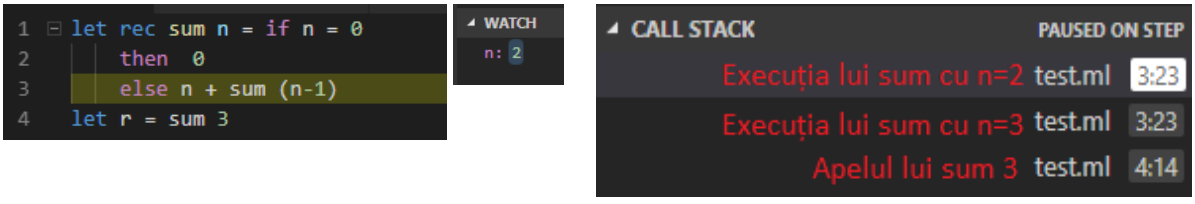
Observăm că din stiva de execuție a dispărut apelul pentru `sum` cu `n = 0`.

Execuția apelului `sum 0` s-a terminat cu succes, rezultatul a fost returnat expresiei care a cerut valoarea, deci intrarea respectivă din stivă nu mai este necesară și este în consecință ștearsă.



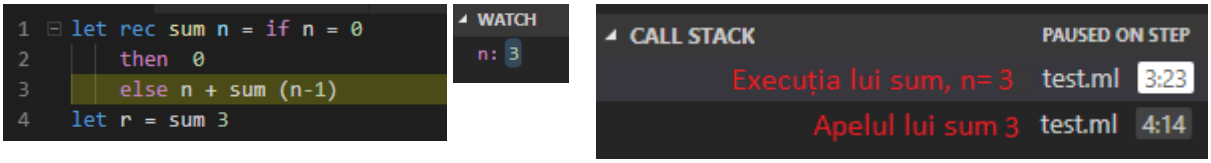
Pas 11: Expresia care s-a oprit din execuție la pasul **6** în cadrul apelului lui `sum` cu `n = 2`, poate acum să fie evaluată. Expresia `n + sum (n-1)`, pt `n = 2`, va fi `2 + sum 1`.

`sum 1` a fost anterior calculat, deci rezultatul lui `sum 2` va fi `3`.

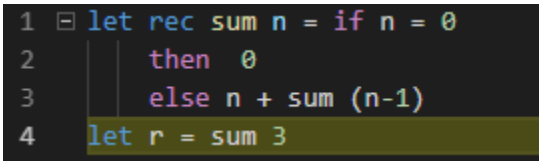


Pas 12: Expresia care s-a oprit din execuție la pasul **4,** în cadrul apelului lui `sum` cu `n = 3`, poate acum să fie evaluată. Expresia `n + sum (n-1)`, pt `n = 3`, va fi `3 + sum 2`.

`sum 2` a fost anterior calculat, deci rezultatul lui `sum 3` va fi `6`.



Pas 13: Valoarea apelului `sum 3` este returnată expresiei care a cerut-o, adică expresiei de inițializare a variabilei `r`, variabila `r` luând astfel valoarea lui `sum 3`.



Notă: Mai sus am prezentat modul în care OCaml execută o funcție recursivă. Conceptele prezentate mai sus sunt aplicabile și în cazul apelurilor nerecursive (programul va trebui să țină minte unde trebuie să se întoarcă după apelul unei funcții). Mai mult, conceptul de stivă de apeluri (call stack) este comun tuturor limbajelor de programare moderne. Înțelegerea modului în care funcțiile sunt apelate este esențială pentru orice programator, indiferent de limbajul cu care va lucra pe viitor.

Funcția de mai sus poate să fie scrisă în diverse feluri, folosind elementele de sintaxă pe care le-am văzut până acum. Putem profita de capacitatea limbajului de a potrivi tipare pentru a simplifica funcția `sum`:

Cu `if`

```
let rec sum n = if n = 0
    then  0
    else n + sum (n-1);;
```

Cu `match`

```
let rec sum n = match n with
    | 0 -> 0
    | _ -> n + sum (n-1);;
```

Cu `function`

```
let rec sum = function
    | 0 -> 0
    | n -> n + sum (n-1);;
```

Cele 3 moduri de a scrie funcția `sum` sunt echivalente. În orice fel scriem funcția, ea se va comporta la fel. Modul pe care îl alegem depinde de preferința personală a programatorului. Programatorii cu experiență preferă de obicei o sintaxă mai concisă, care ne permite să ne concentrăm pe logica codului pe care îl scriem (sintaxa cu `function` va fi preferată). Un programator novice ar putea să fie indus în eroare de parametrul ascuns care îl introduce `function`, și ar putea să prefere simplitatea implementării cu `if`. Oricare sintaxă alegeți, asigurați-vă însă că vă este clar ce face.

Exerciții

1. **Factorial**

Scrieți o funcție care să calculeze n! = 1 * 2 * 3 * ...* (n - 1) * n

2. **Cel mai mare divizor comun**

Știind că cmmdc(a, b) = cmmdc(b, a mod b) dacă b ≠ 0, scrieți o funcție recursivă pentru cel mai mare divizor comun. Care e cazul de bază ?

3. **Aplicarea repetată a unei funcții**

în laboratorul trecut am scris funcții de ordin superior (funcționale) care aplicau o funcție de 2, 3, 4 ori. Definiți (recursiv) o funcție care ia ca parametru un întreg n și o funcție, și returnează funcția compusă cu ea însăși de n ori.

4. **Cifrele unui număr**

Un număr e reprezentat uzual în scris ca un șir de cifre în baza 10.

Un șir e o noțiune recursivă (un element, sau un șir urmat de un element).

Putem spune atunci că un număr n e fie o singură cifră, fie ultima cifră (n mod 10) precedată de alt număr (n / 10).

Folosind această definiție scrieți funcții recursive care calculează: suma cifrelor unui număr, numărul de cifre, produsul lor, cifra maximă / minimă, etc.

5. **Resturi modulo p**

În matematică știm că dacă p e un număr prim, și a nu se divide cu p, atunci șirul a, a2, a3, ... va ajunge la 1, luând numerele modulo p (adică resturile la împărțirea cu p).

De exemplu, fie p = 7 și a = 4. Atunci a2 = 16 ≡ 2 (mod 7), și a3 = a2 * a ≡ 2 * 4 ≡ 1 (mod 7).

(Se spune că mulțimea resturilor nenule modulo p prim formează un grup multiplicativ.)

Scrieți o funcție care ia ca parametru un număr întreg a și un număr p (presupus prim) și returnează cea mai mică putere n pentru care an ≡ 1 mod p (sau returnează 0 dacă a se divide cu p).

Indicație: scrieți o funcție auxiliară care mai are ca parametri și exponentul k respectiv valoarea ak (mod p), și care se apelează recursiv până când ak ≡ 1 (mod p).

Depanarea programelor (opțional)

Văzând pozele de la rularea pas cu pas din capitolul anterior, s-ar putea să vă întrebați: „De unde sunt aceste poze? Au fost oare doar un exercițiu de editare de imagini din partea autorului?” Răspunsul este nu. Rularea pas cu pas este un instrument foarte important, atât în înțelegerea codului cât și în găsirea erorilor. Să vedem cum putem să rulăm un program pas cu pas.

Notă: Cu toate că unele din instrucțiunile care urmează sunt specifice pentru OCaml, majoritatea instrucțiunilor sunt aplicabile în depanării oricărui limbaj, în editorul Visual Studio Code. Mai mult, majoritatea conceptelor prezentate le veți regăsii în toate mediile de dezvoltare moderne.

Pregătirea proiectului

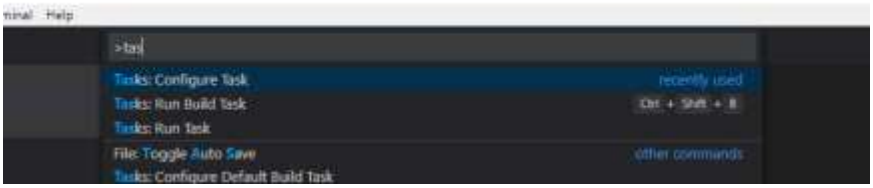
Până acum, am rulat codul în interpretor. Interpretorul nu ne permite să depanăm și să executăm pas cu pas codul executat. Pentru aceasta, trebuie să folosim compilatorul OCaml.

Compilatorul, spre deosebire de interpretor, va genera un fișier executabil (pe Windows un fișier .exe). Acest fișier executabil poate fi rulat de sine stătător pe orice calculator compatibil (chiar dacă nu are

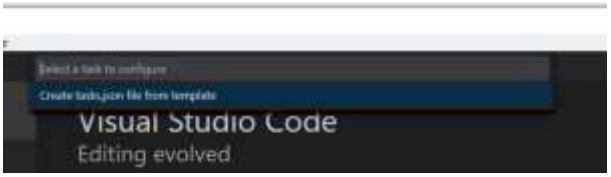
instalat mediul de dezvoltare pentru OCaml). Cu ajutorul unui program specializat (denumit `ocamldebug`), putem să executăm fișierul executabil pas cu pas, și să inspectăm starea programului.

Programul `ocamldebug` este un program de linie de comandă. Din fericire, nu trebuie să interacționăm direct cu acest utilitar. Visual Studio Code va interacționa cu `ocamldebug` și ne va oferi o interfață prietenoasă pentru execuția pas cu pas.

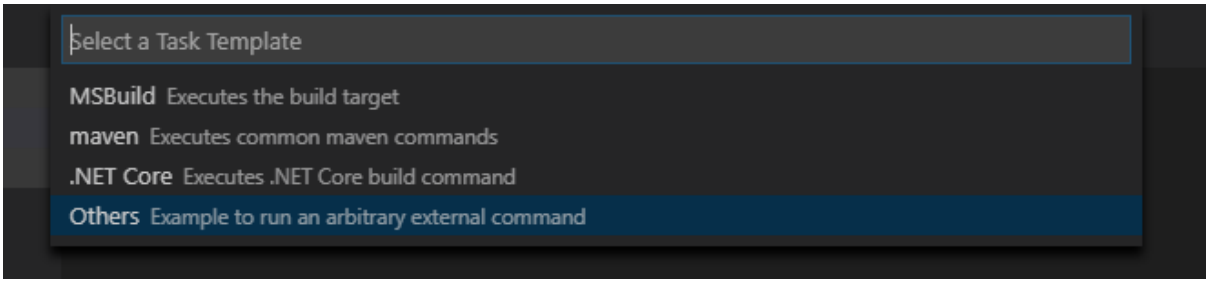
După ce deschideți un director în Visual Studio Code, deschideți paleta de comenzi (Ctrl + Shift + P) și căutați comanda „Configure Task”:



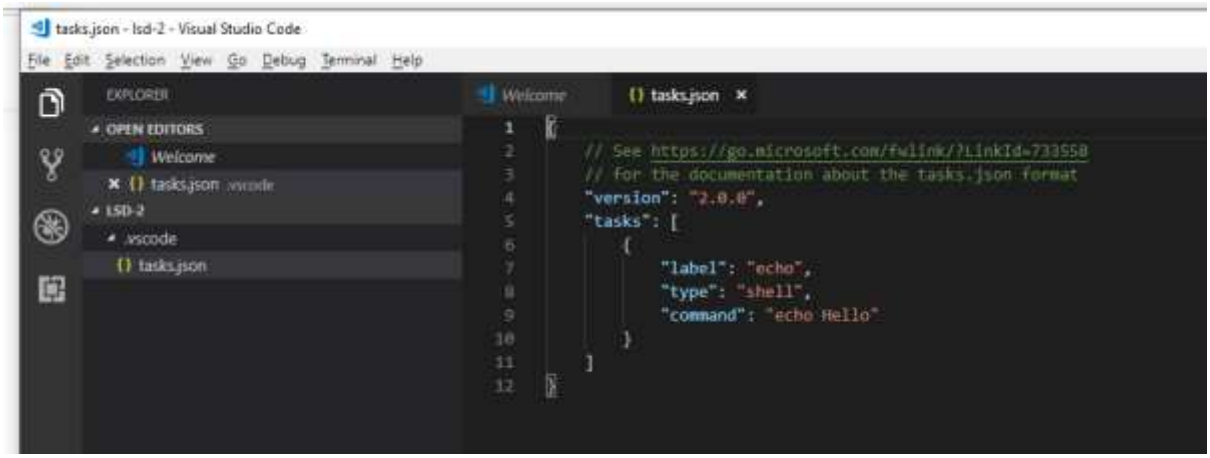
Veți fi întrebați dacă doriți să creați un nou fișier `tasks.json` , după un șablon (template). Apăsați tasta „enter” pentru a selecta opțiunea.



Veți fi apoi întrebați care să fie șablonul utilizat, selectați „Other” din listă și apăsați apoi tasta „enter”:



Ar trebui să de deschidă fișierul `tasks.json` nou creat:

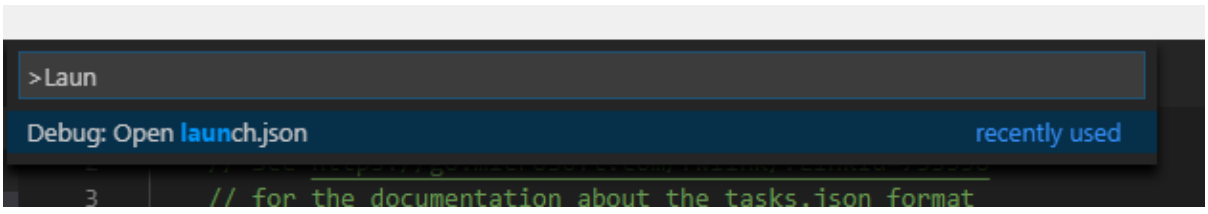


Fișierul `tasks.json` conține instrucțiuni despre cum trebuie Visual Studio Code să invoce compilatorul pentru a „construi” (build) fișierul executabil.

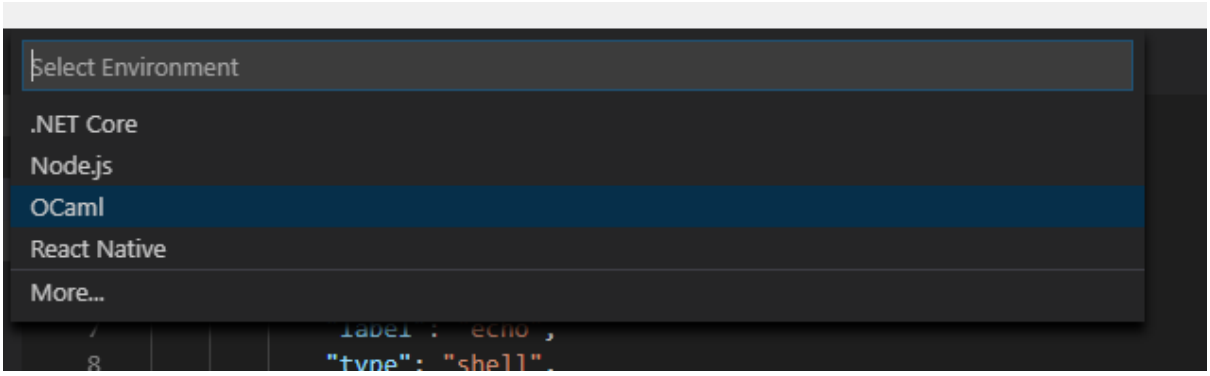
Înlocuiți conținutul fișierului cu:

```
{
    "version": "2.0.0",
    "tasks": [{
        "label": "build",
        "command": "ocamlc",
        "args": [
            "-g", "-o", "${fileBasenameNoExtension}.exe",
            "${relativeFile}"
        ],
        "options": {
            "cwd": "${workspaceRoot}"
        },
        "problemMatcher": "$ocamlc"
    }]
}
```

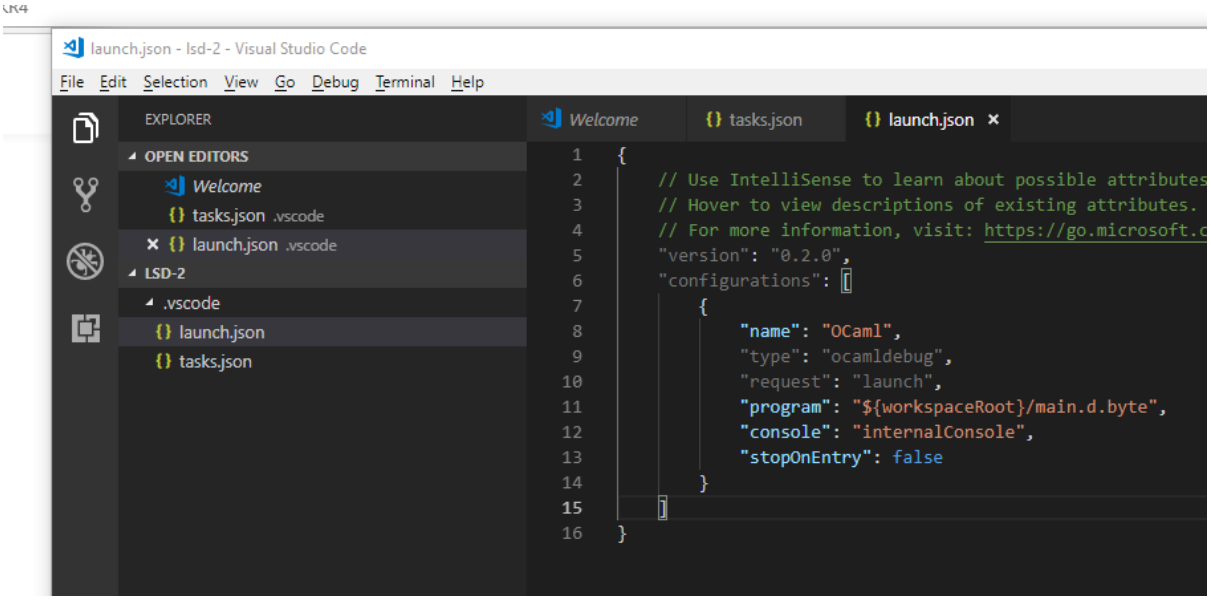
Deschideți din nou paleta de comenzi (Ctrl+Shift+P), căutați comanda „Open launch.json” și apoi apăsați tasta „Enter”



Din lista care apare, selectați „OCaml” și apăsați tasta „Enter”.



Ar trebui să se fi deschis un nou fișier numit `launch.json`.



Fișierul `launch.json` va conține informații despre cum trebuie Visual Studio Code să pornească execuția programului pentru depanare.

Înlocuiți conținutul fișierului cu:

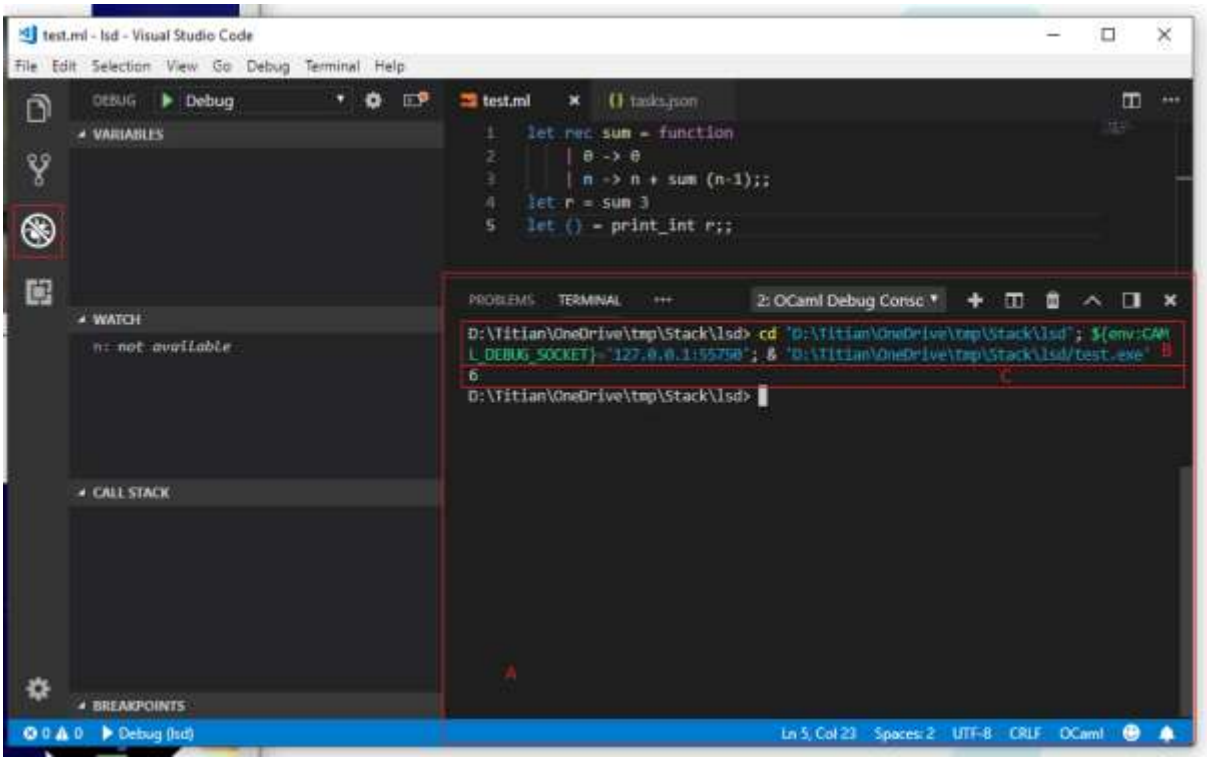
```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "ocamldebug",
            "request": "launch",
            "program": "${workspaceRoot}/${fileBasenameNoExtension}.exe",
            "name": "Debug",
            "console": "integratedTerminal",
            "preLaunchTask": "build"
        }
    ]
}
```

Notă: Cele două fișiere (`launch.json` și `tasks.json`) sunt puse într-un director numit `.vscode`. După ce ați trecut prin procesul anterior, puteți să copiați acest director în orice alt director pe care îl deschideți în VS Code.

Rularea programului

Deschideți un fișier `.ml` în directorul pe care l-ați configurat anterior. Pentru a compila și rula programul, apăsați tasta `F5`.

Când rulăm programul, se va selecta perspectiva de depanare (). Observăm că, în partea de jos a ferestrei, s-a deschis consola de depanare OCaml (marcată cu A). Primul rând va conține comanda care pornește execuția (marcată cu B). Următoarele rânduri vor conține ce a scris programul în consolă.



Notă: Există o diferență între rularea programului compilat și rularea în interpretor. Interpretorul va afișa valoarea tuturor variabilelor globale declarate. Programul compilat nu va face acest lucru. Programul compilat va afișa doar ce afișăm în mod explicit cu una din funcțiile dedicate de afișare.

Spre exemplu, așa arată codul de mai sus rulat în interpretor, respectiv prin programul compilat:

Rulat în interpretor	**Rulat în program compilat**

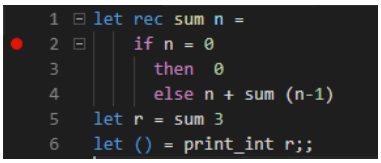
```
        OCaml version 4.06.1

# let rec sum n = if n = 0
      then  0
      else n + sum (n-1)
  let r = sum 3
  let () = print_int r;;
6val sum : int -> int = <fun>
val r : int = 6
```

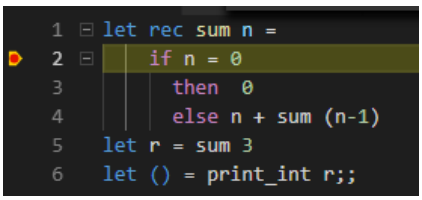
```
D:\Titian\OneDrive\tmp\Stack\lsd> cd 'D:\Titian\O
ML_DEBUG_SOCKET}='127.0.0.1:55750'; & 'D:\Titian\
ML_DEBUG_SOCKET}='127.0.0.1:55750'; & 'D:\Titian\
OneDrive\tmp\Stack\lsd/test.exe'
6
D:\Titian\OneDrive\tmp\Stack\lsd>
```

Puncte de oprire

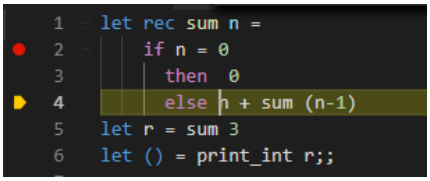
Până acum, rularea programului s-a petrecut de la cap la coadă. Ar fi util să putem opri rularea programului la un punct de interes. Pentru a crea un punct de oprire (break point), putem fie să mutăm cursorul pe linia care conține codul și să apăsăm tasta F9, fie să facem click în partea din stânga numărului liniei care ne interesează. În oricare caz, vom avea o bulină roșie pe linia unde execuția programului se va opri:



Dacă rulăm acum programul apăsând tasta F5, programul se va opri la linia marcată cu roșu:



Linia galbenă va marca linia de cod care urmează să fi executată. Pentru a executa linia și a muta execuția la următoarea linie, apăsăm tasta `F11`.



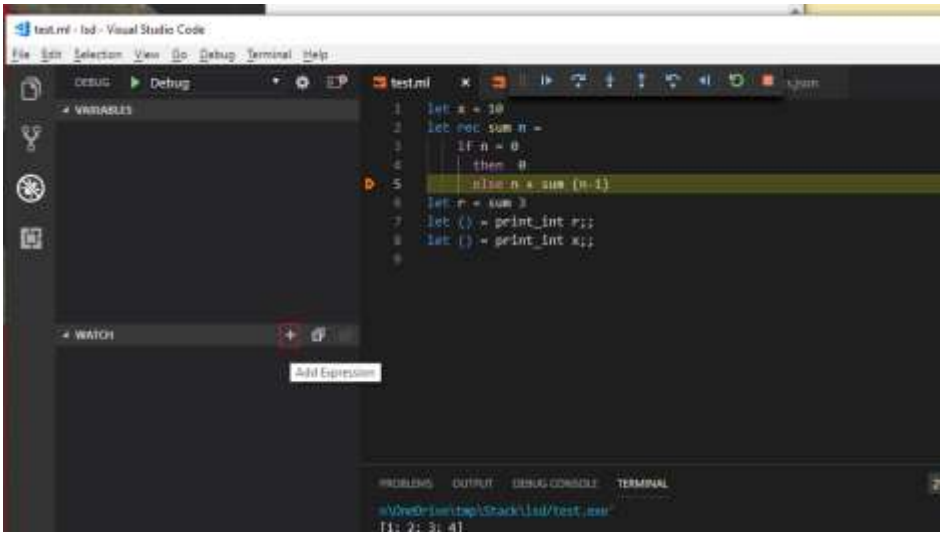
Dacă linia care urmează să fie executată conține un apel de funcție, putem să executăm linia în două moduri:

- Cu tasta `F11` – modul „Step into” - Acest mod va duce la intrarea în funcția care va fi executată
- Cu tasta `F10` – modul „Step over” – Acest mod va executa linia fără să intre pas cu pas prin funcțiile care sunt prezente în linia respectivă.

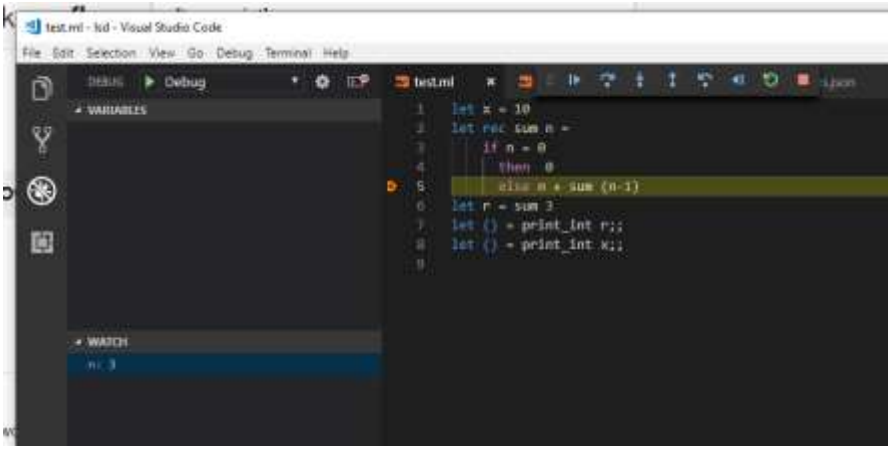
Inspectarea valorilor

Dacă programul este în modul de rulare pas cu pas, putem inspecta valoarea parametrilor și variabilelor folosind panoul de „Watch”.

Pentru a adăuga o variabilă în panoul de „Watch”, apăsăm pe butonul „+”:



După ce adăugăm o variabilă/parametru (de exemplu `n`), îi vom vedea valoarea curentă.



Recursivitate finală / tail recursive

Să încercăm să rulăm următorul program:

```
# let rec sum = function
    | 0 -> 0
    | n -> n + sum (n-1)
  let r =  sum 1000000;;
  ;;
Stack overflow during evaluation (looping recursion?).
```

Programul de mai sus încearcă să calculeze suma pentru primele 1.000.000 de numere. În loc să afișeze rezultatul, programul afișează o eroare: „`Stack overflow during evaluation (looping recursion?).`”

Motivul din spatele acestei erori este modul în care se face apelul recursiv. Fiecare apel duce la crearea unei intrări noi în stiva de apeluri care va stoca valorile parametrilor și a variabilelor locale. Spațiul alocat stivei de apeluri este limitat (depinde de sistemul de operare, pe Windows spațiul este de 2MB). Modul

în care funcționează funcția noastră recursivă, ar necesita 1.000.000 de intrări în stiva de apeluri, ceea ce duce la **depășirea spațiului alocat pentru stivă**, ceea ce tradus în engleză ne să eroarea „Stack overflow”.

Ar fi ideal dacă un apel recursiv nu ar rămâne pe stivă. Motivul pentru care toate apelurile sunt păstrate pe stivă este faptul că expresia care dă rezultatul funcției nu poate să fie calculată fără rezultatul apelului recursiv. De exemplu, expresia `n + sum (n-1)` nu poate să fie calculată până nu avem rezultatul lui `sum (n-1)`.

Dacă rezultatul apelului lui `sum n` ar fi același cu rezultatul apelului `sum (n - 1)`, apelul `sum n`, ar putea să fie scos de pe stivă, deoarece nu mai are nimic de calculat. Rezultatul apelului `sum (n-1)` ar fi returnat direct către cine a cerut rezultatul pentru `sum n`. Deoarece ele au aceeași valoare, programul ar funcționa corect.

Modificând codul funcției `sum` conform cu dezideratul de mai sus, obținem:

```
# let rec sum  = function
    | 0 -> 0
    | n -> sum (n - 1)
  let r = sum 1000000;;
  ;;
val sum : int -> int = <fun>
val r : int = 0
```

Codul de mai sus va rula, și va produce un rezultat. Este important să notăm că nu vom mai avea depășire de stivă. Apelurile recursive sunt realizate cu succes.

Datorită faptului că `sum n = sum (n-1)`, intrarea din stiva de apeluri pentru `sum n` poate să fie eliminată imediat ce începe execuția lui `sum (n-1)`, evitând depășirea stivei. „Operația a reușit! Pacientul, din păcate, a murit.” Funcția noastră rulează acum, dar furnizează rezultatul greșit.

Prin eliminarea expresiei care face suma, am pierdut exact partea utilă a funcției. Problema este identificare unui loc unde putem face calculul fără a modifica relația `sum n = sum (n-1)`.

O soluție ar fi să adăugăm un nou parametru la funcția `sum`. Acest parametru ar reține suma parțială a apelurilor anterioare. Când funcția este apelată cu `n = 0`, funcția va returna suma parțială. Suma parțială va porni de la o valoare neutră, în acest caz, valoarea `0`.

```
#let rec sum sum_partial = function
    | 0 -> sum_partial
    | n -> sum (sum_partial + n) (n - 1)
  let r = sum 0 (1_000_0000)
  ;;
val sum : int -> int = <fun>
val r : int = 50000005000000
```

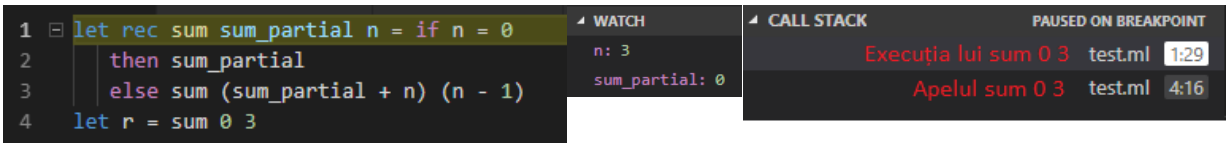
Rulare pas cu pas

Să rulăm funcția pas cu pas pentru a vedea cum diferă de varianta recursivă inițială.

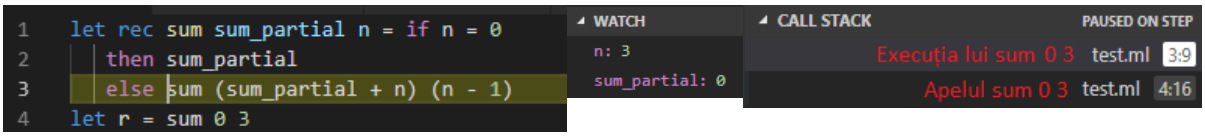
Pas 1: Execuția programului ajunge la apelul `sum 0 3`.

```
1  let rec sum sum_partial n = if n = 0
2      then sum_partial
3      else sum (sum_partial + n) (n - 1)
4  let r = sum 0 3
5  ;;
```

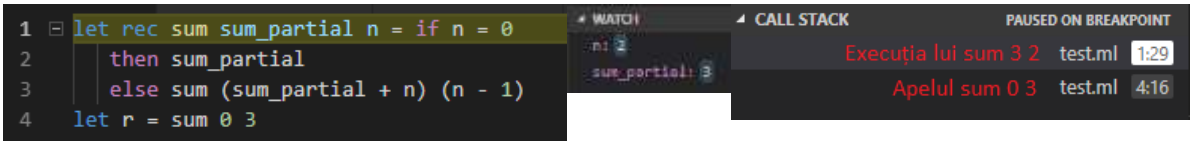
Pas 2: Pentru a calcula valoarea lui `sum 0 3`, programul va începe evaluarea expresiei de definiție pentru `sum_partial=0, n = 3`.



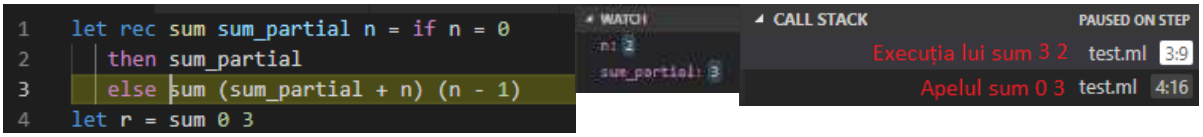
Pas 3: Programul va evalua condiția din expresia `if` și va decide să meargă pe ramura `else`.



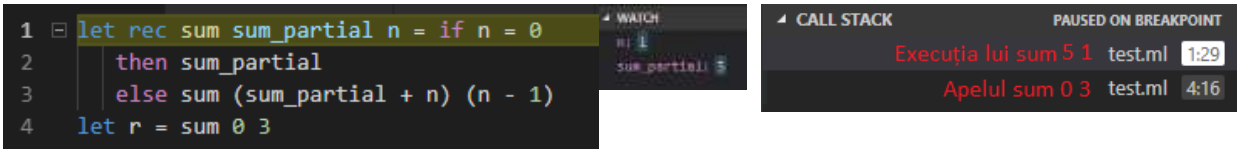
Pas 4: Evaluarea expresiei `sum (sum_partial + n) (n - 1) = sum (0 + 3) (3 – 1) = sum 3 2` necesită apelul recursiv al lui `sum` , după calculul celor două argumente. Deoarece rezultatul lui `sum 0 3 = sum 3 2` , intrarea pentru `0 3` poate fi scoasă de pe stivă.



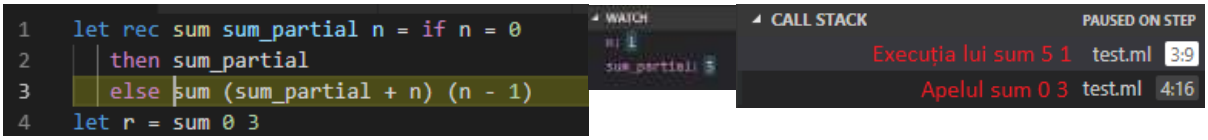
Pas 5: Programul va evalua condiția din expresia `if` și va decide să meargă pe ramura `else`.



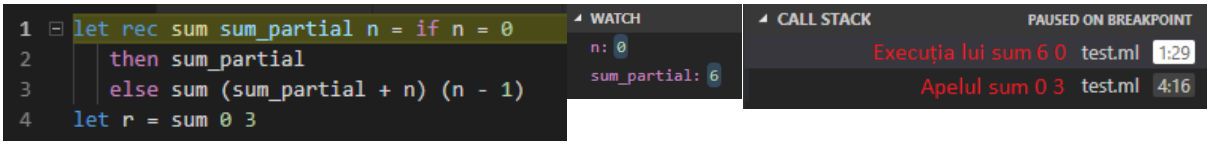
Pas 6: Evaluarea expresiei `sum (sum_partial + n) (n - 1) = sum (3 + 2) (2 - 1) = sum 5 1` necesită apelul recursiv al lui `sum` după calculul celor două argumente. Deoarece rezultatul lui `sum 3 2 = sum 5 1` , intrarea pentru `sum 3 2` poate fi scoasă de pe stivă.



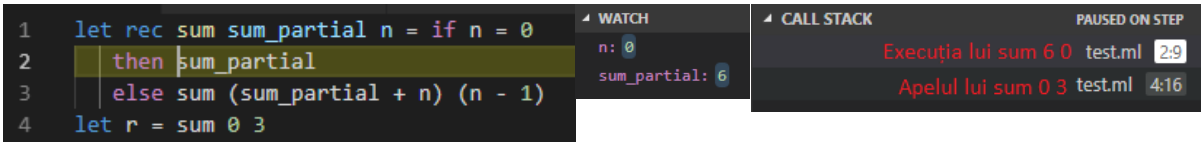
Pas 5: Programul va evalua condiția din expresia `if` și va decide să meargă pe ramura `else`.



Pas 6: Evaluarea expresiei `sum (sum_partial + n) (n - 1) = sum (5 + 1) (1 - 1) = sum 6 0` necesită apelul recursiv al lui `sum` după calculul celor două argumente. Deoarece rezultatul lui `sum 5 1 = sum 6 0`, intrarea pentru `sum 5 1` poate fi scoasă de pe stivă.



Pas 7: Programul va evalua condiția din expresia `if` și va decide să meargă pe ramura `then` și va returna `sum_partial.`



Pas 8: Rezultatul apelului `sum 6 0` este returnat în expresia care a cerut rezultatul pentru `sum 0 3` (după cum am văzut cele două apeluri au aceeași valoare)

Încapsularea detaliilor de implementare

Existența parametrului `sum_partial` nu ar trebui să fie cunoscută pentru un alt programator care folosește funcția `sum`. Parametrul `sum_partial` este un rezultat al modului în care am ales să implementăm funcția `sum` (folosind recursivitatea finală). Faptul acest parametru poate să fie pasat din exteriorul funcției pune câteva probleme:

1. Un utilizator al funcției `sum`, trebuie să știe că singura valoare corectă pentru `sum_partial` este 0. De unde ar putea să afle utilizatorul funcției această informație ? Fie dintr-un comentariu (pe care nu l-am scris) fie din inspectând codul funcției (ceea ce dă de lucru în plus unui programator, care va fi probabil leneș)
2. Parametrul `sum_partial` trebuie specificat de fiecare dată, cu toate că va avea tot timpul aceeași valoare. De ce să îl scrie de atâtea ori dacă nu poate să se modifice?
3. Un utilizator poate să paseze o valoare greșită pentru `sum_partial` și să determine o funcționare incorectă a funcției `sum`
4. Dacă modificăm implementarea funcției (să nu folosească recursivitatea finală) trebuie să modificăm toate locurile unde folosim funcția `sum`.

Putem rezolva toate aceste probleme dacă mutăm codul recursiv într-o funcție auxiliară în interiorul funcției `sum`, și apelăm funcția auxiliară în definiția lui `sum` cu valoarea corectă a lui `sum_partial`:

```
# let sum n =
    let rec aux sum_partial n = if n = 0
      then sum_partial
      else aux (sum_partial + n) (n - 1)
    in aux 0 n
  let r = sum 10000;;
val sum : int -> int = <fun>
val r : int = 50005000
```

Putem să simplificăm funcția de mai sus folosind aplicarea parțială pentru funcția aux, și funcții care potrivesc tipare. Funcția de mai jos este echivalentă cu cea de mai sus.

```
#let sum =
    let rec aux sum_partial = function
      | 0 -> sum_partial
      | n -> aux (sum_partial + n) (n - 1)
    in aux 0
  let r = sum 10000;;
val sum : int -> int = <fun>
val r : int = 50005000
```

Notă: O implementare corectă va ascunde detaliile de implementare a funcției.

Exerciții

Rezolvați folosind recursivitatea finală următoarele probleme

6. **Factorial**

Scrieți o funcție care să calculeze n! = 1 * 2 * 3 * ...* (n - 1) * n

7. **Aplicarea repetată a unei funcții**

în laboratorul trecut am scris funcții de ordin superior (funcționale) care aplicau o funcție de 2, 3, 4 ori. Definiți (recursiv) o funcție care ia ca parametru un întreg n și o funcție, și returnează funcția compusă cu ea însăși de n ori.

8. **Cifrele unui număr**

Un număr e reprezentat uzual în scris ca un șir de cifre în baza 10.

Un șir e o noțiune recursivă (un element, sau un șir urmat de un element).

Putem spune atunci că un număr n e fie o singură cifră, fie ultima cifră (n mod 10) precedată de alt număr (n / 10).

Folosind această definiție scrieți funcții recursive care calculează: suma cifrelor unui număr, numărul de cifre, produsul lor, cifra maximă / minimă, etc.

9. **Resturi modulo p**

În matematică știm că dacă p e un număr prim, și a nu se divide cu p, atunci șirul a, a2, a3, ... va ajunge la 1, luând numerele modulo p (adică resturile la împărțirea cu p).

De exemplu, fie p = 7 și a = 4. Atunci a2 = 16 ≡ 2 (mod 7), și a3 = a2 * a ≡ 2 * 4 ≡ 1 (mod 7).

(Se spune că mulțimea resturilor nenule modulo p prim formează un grup multiplicativ.)

Scrieți o funcție care ia ca parametru un număr întreg a și un număr p (presupus prim) și returnează cea mai mică putere n pentru care an ≡ 1 mod p (sau returnează 0 dacă a se divide cu p).

Indicație: scrieți o funcție auxiliară care mai are ca parametri și exponentul k respectiv valoarea ak (mod p), și care se apelează recursiv până când ak ≡ 1 (mod p).