Mihai Gontineac

Programare Funcțională

Haskell B. Curry (1900-1982)

O introducere utilizând limbajul Haskell

Alexandru Myller
Iaşi

EDITURA „ALEXANDRU MYLLER”
Iaşi, B-DUL CAROL I, nr.11,
tel. 0232-201061 / fax. 0232-201060
http://www.math.uaic.ro/~sm/

Descrierea CIP a Bibliotecii Naţionale a României
GONTINEAC, MIHAI
Programare funcţională : o introducere utilizând limbajul Haskell / Mihai Gontineac. - Iaşi : Editura Alexandru Myller, 2006
Bibliogr.
ISBN (10) 973-86987-6-6 ; ISBN (13) 978-973-86987-6-6

004.43

Referent ştiinţific
Lect. Univ. Dr. Dănuţ Rusu
Facultatea de Matematică
Universitatea „Alexandru Ioan Cuza” Iaşi

Memoriei tatălui meu

Cuprins

Introducere

În cele ce urmează, vom încerca să oferim motivația abordării unui model total diferit de ceea ce se înțelege, de obicei, prin programare.

O dată ce software-ul devine din ce în ce mai complex, este foarte util şi indicat să-l structurăm cât mai bine. Software-ul care este bine structurat este uşor de scris, uşor de corectat şi ne furnizează multe module care se pot reutiliza pentru reducerea costurilor de programare.

Numele de *programare funcțională* provine din faptul că un program este constituit în întregime din funcții. Însuşi programul principal este scris ca o funcție care primeşte intrarea ca argument şi furnizează ieşirea ca rezultat. De obicei, funcția principală este definită în termeni de alte funcții, care la rândul lor sunt definite în termeni de alte funcții, ş.a.m.d. până la funcțiile de bază, care constitue primitivele limbajului.

Programele în limbajele tradiționale, cum ar fi FORTRAN, C, Pascal, se bazează în mod esențial pe modificarea valorilor unei colecții de variabile, numită *stare*. Cu excepția cazurilor în care un program poate rula în mod continuu (de exemplu un "controler" al unui proces de producție), putem rezuma totul în urmatoarea abstractizare:

Înainte de execuție, sistemul se află în starea inițială, s_0, ce reprezintă intrarea programului (colecția datelor de intrare, de exemplu), şi, în momentul în care programul s-a terminat, starea are o valoare nouă *s'*, care include şi

rezultatele. Mai mult, execuția programului se face prin executarea unor comenzi, fiecare comanda schimbând starea. În acest fel, obținem un şir de tranziții prin diferite stări:

$$s = s_0 \to s_1 \to s_2 \to \ldots \to s_n = s'$$

Starea se modifică, în mod uzual, prin comenzi de *atribuire* (*asignare*), scrise de obicei sub forma `v=E` sau `v:=E`, unde `v` este o variabilă şi `E` este o expresie. Comenzile se execută în manieră secventială prin scrierea lor una dupa alta în cadrul programului, separate de obicei prin ";". Programul conține şi o serie de instrucțiuni asupra modului în care sunt făcute aceste schimbări de stare şi, din acest motiv, acest stil de programare se numeşte *procedural* sau *imperativ*. În mod corespunzător, limbajele care suportă acest tip de programare se numesc *limbaje procedurale* sau *imperative* (cum ar fi C, C++, Fortran, Pascal, ş.a.).

Programarea funcțională reprezintă o despărțire clară de acest model de programare. În mare, un program funcțional este o *expresie* iar execuția înseamnă *evaluarea* expresiei (din acest motiv, programarea funcționala se mai numeşte şi *programare aplicativă* deoarece mecanismul de bază este aplicarea unor funcții argumentelor lor).

Presupunând că un program imperativ (ca un întreg) este determinist, adică ieşirea sa este complet determinată de intrare, putem spune că starea finală este funcție de starea initială, adică $s'=f(s_0)$ sau, încă *Output* = *Program*(*Input*).

În programarea funcțională, această modalitate este pusă mai tare în evidență: un program nu este altceva decât o expresie ce corespunde unei funcții matematice *f*. Limbajele funcționale suportă construcția acestor expresii datorită unor componente funcționale extrem de puternice.

Programarea funcțională poate fi comparată cu cea imperativă şi în sens negativ şi în sens pozitiv.

- Astfel, în sens negativ, programele funcționale stricte nu utilizează variabilele - nu există stări. Ca şi consecință, ele nu pot utiliza atribuirea, întrucât nu există nimic căruia sa i se poată atribui ceva. Drept urmare, ideea execuției comenzilor în serie nu are sens, intrucât prima comandă poate să nu facă diferențiere de o a doua deoarece nu există stări care să facă legătura dintre ele (există totuşi şi o modalitate de manipulare secvențială a variabilelor, pe care o vom întâlni pe parcursul acestei cărți).
- În mod pozitiv, programele funcționale pot utiliza funcțiile într-un mod mult mai sofisticat. Funcțiile pot fi tratate în acelaşi mod în care tratăm orice alt obiect, oricât de simplu ar fi el; ca şi în cazul obiectelor intregi, ele pot fi date altor funcții ca argumente şi, în general, se poate calcula cu ele. În loc de inşiruire şi cicli, programele funcționale utilizează funcțiile recursive, adică funcțiile care sunt definite în funcție de ele însele. Prin contrast, majoritatea limbajelor tradiționale au facilitați sărace în acest sens. C - ul permite manipularea limitată a funcțiilor cu ajutorul pointerilor, însă nu permit crearea funcțiilor în mod dinamic, iar FORTRAN - ul nici măcar nu suportă recursia.

Pentru ilustrarea modului în care se poate face distincția între programarea funcțională şi cea imperativă, să dăm exemplul funcției factorial.

Codată imperativ în C, fără a utiliza atribuiri neuzuale, ea ar apărea astfel:

```
int fact(int n)
{ int x = 1;
```

```
while (n > 0)
{ x = x * n;
n = n - 1;
}
```

Aceeaşi funcție se poate da şi ca o funcție recursivă, de exemplu, utilizând Haskell:

```
factorial 1 = 1
factorial n = n * factorial (n - 1)
```

Vom reveni asupra unor astfel de exemple.

Probabil că principalul motiv pentru care utilizăm programele funcționale este că ele corespund mult mai direct obiectelor matematice, şi este mai uşor să le înțelegem. Un alt avantaj potențial ar fi faptul că evaluarea expresiilor se face fără efecte secundare în orice stare, adică subexpresii separate pot fi evaluate în orice ordine fără să se influenteze una pe cealalta. Aceasta ne conduce la concluzia că programele funcționale se pretează uşor la implementare paralelă. Un alt motiv este faptul că algoritmii în Haskell se scriu foarte asemănător cu descrierea lor în limbaj natural.

Lucrarea realizează o "mică" introducere în ceea ce se vrea a fi modelul de programare funcțional. După o scurtă introducere în lambda calcul, adică fundamentul teoretic al acestui model de programare, vom începe studiul posibilităților oferite de programarea funcțională cu ajutorul limbajului Haskell. Acesta este unul din limbajele cele mai reprezentative pentru programarea funcțională.

Din păcate, o serie de facilități şi tehnici foarte importante din Haskell nu şi-au putut găsi locul în această carte, întrucât, pentru înțelegerea completă a

lor, ar fi fost nevoie de o introducere teoretică ceva mai lungă şi, mai ales, pentru tinerii cititori, cartea ar fi putut deveni "plicticoasă". Vom încerca, cât de curând, să oferim cititorilor interesați o continuare a acestei introduceri.

Până acum s-ar putea spune că Haskell ar fi limbajul perfect. Întrebarea, naturală de altfel, este: de ce nu este utilizat peste tot? Principalul motiv ar fi că limbajele funcționale, în general, şi Haskell în particular, nu sunt încă atât de cunoscute. Însă, de curând (în acest an), a apărut Visual Haskell cu versiunile 0.0 şi 0.2 construit pe baza GHC 6.6. Putem spune că, odată cu apariția sa, începe o nouă eră pentru programarea funcțională şi asta datorită faptului că Visual Haskell este integrat în Visual Studio .NET 2003 şi Visual Studio .NET 2005, principalele medii de dezvoltare software în momentul de față.

Cartea are ca bază cursul opțional de "Introducere în Programarea Funcțională" ținut la anul III, secția Matematică-Informatică, Facultatea de Matematică din Universitatea "A. I. Cuza", Iaşi.

Capitolul I

Elemente ale lambda calculului

Cititorii care sunt obisnuiți cu programarea imperativă, vor vedea tranziția către programarea funcțională ca fiind, în mod inevitabil, destul de dificilă. Am ales să introducem mai întâi *lambda calculul*, aratând modul în care el poate fi văzut ca fundament teoretic al limbajelor funcționale (şi asta chiar dacă unii cititori vor fi nerăbdători să înceapă repede programarea "reală").

1 Importanța

Importanța λ-calculului în programarea funcțională şi în informatica generală este dată de câteva proprietăți caracteristice:

- Se pot modela mecanismele de chemare a funcțiilor de următoarele tipuri: *call-by-name, call-by-value* şi *call-by-need* (ultimele două mai sunt cunoscute şi ca *evaluare strictă* respectiv *evaluare întârziată*).
- λ-calculul este Turing universal, adică are aceeaşi putere ca a maşinii Turing universale; este, probabil, cel mai natural model de calcul. *Teza lui Church* afirmă că "funcțiile calculabile" sunt exact acele funcții care se pot reprezenta în λ-calcul .

- Noțiunile de *confluență*, *terminare* şi *formă normală* din acest model de calcul se aplică în teoria rescrierii (*rewriting theory*) .
- *Lisp*, unul din primele limbaje de programare importante, a fost inspirat de λ-calcul .
- λ-calculul (ca şi extensiile sale) se poate folosi în dezvoltarea unor sisteme mai bine scrise, prin utilizarea, de exemplu, a *polimorfismului*.
- datorită proprietăților sale, se poate utiliza în investigarea unor chestiuni teoretice din programare, cum ar fi *sinteza unui program* .
- *Semantica denotațională*, care este o metodă importantă pentru specificarea formală a limbajelor de programare, utilizează λ-calculul pentru notațiile sale.

2 Lambda Calcul

Introducere

Lambda calculul este bazat pe aşa numita *notație lambda* pentru notarea funcțiilor. În matematica informală, atunci când cineva doreşte să facă referire la o anumită funcție, îi dă mai intai un nume arbitrar dupa care utilizează acel nume. Marea majoritate a limbajelor de programare prezintă acelaşi aspect: definim funcțiile dându-le un nume.

În orice caz, acest tip de notație devine greoi atunci când apar funcții de ordin înalt. Dacă dorim să tratăm funcțiile ca pe orice alt obiect matematic, devine inconsistent să insistăm pe nume. În acest sens avem drept exemplu expresiile aritmetice, care sunt obținute, la rândul lor, din altele mai simple. Pur şi simplu scriem subexpresiile fără a fi necesar să le dăm un nume.

Imaginați-vă ce ar insemna să utilizăm expresiile aritmetice în modul următor:

"Definim x, y, z prin $x = 3$, $y = 5$ şi $z = 4$. Atunci $x^2 + y^2 = z^2$".

Notația lambda permite notarea funcțiilor în acelaşi mod ca orice altă categorie de obiecte matematice. Din acest punct de vedere, există deja o notație utilizată în acest scop, în matematică, chiar dacă ea este utilizată ca o parte a definiției unui nume temporar. Putem scrie:

$$x \mapsto f(x)$$

pentru a considera funcția care duce argumentul x într-o expresie arbitrară $f(x)$, care poate sau nu să conțină pe x. Pentru asta vom utiliza o altă notație, introdusă de A. Church (1941):

$$\lambda x.f[x]$$

care se citeşte în acelaşi mod. De exemplu, $\lambda x.x$ este funcția identitate, $\lambda x.x^n$ este ridicarea la putere, ş.a.m.d..

Din punct de vedere anecdotic, trebuie să subliniem faptul că simbolul λ nu are nici o semnificație şi alegere lui este întâmplătoare. Ea a apărut datorită dificultății scrierii $\hat{x}.f[x]$ (cum apăruse la început în manuscris). Întrucât dactilograful nu putea scrie la maşina de scris în acest mod, el a scris $\wedge x.f[x]$, care, în "mâinile" altui dactilograf, a devenit $\lambda x.f[x]$.☺

Utilizând notația lambda, se pot clarifica unele confuzii generate de notația matematică. De exemplu, se obisnuieşte să se scrie $f(x)$, lăsând contextului să decidă dacă este vorba de funcția însăşi, sau de rezultatul aplicării ei într-un x particular. Dacă pornim cu variabile şi constante, şi construim expresii utilizând λ-abstractizarea şi aplicarea funcțiilor argumentelor lor, putem reprezenta expresii matematice foarte complicate.

Vom utiliza notația convențională $f(x)$ pentru aplicarea funcției f în argumentul x; singura deosebire este că, în notație lambda, parantezele se pot omite, şi putem scrie $f\ x$. Se presupune că aplicarea funcției începe de la stânga, adică $f\ x\ y$ inseamna $(f(x))(y)$. Drept prescurtare pentru $\lambda x.\lambda y.f(x,y)$ vom scrie $\lambda x\ y.f\ x\ y$, şi aşa mai departe.

De asemenea, se presupune că abstractizarea cu lambda se extinde la dreapta cât de mult este posibil. În această manieră, $\lambda x.x\ y$ înseamnă $\lambda x.(x\ y)$ şi nu $(\lambda x.x)\ y$.

La o primă vedere, s-ar părea că avem nevoie de o notație speciala pentru funcții de mai multe variabile. Avem însa o metodă, numită *metoda Curry* (sau procedeul de *curry-zare*), după numele logicianului Haskell B. Curry. Ideea de bază poate fi privită şi ca provenind din teoria categoriilor, mai exact din *adjuncție*. Fără a intra însă, acum, în prea multe amănunte, putem transforma orice funcție de două variabile $A\times B\to C$, într-o funcție de o variabilă, $A\to(B\to C) = A\to C^B$.

De exemplu, să considerăm expresia de $\lambda xy.x+y$. Ea poate fi văzută, în virtutea comentariului precedent, ca o funcție $\boldsymbol{R}\to(\boldsymbol{R}\to\boldsymbol{R})$; se spune despre ea că este o *funcție de ordin superior*, sau o *funcțională*, întrucât odată aplicată unui argument ea produce o altă funcție, care acceptă următorul argument. În acest mod, argumentele se iau unul după altul şi nu amândouă odată:

$(\lambda xy.x+y)1\ 2=(\lambda y.1+y)2=1+2$

Definiția 1. *Termii* λ-calculului, numiți şi λ-*termi* (sau simplu *termi*), se construiesc recursiv dintr-o mulțime de *variabile* $x,y,z,....$ Ei pot lua una din formele următoare:

- *x variabilă*

- ($\lambda x.M$) *abstractizarea termilor*, unde M este term
- (MN) *aplicarea termilor*; în acest mod se notează rezultatul aplicarii funcției notate prin M funcției notate prin N.

Vom utiliza litere mari , $L,M,N,...$ pentru termi. Vom scrie $M \equiv N$ pentru a sublinia faptul că M şi N sunt λ-termi identici. Vom discuta mai târziu egalitatea termilor.

Această definiție a termilor ne arată că, în principal, proprietățile lor se pot demonstra prin inducție. În plus, se poate descrie sintaxa lambda termilor, în acelaşi mod în care se descrie sintaxa oricărui limbaj de programare:

$$Exp = Var \mid Const \mid Exp\ Exp \mid \lambda\ Var.Exp$$

2 Legarea variabilelor şi substituția.

În orice term ($\lambda x.M$) , x poartă numele de *variabilă legată* iar M este *corpul* termului. Orice apariție a lui x in M este *legată* prin abstractizare. O apariție a unei variabile se numeşte *liberă*, dacă nu este legată printr-o abstractizare care să o conțină. De exemplu, x apare legat şi y apare liber în ($\lambda z.(\lambda x.(xy))$).

Variabile libere şi variabile legate au existat din totdeauna în matematică. De exemplu:

- în $\int_a^b yf(x)dx$, variabila x este legată iar variabila y este liberă;
- în suma $\sum_{k=0}^{n} p(k)$, variabila k este legată;
- Cuantificatorii logici $\exists$ şi $\forall$ sunt de utilizați în matematică pentru legarea variabilelor.

Abstractizarea ($\lambda x.M$) reprezintă funcția f având proprietatea că $f(x)=M$ pentru toți x. Aplicarea lui f in N conduce la substituirea cu N a tuturor aparițiilor lui x în M. Astfel, $(\lambda x.x)E = E$.

Fie M un term.

Definiția 2. Mulțimea tuturor *variabilelor legate* din M, $BV(M)$, se defineşte prin:

- $BV(x) = \varnothing$;
- $BV(\lambda x.M) = BV(M) \cup \{x\}$;
- $BV(MN) = BV(M) \cup BV(N)$.

Definiția 3. Mulțimea tuturor *variabilelor libere* din M, $FV(M)$, se defineşte prin:

- $FV(x) = \{x\}$;
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$;
- $FV(MN) = FV(M) \cup FV(N)$.

Definiția 4. Dacă L,M sunt λ-termi, *rezultatul substituirii cu* L a tuturor aparițiilor libere ale lui y în M, este dat prin:

$$x[L/y] \equiv \begin{cases} L & \text{daca } y \equiv x \\ x & \text{in rest} \end{cases}$$

$$(\lambda x.M)[L/y] \equiv \begin{cases} \lambda x.M & \text{daca } y \equiv x \\ (\lambda x.M[L/y]) & \text{in rest} \end{cases}$$

$$(MN)[L/y] \equiv (M[L/y]\ N[L/y])$$

Notațiile definite anterior nu fac parte, ele însele, din λ-calcul. Ele aparțin meta-limbajului utilizat pentru a vorbi despre λ-calcul.

Atenție: atunci când se face o substituție nu trebuie "deranjată" legarea variabilei. Spre exemplificare, consideram term-ul $(\lambda x.(\lambda y.x))$. Acest term ar trebui să reprezinte o funcție care, atunci când ar fi aplicată unui argument N, conduce la funcția constantă $(\lambda y.N)$. Din păcate, acest lucru nu funcționează în cazul în care $N \equiv y$; noi am definit substituția prin $(\lambda y.x)[y/x] \equiv (\lambda y.y)$. Apariția liberă a lui x devine o apariție legată a lui y - un exemplu de *captură a variabilei*. Dacă s-ar permite așa ceva, λ-calculul nu ar fi consistent.

Substituția $M[N/x]$ este *sigură*, dacă mulțimea variabilelor legate ale lui M este disjunctă de cea a variabilelor libere ale lui N:

$BV(M) \cap FV(N) = \varnothing$

Pentru a putea face substituții sigure, este indicat să se opereze mai întâi o redenumire a variabilelor legate ale lui M (bineînțeles, numai dacă acest lucru este necesar), astfel încât să fie indeplinită condiția precedentă. Pentru exemplul prezentat, putem schimba mai întâi $(\lambda y.x)$ în $(\lambda z.x)$, după care se obține substituția corectă $(\lambda z.x)[y/x] = (\lambda z.y)$.

3 Conversii

Există trei tipuri de *conversii* utilizate în λ-calcul pentru a transforma un term într-un altul. Numele acestor conversii (primele două apărând în considerațiile lui A. Churh cu altă denumire) a fost dat de Haskell Curry.

- ***α-conversia*** $(\lambda x.M) \rightarrow_{\alpha} (\lambda y.M[y/x])$;

ea redenumeşte abstractizarea variabilei legate de la x la y. Se poate realiza numai în cazul în care y nu apare în M. În general se va ignora distincția între termi care pot fi făcuți identici printr-o α-conversie. Mai mult, este clar că această conversie nu este unidirectionala.

- ***β-conversia*** $((\lambda x.M)N) \to_\beta M[N/x]$;

ea substituie argumentul, *N*, în corpul abstractizării, *M*. Pentru ca această conversie să fie sigură trebuie ca $BV(M) \cap FV(N) = \varnothing$.

- ***η- conversia*** $(\lambda x.(Mx)) \to_\eta M$;

orice abstractizare $(\lambda x.(Mx))$ pentru care *x* nu apare ca variabilă liberă în *M* se poate reduce la *M*.

Dintre toate conversiile menționate, *β*-conversia este cea mai importantă din punct de vedere al programării. În timp ce *α*-conversia este, din punct de vedere tehnic, numai o schimbare a numelor variabilelor legate, *η*- conversia este o forma de extindere, de interes pentru partea logică a lambda calculului.

4 Reduceri

Până acum am vorbit despre termi identici şi nu am pomenit absolut nimic despre o "egalitate" a lor. Considerațiile acestei secțiuni vor fi utilizate în acest scop. În general, datorită faptului că *α*-conversia nu este unidirecțională, ea se ignoră.

Definiția 5. Spunem că term-ul *M se reduce simplu* la term-ul *N* (sau că *N* este *obținut* din *M printr-o reducere simplă*), dacă $M \to_\beta N$ sau $M \to_\eta N$. Vom nota acest lucru prin $M \to N$. Dacă un term nu admite nici o reducere, vom spune că el este în *formă normală.* Un term *admite o formă normală* dacă se poate reduce la un term aflat în formă normală printr-un număr finit de reduceri simple. A *normaliza* un term înseamnă a aplica reduceri simple pâna se obține un term aflat în formă normală.

Vom introduce, în mod formal, *regulile de inferență* pentru $\rightarrow$:

- Dacă $M \rightarrow M'$ atunci $(\lambda x.M) \rightarrow (\lambda x.M')$;
- Daca $M \rightarrow M'$ atunci $(MN) \rightarrow (M'N)$;
- Daca $M \rightarrow M'$ atunci $(LM) \rightarrow (LM')$.

Exemple.

1. Termii $\lambda yx.y$ si xyz sunt in forma normala.
2. $(\lambda x.x)y$ nu este în formă normală; forma sa normala este y.
3. $(\lambda x.xx)(\lambda x.xx)$ se reduce prin β-conversii la el însusi. Chiar dacă nu este afectat de reducere, el nu este în formă normală. Acest term se numeşte, de obicei, Ω. Vom mai avea prilejul să vorbim despre acest λ term.

5 "Curry"-zarea funcțiilor

Această tehnică a fost introdusă de Schönfinkel după numele lui Haskell B. Curry; ea a apărut datorită faptului că λ-calculul are numai funcții de un singur argument. O funcție având mai multe argumente se exprimă, cu ajutorul acestei tehnici, ca o funcție având un singur argument şi care ia valori tot funcții.

Să considerăm că L este un term, având două variabile libere x şi y, şi dorim să formalizăm funcția $f(x,y) = L$. Abstractizarea $(\lambda y.L)$ conține variabila liberă x; pentru fiecare x, se obține o funcție în variabila y. Aşadar abstractizarea $(\lambda x.(\lambda y.L))$ nu conține variabile libere; atunci când se aplică argumentelor M şi N, rezultatul se obține inlocuind x cu M şi y cu N în L. Din

punct de vedere simbolic, vom face β-conversii, şi vom ignora eventualele α-conversii:

$$(((\lambda x.(\lambda y.L))M)N) \rightarrow_\beta ((\lambda y.L[M/x])N) \rightarrow_\beta L[M/x][N/y]$$

Este clar că această tehnică se poate aplica pentru orice numar de argumente ale funcțiilor.

Funcțiile curry-zate se bucură de o mare popularitate în programarea funcțională întrucât se pot aplica câtorva dintre primele argumente obținând alte funcții care sunt, ele însele, de interes.

6 Convenții de utilizare a parantezelor

Scrierea cu prea multe paranteze este destul de greoaie şi este generatoare de erori. Din acest motiv, se fac o serie de simplificări sau abrevieri pentru a uşura scrierea. De exemplu, vom face următoarele abrevieri:

$(\lambda x_1.(\lambda x_2.(...(\lambda x_n.M)...)))$ se abreviază prin $(\lambda x_1 x_2 ... x_n.M)$

$(...(M_1 M_2)... M_n)$ se abreviază prin $(M_1 M_2 ... M_n)$

În fine, renunțăm la parantezele care se află la exterior şi la acelea care conțin corpul abstractizării. De exemplu:

$(\lambda x.(x(\lambda y.(yx))))$ se poate scrie $\lambda x.x(\lambda y.yx)$

Trebuie acordată foarte multă atenție modului de aşezare a parantezelor atunci când este nevoie. Am putea, de exemplu, să avem reducerea

$$\lambda z.(\lambda x.M)N \rightarrow_\beta \lambda z.M[N/x]$$

însă un termen asemanator, $\lambda z.z(\lambda x.M)N$ nu admite alte reduceri decât acelea care apar în interiorul termilor M şi N, deoarece $\lambda x.M$ nu se aplică nici unui term. Pe de altă parte, o reducere pe care am făcut-o mai sus (când am explicat curry-zarea) arată, fără paranteze, astfel:

$$(\lambda x.\lambda y.L)M)N) \rightarrow_\beta (\lambda y.L[M/x])N \rightarrow_\beta L[M/x][N/y]$$

Să mai observăm că, din punct de vedere notațional, $\lambda x.MN$ abreviază $\lambda x.(MN)$ şi nu $(\lambda x.M)N$. De asemenea, în mod uzual, xyz abreviază $(xy)z$ şi nu $x(yz)$.

3 Egalitate şi normalizare

λ-calculul este, în esență, o teorie ecuațională, adică este constituită din reguli care să demonstreze egalitatea a doi λ-termi. Una din proprietățile cheie în acest sens este că doi termi sunt egali dacă se pot reduce la un acelaşi term.

Reducerea în mai mulți paşi.

În general, când scriem $M \rightarrow N$, se intelege faptul că M se reduce la N după exact un *singur* pas de reducere. De obicei, vom fi interesați de posibilitatea reducerii dupa un numar oarecare de paş, şi vom scrie $M \rightarrow^* N$ dacă există $k \geq 0$ astfel încât să avem:

$$M \rightarrow M_1 \rightarrow M_2 \rightarrow ... \rightarrow M_k \equiv N$$

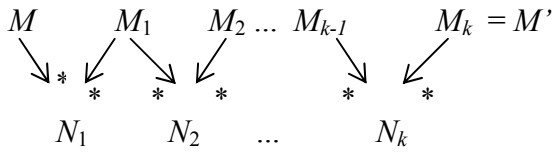
Notația nu este întâmplătoare, fiind clar că relația "$\rightarrow^*$" este închiderea reflexivă şi tranzitivă a relației "$\rightarrow$".

Exemplu. $((\lambda z.(zy))(\lambda x.x)) \rightarrow^* y$.

Egalitatea λ-termilor

Prin *extindere* vom înțelege relația inversă reducerii,

Din punct de vedere informal, doi termi M,M' sunt *egali* dacă unul se poate obține din celălalt printr-un număr finit de *transformări* (adică reduceri sau extinderi). Putem exprima acest lucru, figurativ, în modul următor:



Exemplu. Avem $a((\lambda y.by)c)=(\lambda x.ax)(bc)$ intrucât ambii termi se reduc la $a(bc)$.

Observații.

- Relația de egalitate este relația de echivalență generată de $\rightarrow$, adică $(\rightarrow \cup \rightarrow^{-1})^*$. Din punct de vedere intuitiv, faptul că $M = M'$ înseamnă că M şi M' au aceleaşi valori.
- În plus, relația de egalitate satisface legile de a fi o congruență pentru fiecare dintre modurile de construcție a λ-termilor. Astfel, dacă $M = M'$ atunci au loc
  - $(\lambda x.M)=(\lambda x.M')$;
  - $(MN)=(M'N)$;
  - $(LM)=(LM')$.

Teorema Church-Rosser

Din punct de vedere intuitiv, această teoremă demonstrează faptul că λ-calculul este *confluent*, adică, pornind de la un acelaşi λ-term, nu există două şiruri de transformări care să ne conducă la forme normale distincte. Cu alte cuvinte, forma normală a unui term este independentă de ordinea în care sunt efectuate reducerile.

Teorema 1. Daca $M = N$ atunci există L astfel încât $M \rightarrow^* L$ şi $N \rightarrow^* L$.

Exemplu. Să considerăm urmatorul term $(\lambda x.ax)((\lambda y.by)c)$. Putem să-i aplicăm reducerile pentru a ajunge la forma normală în două moduri. Subtermii asupra cărora vom aplica transformări apar îngroşaţi:

$$\mathbf{(\lambda x.ax)((\lambda y.by)c)} \rightarrow a(\mathbf{(\lambda y.by)c}) \rightarrow a(bc)$$

$$(\lambda x.ax)(\mathbf{(\lambda y.by)c}) \rightarrow \mathbf{(\lambda x.ax)(bc)} \rightarrow a(bc)$$

Consecințe.

- Daca $M = N$ şi N este în formă normală, atunci $M \rightarrow^* N$; adică, dacă un term se poate transforma în formă normală utilizând reduceri şi expansiuni, atunci forma normală se atinge numai prin reduceri.
- Daca $M = N$ şi ambii termi M şi N sunt în formă normală, atunci $M \equiv N$ (până la o redenumire a variabilelor legate). Reciproc, dacă M şi N sunt în formă normală şi sunt distincti atunci $M \neq N$, adică nu există nici o modalitate să transformăm M în N. De exemplu, $\lambda xy.x \neq \lambda xy.y$.

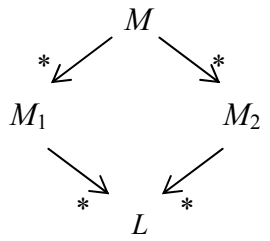
Observație. O teorie ecuaționala este *inconsistentă* dacă toate ecuațiile sale se pot demonstra. Datorită teoremei Church-Rosser, putem spune că λ-calculul este consistent, intrucât nu există nici o posibilitate de obținere a unor forme normale distincte prin aplicarea unor strategii distincte de transformare. Fără această proprietate, λ-calculul nu ar fi avut nici o relevanță în calcul.

Proprietatea "Diamant"

Unul dintre paşii cei mai importanți în demonstrarea teoremei Church-Rosser este demonstrarea *proprietății diamant*:

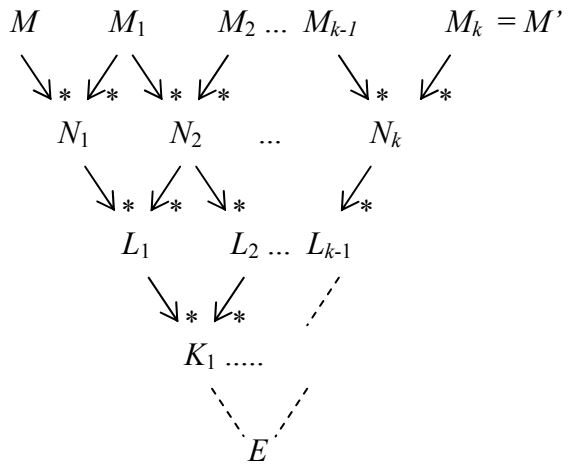
Teorema 2. Dacă $M \to^* M_1$ şi $M \to^* M_2$ atunci există un term L astfel încât să avem $M_1 \to L$ şi $M_2 \to L$.

Putem "vizualiza" teorema prin diagrama:



Această proprietate se poate demonstra utilizând o "fragmentare", considerând mai întâi cazul în care $M \to M_1$ (într-un singur pas) şi $M \to^* M_2$ (în mai mulți paşi). Aceste fragmente se lipesc apoi până la completarea "diamantului". Detaliile, în schimb, implică o analiză laborioasă pe cazuri posibile de reducere datorate diverselor tipuri de reducere pentru diferitele tipuri de termi. Pentru cei interesați, recomandăm parcurgerea bibliografiei de la sfârşitul acestui capitol.

În orice caz, urmatoarea diagramă ilustrează principiul de "funcționare" a proprietății diamant:



Posibilitatea neterminării

Chiar dacă, aplicând două şiruri distincte de transformări, nu se pot obţine forme normale diferite ale unui acelaşi term, ele pot furniza comportamente diferite: unul poate să se oprească, în timp ce celălalt va lucra la nesfârşit. În mod uzual, dacă M are o formă normală şi admite un şir de reduceri infinit, el va conţine un subterm L care nu are o formă normală; acest subterm poate fi şters printr-o reducere.

Sa ne amintim că termul $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ se reduce la el insuşi. Reducerea $(\lambda y.a)\ \Omega \rightarrow a$ îşi atinge forma normală prin ştergerea lui Ω.

Aceasta corespunde aşa numitei metode *call-by-name* de utilizare a funcţiilor: argumentul funcţiei nu se reduce ci se substitue aşa cum este el în corpul abstractizării.

Încercând să normalizăm mai întâi argumentul, în termul anterior, se generează un şir de reduceri care nu se poate termina:

$$(\lambda y.a)\ \Omega \rightarrow (\lambda y.a)\ \Omega \rightarrow (\lambda y.a)\ \Omega \rightarrow \ldots$$

Evaluarea argumentului înaintea substituirii în corpul abstractizării corespunde aşa-numitei metode *call-by-value* de utilizare a unei funcţii. În exemplul prezentat, această strategie nu conduce niciodată la o formă normală.

Ordinea normală de reducere

Metoda de reducere în *ordine normală* este ca, la fiecare pas, să se efectueze β-reducerea cea mai din stânga şi cea mai "exterioară" (η-reducerile pot fi lăsate la sfârşit). *Cea mai din stânga* înseamnă, să reducem L înaintea lui N în termul LN. *Cea mai exterioară* înseamnă, de fapt, să se reducă $(\lambda y.M)N$ înainte de a reduce M sau N.

Ordinea normala de reducere corespunde evaluarii "call-by-name".

Se poate demonstra că, dacă există formă normală, atunci ea poate fi atinsă prin ordinea normală de reducere. În orice caz, putem observa că, reducând termul L mai intâi, în termul LN, se poate transforma L într-o abstractizare, de exemplu $\lambda x.M$. Reducerea lui $(\lambda x.M)N$, poate şterge N.

Evaluare întârziată (lazy evaluation)

Din punct de vedere teoretic, ordinea normală de reducere este optimală, în sensul că oferă forma normală ori de câte ori ea există. Pentru calculul practic, însă, ea nu este prea eficientă.

Să presupunem că avem deja o "codificare" în lambda calcul numerelor naturale (vom vedea în următoarea secţiune cum codificăm diferite tipuri de date în λ-calcul) şi să definim funcţia de "ridicare la pătrat" prin

$$\mathbf{sqr} \equiv \lambda n.\mathbf{mult}\ nn.$$

Atunci

$$\mathbf{sqr}(\mathbf{sqr}\ N) \rightarrow \mathbf{mult}(\mathbf{sqr}N)(\mathbf{sqr}N) \rightarrow \mathbf{mult}(\mathbf{mult}\ NN)(\mathbf{mult}\ NN)$$

şi va trebui să se evalueze **patru** copii ai termului *N*.

Evaluarea call-by-value ar fi evaluat *N* înainte de orice, o singură dată, dar, după cum s-a văzut deja, există posibilitatea neterminării.

Lazy evaluation (evaluarea întârziată), mai este numită şi evaluare *call-by-need*. Ea nu evaluează un argument de mai multe ori. Un argument se evaluează doar atunci când valoarea sa este cerută sa producă un răspuns; chiar şi atunci, argumentul se evaluează doar în punctul în care este nevoie (şi, prin urmare, putem lucra şi cu liste infinite!). Evaluarea întârziată se poate implementa prin reprezentarea unui term printr-un graf (şi nu printr-un arbore). Orice nod asociat mai multor arce reprezintă un subterm de a cărui valoare este nevoie de mai multe ori. Ori de câte acel subterm este redus, rezultatul va rescrie nodul, iar toate referirile la acel subtem vor avea acces imediat la această înlocuire.

Exerciții

1. Efectuați substituțiile $(\lambda y.x(\lambda x.x))[(\lambda y.yx)/x]$ şi $(y(\lambda z.xz))[(\lambda y.zy)/x]$.

2. Reduceți în două moduri $(\lambda x.xx)(\mathbf{I}a)$, unde $\mathbf{I} \equiv (\lambda x.x)$.

3. Ce se întâmplă cu reducerea lui $(\lambda xy.L)MN$ dacă y este liberă în M

4. Reduceți în două moduri $(\lambda x.(\lambda y.xy)z)y$.

5. Reduceți $((\lambda x.\lambda y.\mathbf{add}\ xy)3)4$.

6. Arătați egalitatea $(\lambda fgx.fx(gx))(\lambda xy.x)(\lambda xy.x) = \lambda x.x$

4 Codificarea datelor in λ-calcul

Lambda calculul este suficient de expresiv pentru a permite codificarea valorilor booleene, a perechilor ordonate, a numerelor naturale şi a listelor - adică toate structurile de date necesare într-un program funcțional. Aceste codificări permit (în mod virtual) să modelăm întreaga programare funcțională între limitele oferite de lambda calcul.

Codificările ar putea, unele dintre ele, să pară nenaturale, şi nu sunt eficiente din punct de vedere computațional. Prin aceasta ele se aseamănă cu codificările şi programele maşinii Turing. Dar, spre deosebire de acestea, codificările din lambda calcul sunt interesante şi din punct de vedere matematic, ele apărând iar şi iar în studiile teoretice. Multe din ele conțin ideea că "orice structură de date poate avea şi structura de control în ea".

Valorile booleene

Orice codificare a valorilor booleene trebuie să definească termii **true**, **false**, **if**, şi să satisfacă (pentru orice M şi N):

$$\textbf{if true } MN = M$$

$$\textbf{if false } MN = N.$$

De obicei se utilizează urmatoarea codificare:

$$\textbf{true} \equiv \lambda xy.x$$

$$\textbf{false} \equiv \lambda xy.y$$

$$\textbf{if} \equiv \lambda pxy.pxy$$

Conform teoremei Church-Rosser, este clar că **true** ≠ **false**, deoarece ele sunt forme normale distincte. Mai mult, **if** nu este nici măcar necesar să fie

definit, deoarece valorile de adevăr au conținute în ele însele operatorul condițional:

$$\textbf{true}\ MN \equiv (\lambda xy.x)MN \rightarrow^* M$$

$$\textbf{false}\ MN \equiv (\lambda xy.y)MN \rightarrow^* N$$

Toate operațiile uzuale pe valorile de adevăr se pot defini cu operator condițional. Codificările pentru *conjuncție*, *disjuncție* şi *negare* sunt:

$$\textbf{and} \equiv \lambda pq.\textbf{if}\ p\ q\ \textbf{false}$$

$$\textbf{or} \equiv \lambda pq.\textbf{if}\ p\ \textbf{true}\ q$$

$$\textbf{not} \equiv \lambda p.\textbf{if}\ p\ \textbf{false}\ \textbf{true}$$

Perechi ordonate

După ce am definit **true** şi **false**, definim următoarele funcții: **pair** care este o funcție de construcție a perechilor şi proiecțiile **fst** şi **snd**. Ele sunt codate în lambda calcul astfel:

$$\textbf{pair} \equiv \lambda xyf.fxy$$

$$\textbf{fst} \equiv \lambda p.p\ \textbf{true}$$

$$\textbf{snd} \equiv \lambda p.p\ \textbf{false}$$

Este clar ca $\textbf{pair}MN \equiv (\lambda xyf.xyf)MN \rightarrow \lambda f.fMN$, "împachetând" pe *M* şi pe *N* împreună. Unei perechi i se poate aplica orice functie de 2 argumente, de forma $\lambda xy.L$, obținându-se $L[M/x][N/y]$.

Proiecțiile lucrează şi ele după acelaşi principiu; se poate verifica uşor, şi se propune ca exercițiu, că $\textbf{fst}\ (\textbf{pair}\ MN) \rightarrow^* M$ şi $\textbf{snd}\ (\textbf{pair}\ MN) \rightarrow^* N$.

Numerele naturale

Alonso Church a dezvoltat o codificare elegantă a numerelor naturale în lambda calcul, şi, chiar dacă după aceasta au mai apărut şi alte codificari, codificarea sa se utilizează şi în lambda calculul de ordin doi.

Definim

$$\mathbf{0} \equiv \lambda fx.x$$

$$\mathbf{1} \equiv \lambda fx.fx$$

$$\mathbf{2} \equiv \lambda fx.f(fx)$$

.....

$$\mathbf{n} \equiv \lambda fx.f(...(fx)...)$$

.....

Vom numi aceşti termi *numeralele lui Church*; aşadar, numeralul **n** este funcția care duce f în f^n.

Codificarea operațiilor pe numerale

Pentru adunare, înmulțire şi ridicare la putere, vom folosi următoarele codificări:

$$\mathbf{add} \equiv \lambda mnfx.mf(nfx)$$

$$\mathbf{mult} \equiv \lambda mnfx.m(nf)x$$

$$\mathbf{expt} \equiv \lambda mnfx.nmfx$$

Aceste operații nu sunt suficiente însă pentru definirea tuturor funcțiilor calculabile pe numere naturale. Vom defini, în continuare, şi alți termi pentru *funcția succesor* şi *testarea cu zero*.

$\mathbf{suc} \equiv \lambda nfx.f(nfx)$

iszero ≡ λ*n*.*n*(λ*x*.**false**) **true**

Se pot verifica uşor următoarele reduceri, pentru orice numeral Church **n**:

sucn →* **n**+**1**

iszero 0 →* **true**

iszero (**n**+**1**) →* **false**

Funcția predecesor şi *scăderea* sunt codificate prin

prefn ≡ λ*fp*.**pair** (*f*(**fst** *p*))(**fst** *p*)

pre ≡ λ*nfx*.**snd** (*n*(**prefn** *f*)(**pair** *xx*))

sub ≡ λ*mn*.*n***pre***m*

Modalitatea de codificare a funcției predecesor **pre** pare greoaie dar este datorată faptului că trebuie sa reducem un $n+1$ iterator la un n iterator. Adică, date f şi x, trebuie să determinăm g şi y astfel încât $g^{n+1}y$ calculează $f^n x$. O astfel de funcție este funcția care satisface $g(x, z) = (f(x),x)$. Se observă imediat că $g^{n+1}(x, x) = (f^{n+1}(x), f^n(x))$.

Termul **pref** *f* construieşte, în fond, funcția *g*. Pentru scădere, **sub m n** calculeaza al *n*-lea predecesor al lui **m**.

Exerciții.

1. Verificați că **add m n** →* **m**+**n** si ca **mult m n** →* **m**×**n**.
2. Verificați că **pre** (**n**+**1**) →* **n** şi **pre** (**0**) →* **0**.

Liste

Numeralele Church se pot generaliza pentru a putea reprezenta liste. O lista de forma $[x_1, x_2, ..., x_n]$ poate fi reprezentată prin funcția care "duce" f şi y în $fx_1(fx_2 ...(fx_n y) ...)$. Astfel construite, listele îşi conțin şi structura de control.

Ca alternativa, listele se pot reprezenta cu ajutorul împerecherii. Această codificare este mai uşor de inteles întrucat este mult mai apropiată de implementarile reale.

Lista $[x_1, x_2, ..., x_n]$ se va reprezenta prin $x_1 :: x_2 :: ... :: \mathbf{nil}$. Pentru a păstra exprimările cât mai simple, vom utiliza două nivele de împerechere. Fiecare "celula **cons**" $x :: y$ se poate reprezenta prin (**false**(x, y)), unde **false** este privit ca un etichetă distinsă. **nil** ar trebui reprezentat printr-o pereche a cărei prima componentă este **true**, cum ar fi (**true**,**true**), dar se poate utiliza şi o definiție mai simplă. Codificările sunt:

$$\mathbf{nil} \equiv \lambda z.z$$
$$\mathbf{cons} \equiv \lambda xy.\mathbf{pair\ false}\ (\mathbf{pair}\ xy)$$
$$\mathbf{null} \equiv \mathbf{fst}$$
$$\mathbf{head} \equiv \lambda z.\ \mathbf{fst}\ (\mathbf{snd}\ z)$$
$$\mathbf{tail} \equiv \lambda z.\mathbf{snd}(\mathbf{snd}\ z)$$

Exerciții. Verificați următoarele proprietăți:

1. **null nil** $\rightarrow$* **true**.
2. **null**(**cons** MN) $\rightarrow$* **false**.
3. **head** (**cons** MN) $\rightarrow$* M .
4. **tail** (**cons** MN) $\rightarrow$* N

Să mai remarcăm că, ultimele două proprietăți, propuse ca exercițiu, au loc pentru orice termi M şi N chiar şi în cazul în care ei nu au forme normale. Din acest motiv, **pair** şi **cons** sunt *constructori* "*leneşi*" sau "*întârziați*" adică

ei nu îşi "evaluează argumentele". Odată cu introducerea definițiilor recursive, vom fi astfel capabili să lucrăm cu liste infinite.

5 Scrierea funcțiilor recursive în lambda calcul

Recursia este esențială în programarea funcționala. Cu ajutorul numeralilor lui Church este posibil să definim "aproape-toate" funcțiile calculabile pe numere naturale. Numeralii lui Church au o sursă internă de repetiție. De aici, putem deriva *recursia primitivă* care, atunci când se aplică utilizând funcții de ordin înalt, definesc o clasă mai largă decât cea a funcțiilor recursive studiate in Teoria Calculabilității. Chiar dacă *funcţia lui Ackermann* nu este primitiv recursivă în sens uzual, ea se poate codifica utilizând numeralii lui Church. Definind

$$\mathbf{ack} \equiv \lambda m.m(\lambda fn.nf(f\,\mathbf{1}))\ \mathbf{suc},$$

putem obține ecuațiile recursive satisfăcute de funcția lui Ackermann, adică

$$\mathbf{ack\ 0\ n} = \mathbf{n+1}$$

$$\mathbf{ack\ (m+1)\ 0} = \mathbf{ack\ m\ 1}$$

$$\mathbf{ack\ (m+1)\ (n+1)} = \mathbf{ack\ m\ (ack(m+1)\ n)}$$

Să le verificăm:

$$\mathbf{ack\ 0\ n} \to^* \mathbf{0}(\lambda fn.nf(f\mathbf{1}))\mathbf{suc\ n} \to^* \mathbf{suc\ n} \to^* \mathbf{n+1}$$

Pentru celelalte două ecuatii, să observăm mai întâi că:

$$\mathbf{ack\ (m+1)\ n} \to^* \mathbf{(m+1)}(\lambda fn.nf(f\mathbf{1}))\mathbf{suc\ n} \to^* (\lambda fn.nf(f\mathbf{1}))(\mathbf{m}(\lambda fn.nf(f\mathbf{1}))\mathbf{suc})\mathbf{n}$$
$$= (\lambda fn.nf(f\mathbf{1}))(\mathbf{ack\ m\ })\mathbf{n} \to^* \mathbf{n\ (ack\ m)\ (ack\ m\ 1)}$$

Particularizând, obținem:

$\mathbf{ack\ (m+1)\ 0} \to^* \mathbf{0(ack\ m)(ack\ m\ 1)} \to^* \mathbf{ack\ m\ 1}$, şi

ack (m+1)(n+1) →* n+1 (ack m)(ack m 1) →* ack m (n (ack m)(ack m 1))
= ack m(ack (m+1) n)

Utilizarea punctelor fixe la functii recursive

Codificarea facută funcției lui Ackermann, chiar dacă funcționează, pare artificială şi este nevoie de perspicacitate pentru a o obține. Probleme apar însă, de exemplu, în cazul unei funcții ale cărei "chemări" recursive implică ceva mai mult decât o simplă scădere cu 1 a argumentului: astfel, împărțirea se face prin scăderi succesive.

În lambda calcul se pot defini toate funcțiile recursive, chiar şi acelea a căror definire poate fi interminabilă pentru unu sau mai multe argumente.

Ideea de codificare a recursiei este uniformă şi este independentă de definiția recursivă şi reprezentarea structurilor de date (spre deosebire de modalitatea prezentată pentru funcția lui Ackermann, care utilizează numeralii lui Church). Ea constă în utilizarea aşa numiților *combinatori de punct fix*, adică un λ- term **W** având proprietatea că **W***F* = *F*(**W***F*) pentru orice term *F*.

Să detaliem puțin terminologia utilizată. Un *punct fix* al unei funcții *F* este orice *X* care satisface *FX* = *X*; în cazul nostru, *X* ≡ **W***F*. Un *combinator* este un λ- term care nu conține variabile libere. Pentru codificarea recursiei, *F* reprezintă corpul definiției recursive; legea **W***F* = *F*(**W***F*) permite ca *F* să fie "desfacută" ori de câte ori este nevoie.

Utilizarea lui W

Vom codifica funcția factorial şi lista infinită [0,0,0,...]. Aceste funcții trebuie să satisfacă ecuațiile recursive:

fact N = **if** (**iszero** N) **1** (**mult** N (**fact** (**pre** N)))

zeroes = **cons 0 zeroes**

Pentru aceasta, definim

fact ≡ **W**(λgn. **if** (**iszero** n) **1** (**mult** n(g(**pre** n))))

zeroes ≡ **W**(λg.**cons 0** g)

În fiecare dintre definiții, chemarea recursivă este inlocuită prin variabilă g în **W**(λg....). Să verificăm recursia pentru **zeroes**:

zeroes ≡ **W** (λg. **cons 0** g) = (λg. **cons 0** g)(**W** (λg. **cons 0** g)) =

(λg. **cons 0** g) **zeroes** → **cons 0 zeroes**

În general, ecuația recursivă $M = PM$, unde P este un term oarecare, este satisfacută dacă definim $M \equiv \mathbf{W}P$. Să considerăm un caz particular, cazul în care M este o funcție de n variabile. Ecuația $M\ x_1\ x_2\ ...\ x_n = PM$ este verificată dacă definim

$$M \equiv \mathbf{W}\ (\lambda g x_1\ x_2\ ...\ x_n.\ Pg)$$

întrucât

$$Mx_1\ x_2\ ...\ x_n \equiv \mathbf{W}\ (\lambda g x_1\ x_2\ ...\ x_n.\ Pg)\ x_1\ x_2\ ...\ x_n$$
$$= (\lambda g x_1\ x_2\ ...\ x_n.\ Pg)\ Mx_1\ x_2\ ...\ x_n$$
$$\rightarrow PM$$

Să considerăm acum cazul definiției recursive "mutuale", pentru doi termi M şi N, având corpurile P, respectiv Q, adică

$$M = PMN$$
$$N = QMN$$

În acest caz, ideea de bază este de a considera punctul fix pentru o funcție F pe perechi, astfel incăt $F(X,Y) = (PXY, QXY)$. Utilizând codificarea dată perechilor, definim

$L \equiv \mathbf{W}\ (\lambda z.\mathbf{pair}\ (P(\mathbf{fst}\ z)(\mathbf{snd}\ z))\ (Q(\mathbf{fst}\ z)(\mathbf{snd}\ z)))$

$M \equiv \mathbf{fst}\ L$

$N \equiv \mathbf{snd}\ L$

Din proprietatea de punct fix,

$L \equiv \mathbf{pair}\ (P(\mathbf{fst}\ L)(\mathbf{snd}\ L))\ (Q(\mathbf{fst}\ L)(\mathbf{snd}\ L))$

şi, aplicând proiecțiile, obținem egalitățile dorite,

$M = P(\mathbf{fst}\ L)(\mathbf{snd}\ L) = PMN$

$N = Q(\mathbf{fst}\ L)(\mathbf{snd}\ L) = QMN$

Exemple de combinatori de punct fix

Combinatorul **W** a fost descoperit de Haskell B.Curry. El este definit prin

$\mathbf{W} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Să verificăm faptul că el satisface proprietatea de punct fix:

$\mathbf{W}\ F \rightarrow (\lambda x.F(xx))(\lambda x.F(xx)) \rightarrow F((\lambda x.F(xx))(\lambda x.F(xx))) = F(\mathbf{W}\ F)$

Verificarea s-a efectuat prin utilizarea a două β-reduceri urmate de o β- expansiune. Nu există nici o reducere de genul $\mathbf{W}\ F \rightarrow^* F(\mathbf{W}\ F)$.

Există şi alți combinatori:

- Combinatorul Θ al lui A. Turing. El este definit prin $\Theta \equiv AA$, unde

$A \equiv \lambda xy.y(xxy)$

- Combinatorul $ al lui Klop. El se defineste ca fiind

$\$ \equiv \mathcal{L}$, iar

$\mathcal{L} \equiv \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$

Ajunşi aici, încheiem scurta introducere în lambda calcul şi în modul prin care orice structura de date se poate codifica în lambda calcul. Cititorul interesat poate consulta bibliografia ([1],[6],[14]).

Merită totuşi, în final, să facem o scurtă comparație între lambda calcul şi maşinile Turing.

λ-calculul poate codifica toate structurile comune de date, cum ar fi valorile booleene şi listele, astfel încât să fie satisfacute proprietățile lor naturale. El poate exprima, de asemenea, definițiile recursive. Întrucât aceste codificări sunt tehnice, ar putea părea ca nu merită studiate, însa nu este cazul:

- Codificarea via numeralii lui Church se utilizează în calcul mult mai avansat, cum ar fi λ-calculul de ordin doi.
- Codificarea listelor via perechi ordonate modelează implementarea lor uzuală pe computer
- Înțelegerea definițiilor recursive ca puncte fixe este metoda uzuală din teoria semantică.

Aceste construcții şi concepte se întâlnesc în toată informatica teoretică, ceea ce nu se poate spune însă despre programele maşinilor Turing.

Capitolul II

Limbajul Haskell 98 – descriere generală

1. Istoric

Limbajul Haskell nu se poate lăuda cu o "vârstă înaintată". Din acest punct de vedere, limbajul Lisp este cu mult mai vechi, şi se bucura de mult mai multă atenție în programele analitice a multor universități.

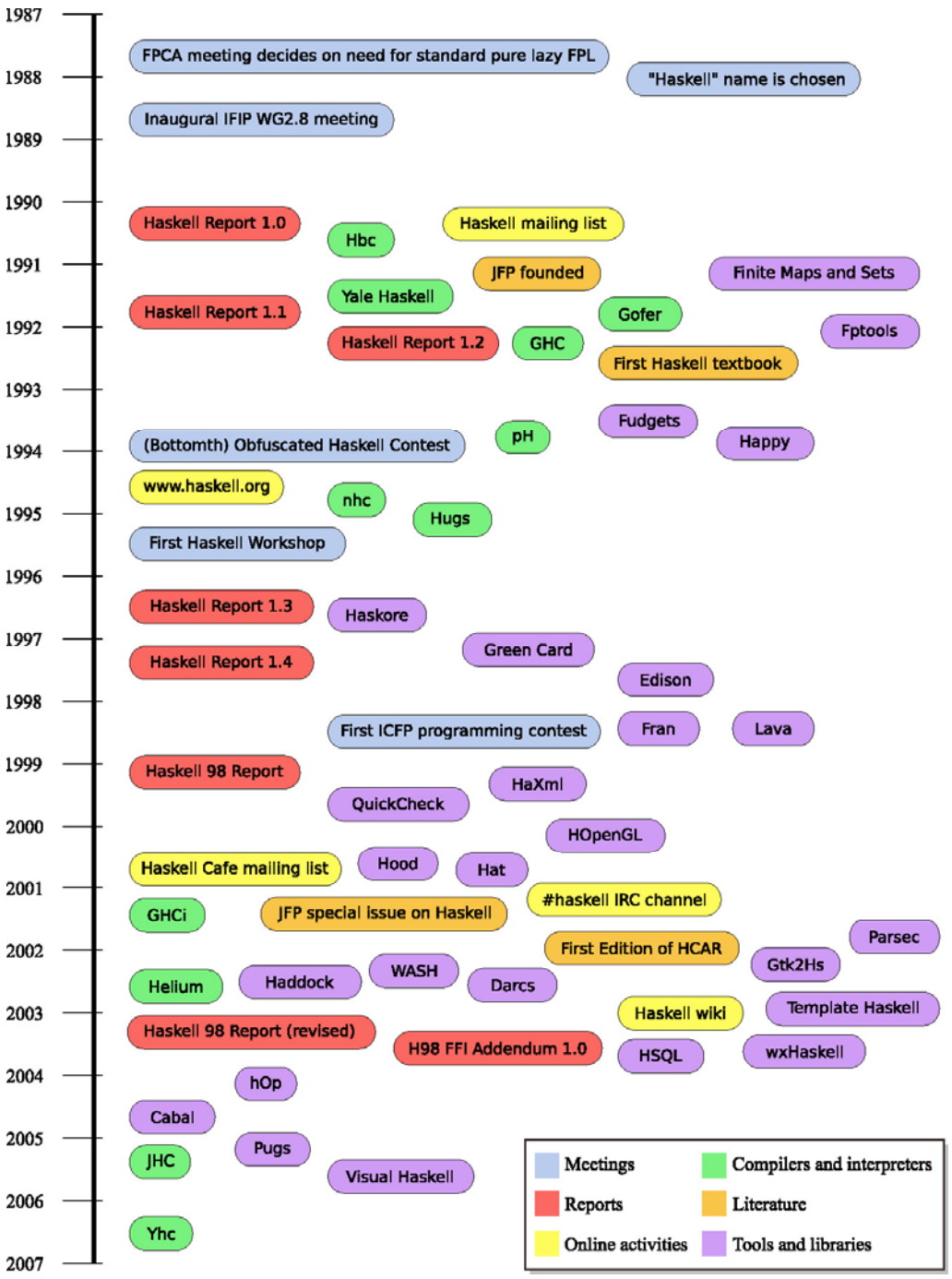
Pe de altă parte, în septembrie 1987, a avut loc o conferință asupra "Functional Programming Languages and Computer Architecture (FPCA '87)" în SUA; în cadrul acestei conferințe s-a hotărât că este momentul să se pună bazele unui nou limbaj de programare funcțională care să aibă cât mai multe dintre proprietățile şi avantajele ale unui limbaj funcțional pur. Acest lucru era necesar întrucât în acel moment existau peste douăsprezece limbaje de programare funcțională, având, în mare, aceeaşi expresivitate şi semantică. Era clar că, în acel moment, datorită "împrăştierii", nu se puteau bucura de succes în aplicarea lor pe scară largă.

Comitetul format în scopul menționat a pornit de la următoarele caracteristici care trebuiau a fi satisfăcute de limbaj, [2]:

1. Trebuie să fie potrivit pentru învățământ, cercetare şi aplicații, inclusiv pentru construcția unor sisteme mari.
2. Trebuie să poată fi descris complet prin publicarea sintaxei şi semanticii sale.
3. Trebuie să fie accesibil în mod liber. Oricine trebuie să aibă voie să implementeze limbajul şi să-l distribuie oricui doreşte.
4. Trebuie să se bazeze pe idei care se bucură de un larg consens.
5. Trebuie să reducă diversitatea (inutilă) a limbajelor de programare funcțională.

Prima variantă a limbajului, Haskell 1.0, a apărut în 1990. De atunci, limbajul s-a dezvoltat continuu, ajungând în 1997 la versiunea 1.4. În acel an, la "Haskell Workshop" din Amsterdam, s-a hotărât că este momentul elaborării primei variante stabile, portabile, cu o bibliotecă standard, şi care să servească drept bază pentru extinderile ulterioare. Această variantă, numită *Haskell 98*, a "suferit" îmbunătățiri continue din punct de vedere al bibliotecii standard numită "the Prelude", şi asta datorită faptului că multe programe scrise în Haskell au nevoie de o bibliotecă de funcții mult mai largă. În momentul de față (2006) se află în plină desfăşurare procesul de definire a succesorului limbajului Haskell 98, numit *Haskell'* ("Haskell Prime"), [4].

Următorul grafic, imaginat şi realizat de Bernie Pope şi Donald Stewart, urmăreşte îndeaproape dezvoltarea, implementarea şi utilizarea limbajului Haskell, şi poate fi găsit în "The History of Haskell", Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler, the Third ACM SIGPLAN History of Programming Languages Conference (HOPL III), San Diego, ACM Press, June 2007." :


1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
FPCA meeting decides on need for standard pure lazy FPL
"Haskell" name is chosen
Inaugural IFIP WG2.8 meeting
Haskell Report 1.0
Hbc
Haskell mailing list
JFP founded
Finite Maps and Sets
Yale Haskell
Haskell Report 1.1
Gofer
Fptools
Haskell Report 1.2
GHC
First Haskell textbook
Fudgets
(Bottomth) Obfuscated Haskell Contest
pH
Happy
www.haskell.org
nhc
Hugs
First Haskell Workshop
Haskell Report 1.3
Haskore
Green Card
Haskell Report 1.4
Edison
First ICFP programming contest
Fran
Lava
Haskell 98 Report
HaXml
QuickCheck
HOpenGL
Haskell Cafe mailing list
Hood
Hat
#haskell IRC channel
GHCi
JFP special issue on Haskell
Parsec
First Edition of HCAR
Gtk2Hs
Helium
Haddock
WASH
Darcs
Haskell wiki
Template Haskell
Haskell 98 Report (revised)
H98 FFI Addendum 1.0
HSQL
wxHaskell
hOp
Cabal
Pugs
JHC
Visual Haskell
Yhc
Meetings
Compilers and interpreters
Reports
Literature
Online activities
Tools and libraries

2. Proprietăți caracteristice

Dintre metodele şi instrumentele utilizate de limbajul Haskell amintim:

- **confruntarea şabloanelor** (pattern matching) sau "nu-nțelege dar le potriveşte";
- **curry-zarea** – a se vedea în capitolul precedent modul de trecere de la funcții de mai multe variabile la compuneri de funcții de o singură variabilă;
- **comprehensiunea listelor** (list comprehension) – procedeu de construcție pentru procesarea listelor, asemănător cu definirea mulțimilor prin proprietăți caracteristice ale elementelor;
- **gărzi** (guards) – utilizate pentru specificarea diferită a corpului funcțiilor în funcție de anumite condiții de îndeplinit;
- **recursivitatea** – (am discutat despre acest concept în capitolul anterior);
- **evaluarea întârziată** (lazy evaluation) – şi despre acest lucru am discutat în capitolul precedent;

Spre deosebire de alte limbaje, Haskell a introdus în programare utilizarea unor concepte noi:

- **monade** – ele sunt inspirate din teoria categoriilor (domeniu al algebrei abstracte) şi sunt utilizate, în principal, pentru a impune în cadrul unui program funcțional executarea unor operații într-o ordine specificată; vom reveni în capitolele următoare asupra lor.
- **clasă de tipuri** (tipul class) – este o construcție a sistemului de tipuri din Haskell prin specificarea operațiilor care trebuie şi pot fi imple-

mentate fiecărui tip din cadrul unei clase. **Atenție:** această noțiune este diferită de noțiunea de *clasă* din programarea orientată obiect.

Având aceste caracteristici şi metode, limbajul Haskell realizează foarte uşor implementarea funcțiilor, lucru care nu se poate afirma despre limbajele de programare procedurale. Înainte de orice, să revenim şi să mai punem în evidență unele proprietăți ale acestui limbaj, care nu se regăsesc la cele imperative:

Haskell este un limbaj leneş sau întârziat ("lazy language"), adică un program scris în Haskell nu face nici un calcul decât atunci când este forțat de necesitatea utilizării acelei evaluări într-un anumit calcul. Acest lucru nu este numai o optimizare a calculului ci, mai degrabă, un mod extrem de puternic de lucru. Liniile de cod care, altfel, sunt bucle infinite sau "consumă" largi porțiuni de memorie devin instrumente foarte simple de utilizat în Haskell, şi asta deoarece nu există `for` sau `while` în limbaj

Haskell face o distincție clară între *valori* (numere 1,2,3,...; şiruri: "abc", "salut", ...; caractere: 'a','b',...; chiar şi funcții: funcția de ridicare la pătrat, sau funcția radical) şi *tipuri* (categoriile din care care fac parte valorile) (*case sensitive*). Aceasta nu este o caracteristică numai a Haskell – ului, întrucât marea majoritate a limbajelor au un anumit sistem de tipuri. Ceea ce este specific însă este faptul că numele date funcțiilor şi valorilor încep cu literă mică şi numele dat tipurilor încep cu literă mare. **Atenție**: dacă un anumit program, de altfel corect, nu poate fi compilat şi dă erori, trebuie verificat dacă nu cumva s-a denumit o funcție cu literă mare.

Haskell nu are efecte secundare. Un efect secundar este ceea ce se întâmplă în cursul execuției unei funcții, care nu este legat de ieşirea funcției. De exemplu, într-un limbaj cum ar fi C sau Java, avem voie să modificăm

variabilele globale dintr-o funcție. Acest lucru este un efect secundar, întrucât modificările aduse acestei variabile globale nu sunt legate de ieşirea produsă de funcție. Mai mult, modificarea starii "lumii reale" este considerată efect secundar: apariția unor lucruri pe ecran, citirea unui fişier, etc., toate sunt operații cu efecte secundare. Funcțiile care nu au efecte secundare se numesc *funcții pure*. Pentru a vedea dacă o anumită funcție este pură sau nu este trebuie studiat dacă: "Date aceleaşi argumente, funcția va produce tot timpul aceleaşi ieşiri?"

Acesta este unul dintre motivele pentru care mulți programatori în C sau Java au probleme în înțelegerea modelului funcțional de programare. Din acest punct de vedere, *valorile* trebuie gândite în mod diferit. De exemplu, o valoare x nu trebuie gândită ca un registru, o locație de memorie sau orice altceva de această natură. x este, simplu, un nume, dupa cum "Haskell" este numele limbajului pe care dorim să-l studiem. Nu se poate decide, în mod arbitrar, să se "stocheze" un limbaj diferit în numele dat. Asta înseamnă că următoarele linii de cod în C nu au contrapartidă în Haskell:

```
Int x = 5;
x = x+1;
```

O chemare de gen x = x + 1 este numită *reînnoire distructivă* (*destructive update*), întrucât se distruge orice era în x înainte şi se înlocuieşte cu noua valoare. Aceste reînnoiri nu există în Haskell. În schimb, o variabilă Haskell "spune" compilatorului ceva de genul: "dacă vreodată ai nevoie să ştii valoarea lui x, uite cum o poți calcula".

Codul scris în Haskell este uşor de înțeles întrucât nu admite reînnoiri distructive. Adică dacă vom defini o funcție f şi chemăm acea funcție cu un anumit argument a la începutul programului şi, la sfârşitul programului

chemăm din nou f cu acelaşi argument a vom şti că vom obține acelaşi rezultat. Acest lucru se întâmplă deoarece ştim că a nu putea să se schimbe şi că f depinde numai de a. Această proprietate care, în esență, afirmă că, dacă două funcții f şi g produc aceleaşi valori pentru aceleaşi argumente, putem înlocui f cu g (şi vice-versa) se numeşte *transparență referențială.*

Haskell se poate **interpreta sau compila** foarte uşor în cod maşină. De asemenea, permite realizarea unei interfețe cu limbajul C; de exemplu, se pot apela uşor funțiile scrise în C şi asta în numai 2-3 linii de cod. Există, de asemenea, interfețe şi cu Java, .NET sau Python

Scrierea codului în Haskell este, în mod surprinzător, foarte intuitivă, iar citirea şi înțelegerea codului scris sunt, de asemenea, foarte uşoare. Din acest motiv, este mai puțină nevoie de un "debugger" pentru Haskell.

Haskell este un **limbaj de programare didactic**. În acest sens, este mai uşor să scriem programe în Haskell decât în (de exemplu) C++, şi asta din cauză că scrierea celui mai simplu program în C++ necesită cunoştințe despre biblioteci, I/O şi alte detalii sintactice; odată scrise ele trebuie compilate şi apoi rulate înainte de a fi testate. Pentru Haskell, interpretoarele GHCi sau Hugs lucrează mai mult ca nişte calculatoare de buzunar. Modul de execuție al Haskell-ului este bazat pe evaluarea expresiilor şi este mai uşor de înțeles. Pentru a înțelege programele C++ trebuie să se înțeleagă mai întâi "modelul maşină". Mai mult, majoritatea algoritmilor se scriu în Haskell aproape în acelaşi "limbaj" ca acela în care descriem algoritmul. Astfel, pentru începătorii în programare este mai uşor să înceapă cu Haskell. Din acest punct de vedere, Haskell este de nivel mai înalt decât C++, fiind orientat mai mult către utilizator decât către maşină.

Sperăm că toate proprietățile şi avantajele enunțate mai sus constituie tot atâtea motive pentru a motiva citirea, în continuare, a acestei cărți.

Însă, pentru aceasta, vom avea nevoie şi de:

3. Implementări şi instalare.

Marea majoritate a implementărilor limbajului Haskell conțin un număr suficient de mare de biblioteci care să suporte construcțiile şi metodele prezentate în paragraful anterior. Informații privind aceste implementări se pot găsi în bibliografia existentă la sfârşitul acestui capitol, iar o listă a lor se poate consulta deja în graficul prezentat la sfârşitul paragrafului dedicat istoricului acestui limbaj de programare.

Implementarea realizată la Universitatea din Glasgow, **Glasgow Haskell Compiler (GHC)**, este, după părerea noastră, cea mai bună, având, simultan, pe lângă compilator şi interpretor, cele mai complete biblioteci, cea mai completă documentație şi cea mai largă răspândire. De asemenea, există variante pentru marea majoritate a sistemelor de operare şi a platformelor existente la această oră.

GHC – ul compilează programele Haskell sau în cod nativ sau în C. Totodată, el implementează şi multe dintre extensiile experimentale ale Haskell 98. El vine, de asemenea, cu un "garbage collector" automat.

Se poate obține liber, de pe pagina web http://haskell.org/ghc/ de unde se poate descărca ultima versiune potrivită platformei pe care se lucrează. Există distribuții binare pentru aproape orice sistem de operare, iar **instalarea** se realizează urmând aceeaşi paşi ca în cazul oricărui alt program.

În ceea ce urmează ne vom referi numai la distribuția dedicată sistemului de operare Windows – şi asta din cauza faptului că este cel mai răspândit sistem de operare din țara noastră. În orice caz, codul care se va scrie de utilizator nu este atât de legat de sistemul de operare, putând fi exportat uşor către alte sisteme.

Rularea compilatorului:

Să presupunem că avem un program scris în Haskell, numit `Test.hs` care conține o funcție `test`. Compilarea acestui program se face prin scrierea următoarei linii în dreptul prompter-ului:

```
% ghc --make Test.hs -o test.exe
```

Opțiunea "-make" spune compilatorului că avem de compilat un program (şi nu avem de a face cu o simplă bibliotecă) şi că trebuie să construim şi toate modulele de care depinde programul. "Test.hs" este numele fişierului de compilat iar "-o test.exe" spune compilatorului unde se "pune" rezultatul compilării, adică în fişierul executabil test.exe.

Rularea interpretorului:

Vom utiliza, pentru testarea rapidă a codului scris, interpretorul GHCi. El se invocă ori prin comanda "ghci" ori prin selectarea iconiței din Start/Programs/GHC/.... Odată pornit, este încărcat modulul principal, Prelude, care conține toate tipurile, operațiile şi modulele de bază. Dând comanda

```
Prelude> :?
```

se obține un "help" imediat.

Să mai observăm că una dintre opțiunile cele mai importante pe care utilizatorul le poate lua în acest moment este de a deschide toate extensiile, lucru realizabil prin comanda:

```
Prelude> :set –fglasgow-exts
```

Acest lucru este indicat ori de câte ori vom dori să rulăm un cod scris de altcineva, şi cu care avem probleme de încărcare.

4. Scrierea codului sursă

Pentru scrierea codului este nevoie de un "editor de texte". Preferabil ar fi unul pentru care există posibilitatea evidenţierii sintaxei. Pentru Windows, UltraEdit sau TextPad realizează acest lucru, însă ambele sunt soft proprietar. La fel, utilizarea lui Visual Haskell (chiar dacă nu are nevoie de editoare de text) presupune existenţa în sistem a Visual Studio .NET 2003 sau 2005. În orice caz, se poate utiliza fără probleme Notepad-ul de sub Windows, fără a avea însă evidenţierea sintaxei.

Tipuri de fişiere:

Există două tipuri de fişiere Haskell, *.hs sau *.lhs.

Fişierele de tipul ***.lhs** (literate Haskell script) sunt bazate pe un stil alternativ de scriere a codului sursă în limbajul Haskell, stil care încurajează comentariile, ele fiind considerate ca fiind "iniţiale". În acest sens, numai liniile care încep cu ">" sunt tratate ca făcând parte din program, toate celelalte linii sunt comentarii. În felul acesta, utilizatorul este "încurajat" să comenteze, făcând codul mai uşor de înţeles. Utilizând acest stil, un program care cere utilizatorului un număr şi afişează factorialul său ar apărea în modul următor:

```
This literate program prompts the user for a
number and prints the factorial of that number:
```

```
> main :: IO ()

> main = do putStr "Enter a number: "
>           l <- readLine
>           putStr "n!= "
>           print (fact (read l))

  This is the factorial function.
> fact :: Integer -> Integer
> fact 0 = 1
> fact n = n * fact (n-1)
```

Fişierele ***.hs** (Haskell script) apar, mai simplu, fără a avea ">" la începutul liniei de cod; în schimb comentariile apar în unul din următoarele moduri:

- *comentarii de linie*: ele încep cu "--" şi se extind până la sfârşitul liniei
- *comentarii de bloc*: ele încep cu "{-" şi se extind până la închidere, prin "-}". Comentariile de bloc pot conține, în interior, alte comentarii de bloc.

Comentariile se utilizează pentru a explica programul (în engleză) şi sunt ignorate de compilatoare şi interpretoare. De exemplu:

```
module Test2
     where

main =
       putStrLn "Hello World"
```

```
 -- write a string
 -- to the screen
{- f is a function which takes an integer and
produces an integer. {- this is an embedded
comment -} the original comment extends to the
matching end-comment token: -}
f x =
     case x of
         0 -> 1       -- 0 maps to 1
         1 -> 5       -- 1 maps to 5
         2 -> 2       -- 2 maps to 2
         _ -> -1      -- everything else maps to -1
```

Acest exemplu utilizează ambele tipuri de comentării.

Ca şi reguli de bază în realizarea fişierelor sursă:

- comentați liniile de program încă din faza de design şi mențineți-le şi în faza de dezvoltare;
- includeți un comentariu de început în care să specificați numele dumneavoastră, data, scopul realizării programului;
- încercați să includeți, de fiecare dată, declarația de tip a valorilor, expresiilor, funcțiilor şi a altor tipuri de date utilizate (chiar dacă Haskell realizează acest lucru automat în majoritatea cazurilor), codul fiind mai uşor de înțeles de alți utilizatori;
- **Important**. Înainte de a defini noi înşine funcții, operatori ş.a.m.d. trebuie, întotdeauna, să aruncăm o privire peste funcțiile predefinite în `Prelude.hs`.

Vom reveni asupra fişierelor sursă, însă menționăm de la început că am ales să utilizăm în această carte cea de a doua modalitate de scriere a codului sursă (adică "Haskell script" şi nu "literate Haskell script").

Notă: În tot ceea ce urmează, liniile de cod vor fi scrise în fontul `Courier cu caractere neîngro•ate` şi, ori de câte ori ne vom referi la interacțiunea noastră cu sistemul de operare sau cu programul utilizat, vom scrie, tot în fontul **`Courier cu caracterele îngro•ate`**.

5. Un tur rapid al `Prelude` - ului

Una din greşelile comune care se fac, atunci când utilizatorul îşi defineşte propriile funcții într-un program, este că se utilizează acelaşi nume cu numele unor funcții predefinite în fişierul `Prelude.hs`. Din acest motiv am decis ca, în încheierea acestui capitol, să prezentăm funcțiile predefinite din Haskell în Prelude. Această secțiune poate fi lăsată la o parte la o primă lectură, urmând a ne întoarce ori de câte ori este nevoie.

Aşadar, ori de câte ori vom dori să definim o funcție cu numele `foo` şi nu suntem siguri dacă ea nu este cumva predefinită în Prelude sau în modulele care sunt importate de programul pe care îl concepem, este bine să ataşăm "particula" my, adică vom numi funcția prin `my_foo`.

Funcțiile din Prelude:

abs

tipul : `abs :: Num a => a -> a`

descriere: returns the absolute value of a number.

definiție:
```
abs x
  | x >= 0  = x
  | otherwise  = -x
```
utilizare:
```
Prelude> abs (-5)
5
```

all

tipul: `all :: (a -> Bool) -> [a] -> Bool`

descriere: aplicată unui predicat şi o listă, rezultatul este `True` dacă toate elementele listei satisfac acel predicat, şi este False în caz contrar. Este asemănătoare funcției `any`.

definiție: `all p xs = and (map p xs)`

utilizare:
```
Prelude> all (<20) [1..19]
True
Prelude> all isDigit "1a2b3c"
False
```

and

tipul: `and :: [Bool] -> Bool`

descriere: funcția face conjuncția unei liste de valori booleene.

definiție: `and xs = foldr (&&) True xs`

utilizare:
```
Prelude> and [True, False, True, True]
False
Prelude> and [True, True, True, True]
True
Prelude> and []
True
```

any

tipul: `any :: (a -> Bool) -> [a] -> Bool`

descriere: aplicată unui predicat şi o listă, rezultatul este True dacă există elemente în listă care satisfac predicatul, şi False în caz contrar.

definitie: `any p xs = or (map p xs)`

utilizare:

```
Prelude> any (<20) [1..19]
True
Prelude> any isDigit "1a2b3c"
True
Prelude> any isDigit "alphabet"
False
```

atan

tipul: `atan :: Floating a => a -> a`

descriere: funcția trigonometrică tangentă.

definiție: definită intern.

utilizare:

```
Prelude> atan (pi/3)
0.808448792630022
```

break

tipul: `break :: (a -> Bool) -> [a] -> ([a],[a])`

descriere: dat un predicat şi o listă, funcția împarte lista în două liste aflate într-o pereche, în primul punct unde este satisfăcut predicatul. În cazul în care predicatul nu este satisfăcut în nici un punct,

perechea rezultată are primul element întreaga listă inițială iar al doilea element este lista vidă, [].

definiție:
```
break p xs
   = span p' xs
     where
     p' x = not (p x)
```
utilizare:
```
Prelude> break isSpace "salut cititorule"
("salut", "cititorule")
Prelude> break isDigit "lista are cifre ?"
("lista are cifre ?","")
```

ceiling

tipul: `ceiling :: (RealFrac a, Integral b) => a -> b`

descriere: aplicată unui număr real ea ne oferă cel mai mic număr întreg mai mare sau egal cu numărul dat.

utilizare:
```
Prelude> ceiling 2.54
3
Prelude> ceiling (- pi)
-3
```

chr

tipul: `chr :: Int -> Char`

descriere: aplicată unui număr întreg între 0 şi 255, ni se oferă caracterul al cărui codificare ASCII este acel. Această funcție are drept inversă funcția `ord`. Aplicată unui număr întreg în

afara domeniului dat mai înainte, se va obține un mesaj de eroare.

definiție: definită în mod intern.

utilizare:
```
Prelude> chr 65
'A'
Prelude> (ord (chr 65)) == 65
True
```

concat

tipul: `concat :: [[a]] -> [a]`

descriere: aplicată unei liste de liste, funcția le unește utilizând operatorul de concatenare ++

definiție: `concat xs = foldr (++) [] xs`

utilizare:
```
Prelude> concat [[4],[1,2,3],[],[5,6,7]]
[4,1,2,3,5,6,7]
```

cos

tipul: `cos :: Floating a => a -> a`

descriere: funcția trigonometrică cosinus; argumentele sunt interpretate ca fiind în radiani.

definiție: definită intern.

utilizare:
```
Prelude> cos pi
-1.0
Prelude> cos (pi/2)
6.123031769111886e-17
```

digitToInt

tipul: `digitToInt :: Char -> Int`

descriere: această funcție face o conversie: transformă o cifră privită ca şi caracter în valoarea corespunzătoare

definiție:

```
digitToInt :: Char -> Int
digitToInt c
  | isDigit c = fromEnum c - fromEnum '0';
  | c >= 'a' && c <= 'f' = fromEnum c -
fromEnum 'a' + 10;
  | c >= 'A' && c <= 'F' = fromEnum c -
fromEnum 'A' + 10;
  | otherwise = error "Char.digitToInt: not a
digit";
```

utilizare:

```
Prelude> digitToInt '3'
3
```

div

tipul: `div :: Integral a => a -> a -> a`

descriere: această funcție calculează câtul împărțirii dintre argumentele sale.

definitie: definită intern.

utilizare:

```
Prelude> 24 `div` 9
2
```

doReadFile

tipul: `doReadFile :: String -> String`

descriere: această funcție citeşte fişiere: dat numele fişierului ca un string ea oferă conținutul fişierului sub forma unui string. În cazul în care fişierul nu poate fi găsit sau nu poate fi deschis se va afişa un mesaj de eroare.

definiție: definită intern.

utilizare:
```
Prelude> doReadFile "foo.txt"
"This is a small text file,\ncalled
    foo.txt.\n".
```

drop

tipul:
```
drop :: Int -> [a] -> [a]
```

descriere: se aplică unui număr şi unei liste. Se va obține o nouă listă din care sunt şterse atâtea caractere cât este valoarea numărului. În cazul în care lista are mai puține elemente decât este numărul, se obține lista vidă.

definiție:
```
drop 0 xs = xs
drop _ [] = []
drop n (_:xs) | n>0 = drop (n-1) xs
drop _ _ = error "PreludeList.drop:
  negative argument"
```

utilizare:
```
Prelude> drop 5 [1..10]
[6, 7, 8, 9, 10]
Prelude> drop 4 "abc"
""
```

dropWhile

tipul: `dropWhile :: (a -> Bool) -> [a] -> [a]`

descriere: aplicată unui predicat şi o listă, ea scoate toate elementele din capul listei care stisfac acel predicat.

definiţie:
```
dropWhile p [] = []
dropWhile p (x:xs)
      | p x = dropWhile p xs
      | otherwise = (x:xs)
```

utilizare:
```
Prelude> dropWhile (<= 5) [1..10]
[6, 7, 8, 9, 10]
```

elem

tipul: `elem :: Eq a => a -> [a] -> Bool`

descriere: se aplică unei valori şi unei liste iar ca ieşire obţinem True dacă valoarea este în listă şi False în rest. Elementele din listă trebuie să fie de acelaşi tip ca şi valoarea din argument.

definitie: `elem x xs = any (== x) xs`

utilizare:
```
Prelude> elem 5 [1..10]
True
Prelude> elem "rat" ["fat","cat","sat"]
False
```

error

tipul: `error :: String -> a`

descriere: aplicată unui string crează valoare de eroare cu un mesaj asociat. Valorile de eroare sunt echivalente valorii nedefinite

(`undefined`); orice încercare de a accasa valoarea, face ca programul să se oprească în acel punct şi să apară un string ca şi diagnostic.

definiţie: definită intern.

utilizare: **`error "this is an error message"`**

exp

tipul: `exp :: Floating a => a -> a`

descriere: funcţia exponenţială (`exp n` este echivalent cu e^n).

definitie: definită intern.

utilizare:
```
Prelude> exp 1
2.718281828459045
```

filter

tipul: `filter :: (a -> Bool) -> [a] -> [a]`

descriere: aplicată unui predicat şi o listă, ea ne oferă ca rezultat o listă care conţine toate elementele care satisfac acel predicat.

definiţie: `filter p xs = [k | k <- xs, p k]`

utilizare:
```
Prelude> filter isDigit "1.sunt10motani"
"110"
```

flip

tipul: `flip :: (a -> b -> c) -> b -> a -> c`

descriere: apcată unei funcții de două variabile, ea ne oferă aceeaşi funcție având argumentele inversate.

definiție:
```
flip f x y = f y x
```

utilizare:
```
Prelude> flip elem [1..5] 5
True
Prelude> flip div 9 24
2
```

floor

tipul:
```
floor :: (RealFrac a, Integral b) => a -> b
```

descriere: aplicată unui număr, ea calculează cel mai mare număr întreg care nu depăşeşte numărul (partea întreagă a unui număr).

utilizare:
```
Prelude> floor 3.6
3
Prelude> floor (-2.8)
-3
```

foldl

tipul:
```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

descriere: "împachetează" o listă utilizând un operator binar dat şi o valoare dată de pornire, în manieră asociativă stângă.

foldl op r [a, b, c] → ((r `op` a) `op` b) `op`c)

definitie:
```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

utilizare:
```
Prelude> foldl (+) 1 [1..10]
```

```
56
Prelude> foldl (flip (:)) [] [1..5]
[5, 4, 3, 2, 1]
Prelude> foldl (/) 2 [2,2,5]
0.1
```

foldl1

tipul: `foldl1 :: (a -> a -> a) -> [a] -> a`

descriere: împachetează la stânga listele nevide.

definitie: foldl1 f (x:xs) = foldl f x xs

utilizare:

```
Prelude> foldl1 max [1, 1, 5, 2, -1]
5
```

foldr

tipul: `foldr :: (a -> b -> b) -> b -> [a] -> b`

descriere: "împachetează" o listă utilizând un operator binar dat şi o valoare dată de pornire, în manieră asociativă la dreapta.

foldr op r [a, b, c] → a `op` (b `op` (c `op`r))

definiţie:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

utilizare:

```
Prelude> foldr (++) [] ["ala","tur","are"]
"alaturare"
Prelude> foldr (/) 2 [2,2,5]
2.5
Prelude> foldr (/) 2 [2,5,2]
0.4
```

foldr1

tipul: `foldr1 :: (a -> a -> a) -> [a] -> a`

descriere: împachetează la dreapta listele nevide.

definiție:
```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

utilizare:
```
Prelude> foldr1 (*) [1..5]
120
```

fromInt

tipul: `fromInt :: Num a => Int -> a`

descriere: face conversia de la un `Int` la un tip numeric aflat în clasa `Num`

utilizare:
```
Prelude> (fromInt 5)::Float
5.0
```

fromInteger

tipul: `fromInteger :: Num a => Integer -> a`

descriere: face conversia de la un `Integer` la un tip numeric aflat în clasa `Num`.

utilizare:
```
Prelude> (fromInteger 100000000000)::Float
1.0e11
```

fst

tipul: `fst :: (a, b) -> a`

descriere: extrage primul element al unei perechi.

definitie:
```
fst (x, _) = x
```

utilizare: **Prelude> fst (45, "supa")**
45

head

tipul: `head :: [a] -> a`

descriere: scoate primul element al unei liste. Când se aplică unei liste vide se obține eroare.

definitie: `head (x:_) = x`

utilizare:
```
Prelude> head [1..100]
1
Prelude> head ["mere", "pere", "caise"]
"mere"
Prelude> head []
*** Exception: Prelude.head: empty list
```

id

tipul: `id :: a -> a`

descriere: funcția identitate.

definitie: `id x = x`

utilizare:
```
Prelude> id 25
25
Prelude> id (id "mar")
"mar"
Prelude> (map id [1..10]) == [1..10]
True
```

init

tipul: `init :: [a] -> [a]`

descriere: aplicată unei liste, ea ne returnează o lista care are toate elementele celei inițiale cu excepția ultimului element; lista inițială trebuie să fie nevidă, altfel se obține o eroare.

definitie:
```
init [x] = []
init (x:xs) = x : init xs
```

utilizare:
```
Prelude> init [1..8]
[1, 2, 3, 4, 5, 6, 7]
Prelude> init "studenti"
"student"
```

isAlpha

tipul: `isAlpha :: Char -> Bool`

descriere: aplicată unui argument de tip `Char`, se obține `True` dacă acel caracter este literă din alfabet şi `False` în rest.

definiție: `isAlpha c = isUpper c || isLower c`

utilizare:
```
Prelude> isAlpha 'd'
True
Prelude> isAlpha '%'
False
```

isDigit

tipul: `isDigit :: Char -> Bool`

descriere: aplicată unui argument de tip `Char`, se obține `True` dacă acel caracter este cifră şi `False` în rest.

definiție: `isDigit c = c >= '0' && c <= '9'`

utilizare:
```
Prelude> isDigit '1'
True
Prelude> isDigit 'a'
False
```

isLower

tipul: `isLower :: Char -> Bool`

descriere: aplicată unui argument de tip `Char`, se obține `True` dacă acel caracter este literă mică din alfabet şi `False` în rest.

definiție: `isLower c = c >= 'a' && c <= 'z'`

utilizare:
```
Prelude> isLower 'a'
True
Prelude> isLower 'A'
False
Prelude> isLower '1'
False
```

isSpace

tipul: isSpace :: Char -> Bool

descriere: aplicată unui argument de tip `Char`, se obține `True` dacă acel caracter este spațiu şi `False` în rest.

definiție:
```
isSpace c = c == ' '||c == '\t'||
        c == '\n'||c == '\r' ||
        c == '\f' || c == '\v'
```

utilizare:
```
Prelude> dropWhile isSpace " \nhello \n"
"hello \n"
```

isUpper

tipul: `isUpper :: Char -> Bool`

descriere: aplicată unui argument de tip `Char`, se obține `True` dacă acel caracter este literă mare din alfabet şi `False` în rest.

definitie: `isDigit c = c >= 'A' && c <= 'Z'`

utilizare:

```
Prelude> isUpper 'A'
True
Prelude> isUpper 'a'
False
Prelude> isUpper '1'
False
```

iterate

tipul: `iterate :: (a -> a) -> a -> [a]`

descriere: `iterate f x` oferă lista infinită $[x, f(x), f(f(x)), ...]$.

definiție: `iterate f x = x : iterate f (f x)`

utilizare:

```
Prelude> iterate (+1) 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, .....
```

last

tipul: `last :: [a] -> a`

descriere: aplicată unei liste nevide, ea returnează ultimul element din listă.

definiție:

```
last [x] = x
last (_:xs) = last xs
```

utilizare: **Prelude> last [1..10]**
10

length

tipul: `length :: [a] -> Int`

descriere: oferă numărul de elemente dintr-o listă.

definiție:
```
length [] = 0
length (x:xs) = 1 + length xs
```

utilizare: **Prelude> length [1..10]**
10

lines

tipul: `lines :: String -> [String]`

descriere: aplicată unei liste de caractere care conțin marcaje de linie nouă, `\n`, ea oferă o listă de liste, prin desfacerea listei originale în liste utilizând marcajele drept delimitări. Marcajele sunt scoase din lista rezultată.

definiție:
```
lines [] = []
lines (x:xs)
     = l : ls
     where
     (l, xs') = break (== '\n') (x:xs)
     ls
       | xs' == [] = []
       | otherwise = lines (tail xs')
```

utilizare: **Prelude> lines "hello\nit's me,\neric\n"**
["hello", "it's me,", "eric"]

log

tipul: `log :: Floating a => a -> a`

descriere: oferă logaritmul natural al argumentului.

definiție: definită intern.

utilizare:

```
Prelude> log 1
0.0
Prelude> log 3.2
1.1631508098056809
Prelude> log (exp 4)
4.0
```

map

tipul: `map :: (a -> b) -> [a] -> [b]`

descriere: aplicată unei funcții, şi unei liste de orice tip, ea oferă o listă în care orice element este rezultatul aplicării funcției elementului corespunzător din lista inițială.

definiție: `map f xs = [f x | x <- xs]`

utilizare:

```
Prelude> map sine [0,pi,pi/2]
[0.0,-1.2246063538223773e-16,1.0]
```

max

tipul: `max :: Ord a => a -> a -> a`

descriere: aplicată în două valori de acelaşi tip, pentru care avem definită o relație de ordine, ea returnează maximul dintre cele două elemente conform operatorului >=.

definiție:
```
max x y
      | x >= y = x
      | otherwise = y
```
utilizare:
```
Prelude> max 3 2
3
```

maximum

tipul: `maximum :: Ord a => [a] -> a`

descriere: aplicată unei liste nevide pentru ale cărei elemente avem definită o relație de ordine, ea ne oferă elementul maxim din listă.

definiție:
```
maximum xs = foldl1 max xs
```
utilizare:
```
Prelude> maximum [-10, 0 , 5, 22, 13]
22
```

min

tipul: `min :: Ord a => a -> a -> a`

descriere: aplicată în două valori de acelaşi tip, pentru care avem definită o relație de ordine, ea returnează minimul dintre cele două elemente conform operatorului =<.

definiție:
```
min x y
      | x <= y = x
      | otherwise = y
```
utilizare:
```
Prelude> min 3 2
2
```

minimum

tipul: `minimum :: Ord a => [a] -> a`

descriere: aplicată unei liste nevide pentru ale cărei elemente avem definită o relație de ordine, ea ne oferă elementul minim din listă.

definiție: `minimum xs = foldl1 min xs`

utilizare:
```
Prelude> minimum [-10, 0 , 5, 22, 13]
-10
```

mod

tipul: `mod :: Integral a => a -> a -> a`

descriere: aplicată în două argumente, se obține restul modulo cel de-al doilea argument a primului.

definiție: definită intern.

utilizare:
```
Prelude> 15 `mod` 6
3
```

not

tipul: `not :: Bool -> Bool`

descriere: returnează negația logică a argumentului său boolean.

definiție:
```
not True = False
not False = True
```

utilizare:
```
Prelude> not (3 == 4)
True
```

```
Prelude> not (10 > 2)
False
```

or

tipul: `or :: [Bool] -> Bool`

descriere: aplicată unei liste de valori booleene, ea returnează disjuncția lor logică.

definiție: `or xs = foldr (||) False xs`

utilizare:

```
Prelude> or [False, False, True, False]
True
Prelude> or [False, False, False, False]
False
Prelude> or []
False
```

ord

tipul: `ord :: Char -> Int`

descriere: aplicată unui caracter, ea returnează codul său ASCII ca întreg.

definiție: definită intern.

utilizare:

```
Prelude> ord 'A'
65
Prelude> (chr (ord 'A')) == 'A'
True
```

pi

tipul: `pi :: Floating a => a`

descriere: numărul π, adică raportul dintre circumferința unui cerc şi diametrul său.

definiție: definită intern.

utilizare:
```
Prelude> pi
3.14159
Prelude> cos pi
-1.0
```

putStr

tipul: `putStr :: String -> IO ()`

descriere: ia ca argument un string şi returnează o acțiune I/O . Efectul secundar al aplicării `putStr` este că face ca string-ul argument să apară pe ecran.

definiție: definită intern.

utilizare:
```
Prelude> putStr "Hello World\nI'm here!"
Hello World
I'm here!
```

product

tipul: `product :: Num a => [a] -> a`

descriere: aplicată unei liste de numere, ea returnează produsul lor.

definiție: `product xs = foldl (*) 1 xs`

utilizare:
```
Prelude> product [1..5]
120
```

repeat

tipul: `repeat :: a -> [a]`

descriere: dată o valoare, ea returnează o listă infinită de elemente cu aceeaşi valoare.

definiție:
```
repeat x
     = xs
     where xs = x:xs
```

utilizare:
```
Prelude> repeat 12
[12, 12, 12, 12, 12, 12, 12, 12, ....
```

replicate

tipul: `replicate :: Int -> a -> [a]`

descriere: dat un număr natural şi o valoare, ea returnează o listă conținând numărul de instanțe al acelei valori.

definiție: `replicate n x = take n (repeat x)`

utilizare:
```
Prelude> replicate 3 "apples"
["apples", "apples", "apples"]
```

reverse

tipul: `reverse :: [a] -> [a]`

descriere: aplicată unei liste de orice tip, ea returnează lista acelor elemente, însă în ordine inversă.

definiție: `reverse = foldl (flip (:)) []`

utilizare:
```
Prelude> reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

round

tipul: `round :: (RealFrac a, Integral b) => a -> b`

descriere: rotunjeşte argumentul la cel mai apropiat număr întreg.

utilizare:

```
Prelude> round 3.2
3
Prelude> round 3.5
4
Prelude> round (-3.2)
-3
```

show

tipul: `show :: Show a => a -> String`

descriere: converteşte o valoare (membră a clasei `Show`),
la reprezentarea sa ca string.

definiţie: definită intern.

utilizare:

```
Prelude>"six plus two equals"++(show (6+2))
"six plus two equals 8"
```

sine

tipul: `sine :: Floating a => a -> a`

descriere: funcţia trigonometrică sinus, argumentele fiind interpretate în radiani.

definiţie: definită intern.

utilizare:

```
Prelude> sin (pi/2)
1.0
Prelude> ((sin pi)^2) + ((cos pi)^2)
1.0
```

snd

tipul: `snd :: (a, b) -> b`

descriere: returnează al doilea element dintr-o pereche.

definiție: `snd (_, y) = y`

utilizare:
```
Prelude> snd ("harry", 3)
3
```

sort

tipul: `sort :: Ord a => [a] -> [a]`

descriere: sortează lista pe care o are drept argument o listă în oredine crescătoare. Elementele listei trebuie să fie în clasa `Ord`.

utilizare:
```
List> sort [1, 4, -2, 8, 11, 0]
[-2,0,1,4,8,11]
```

observație: Această funcție nuj este definită în `Prelude`. Trebuie importat modulul `List.hs`.

span

tipul: `span :: (a -> Bool) -> [a] -> ([a],[a])`

descriere: dat un predicat şi o listă, funcția oferă o pereche de două liste în care primul element al perechii este o listă care conține toate elementele de la începutul listei inițiale care satisfac predicatul, al doilea element al perechii fiind lista restului de elemente din lista inițială.

definiție: `span p [] = ([],[])`

```
span p xs@(x:xs')
     | p x = (x:ys, zs)
     | otherwise = ([],xs)
       where (ys,zs) = span p xs'
```

utilizare:

```
Prelude> span (<5) [1,2,3,3,4]
([1,2,3,3,4],[])
Prelude> span (<5) [1,6,3,3,4]
([1],[6,3,3,4])
```

splitAt

tipul: `splitAt :: Int -> [a] -> ([a],[a])`

descriere: dat un număr natural *n* şi o listă, această funcţie scindează lista într-o pereche de două liste, prima având primele *n* elemente din lista inițială, cea de-a doua având restul elementelor. Dacă *n* este mai mare decât lungimea listei, ea returnează perechea care are prima componentă lista inițială, a doua componentă fiind lista vidă.

definiţie:

```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
     | n > 0 = (x:xs',xs'')
          where
          (xs',xs'') = splitAt (n-1) xs
splitAt _ _ = error "PreludeList.splitAt:
  negative argument"
```

utilizare:

```
Prelude> splitAt 3 [1..10]
([1, 2, 3], [4, 5, 6, 7, 8, 9, 10])
Prelude> splitAt 5 "abc"
```

```
("abc", "")
```

sqrt

tipul: `sqrt :: Floating a => a -> a`

descriere: returnează rădăcina pătrată a unui număr.

definiție: `sqrt x = x ** 0.5`

utilizare:

```
Prelude> sqrt 16
4.0
```

subtract

tipul: `subtract :: Num a => a -> a -> a`

descriere: scade primul argument din cel de-al doilea argument.

definiție: `subtract = flip (-)`

utilizare:

```
Prelude> subtract 7 10
3
```

sum

tipul: `sum :: Num a => [a] -> a`

descriere: calculează suma unei liste finite de numere.

definiție: `sum xs = foldl (+) 0 xs`

utilizare:

```
Prelude> sum [1..10]
55
```

tail

tipul: `tail :: [a] -> [a]`

descriere: aplicată unei liste nevide, returnează lista fără primul element.

definiție:
```
tail (_:xs) = xs
```

utilizare:
```
Prelude> tail [1,2,3]
[2,3]
Prelude> tail "hugs"
"ugs"
```

take

tipul:
```
take :: Int -> [a] -> [a]
```

descriere: aplicată unui număr natural şi o listă, ea returnează numărul specificat de elemente din listă. Dacă lista are mai puține elemente decât cel specificat, ea returnează întreaga listă.

definiție:
```
take 0 _ = []
take _ []= []
take n (x:xs)
     | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take:
   negative argument"
```

utilizare:
```
Prelude> take 4 "goodbye"
"good"
Prelude> take 6 [1,2,3]
[1,2,3]
```

takeWhile

tipul:
```
takewhile :: (a -> Bool) -> [a] -> [a]
```

descriere: aplicată unui predicat şi o listă, ea returnează o listă care conține elementele din capul listei care satisfac predicatul.

definiție:
```
takeWhile p [] = []
takeWhile p (x:xs)
      | p x = x : takeWhile p xs
      | otherwise = []
```

utilizare:
```
Prelude> takeWhile (<5) [1, 2, 3, 10, 4, 2]
[1, 2, 3]
```

tan

tipul: `tan :: Floating a => a -> a`

descriere: funcția trigonometrică tangenta, argumentele fiind interpretate în radiani.

definiție: definită intern.

utilizare:
```
Prelude> tan (pi/4)
1.0
```

toLower

tipul: `toLower :: Char -> Char`

descriere: această funcție converteşte caracterele alfabetice mari în corespondentele lor mici. Dacă există şi alte caractere, ele rămân nemodificate.

definiție:
```
toLower c
      | isUpper c = toEnum (fromEnum c -
        fromEnum 'A' + fromEnum 'a')
      | otherwise = c
```

utilizare:
```
Prelude> toLower 'A'
'a'
Prelude> toLower '3'
'3'
```

toUpper

tipul: `toUpper :: Char -> Char`

descriere: această funcție convertește caracterele alfabetice mici în corespondentele lor mari. Dacă există și alte caractere, ele rămân nemodificate.

definiție:
```
toUpper c
     | isLower c = toEnum (fromEnum c -
       fromEnum 'a' + fromEnum 'A')
     | otherwise = c
```

utilizare:
```
Prelude> toUpper 'a'
'A'
Prelude> toUpper '3'
'3'
```

truncate

tipul: `truncate ::(RealFrac a,Integral b)=> a-> b`

descriere: trunchează partea fracțională.

utilizare:
```
Prelude> truncate 3.2
3
Prelude> truncate (-3.2)
-3
```

unlines

tipul: `unlines :: [String] -> String`

descriere: este funcția care este inversa funcției `lines`. Converteşte o listă de string-uri într-un singur string, plasând marcajul de linie nouă "`\n`" între fiecare dintre string-urile inițiale.

definiție:
```
unlines xs
     = concat (map addNewLine xs)
     where
     addNewLine l = l ++ "\n"
```

utilizare:
```
Prelude> unlines ["Maria","are","pere"]
"Maria\nare\npere\n"
```

until

tipul: `until :: (a -> Bool) -> (a -> a) -> a -> a`

descriere: dat un predicat, o funcție unară şi o valoare, funcția este reaplicată în mod recursiv până când este satisfăcut predicatul. Dacă predicatul nu este satisfăcut niciodată, `until` nu va termina.

definiție:
```
until p f x
     | p x = x
     | otheriwise = until p f (f x)
```

utilizare:
```
Prelude> until (>1000) (*2) 1
1024
```

unwords

tipul: `unwords :: [String] -> String`

descriere: concatenează o listă de string-uri într-unul singur, punând câte un spațiu între ele.

definiție:
```
unwords [] = []
unwords ws
     = foldr1 addSpace ws
     where
     addSpace w s = w ++ (' ':s)
```

utilizare:
```
Prelude> unwords ["Maria","invata","bine"]
"Maria invata bine"
```

words

tipul:
```
words :: String -> [String]
```

descriere: inversa funcției de mai sus, ea sparge string-ul argument într-o listă de cuvinte, fiecare cuvănt fiind delimitat de unul sau mai multe spații libere.

definiție:
```
words s
     | findSpace == [] = []
     | otherwise = w : words s''
     where
     (w, s'') = break isSpace findSpace
     findSpace = dropWhile isSpace s
```

utilizare:
```
Prelude> words "Maria invata bine"
["Maria","invata","bine"]
```

zip

tipul:
```
zip :: [a] -> [b] -> [(a,b)]
```

descriere: are ca argument două liste, şi returnează o listă de perechi obținută prin punerea în aceeaşi pereche a elementelor corespunzătoare ale listelor date. Lista rezultată are lungimea listei mai scurte din argument.

definiție:
```
zip xs ys
= zipWith pair xs ys
     where
     pair x y = (x, y)
```

utilizare:
```
Prelude> zip [2..7] "abcd"
[(2, 'a'), (3, 'b'), (4, 'c'), (5, 'd')]
```

zipWith

tipul: `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

descriere: aplicată unei funcții binare şi două liste, ea returnează o listă care conține elementele formate prin aplicarea funcției elementelor corespunzătoare din liste.

definiție:
```
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

utilizare:
```
Prelude> zipWith (*) [1..5] [6..10]
[6, 14, 24, 36, 50]
```

Capitolul III

Bazele limbajului Haskell

1. Aritmetică în Haskell.

Să începem primele explorări în Haskell prin evaluarea unor *expresii*. Prin expresie vom înțelege ceva care are o valoare. De exemplu, 3 este o expresie a cărei valoare este 3; 2*3+4 este o expresie a cărei valoare este 10.

De fapt, toate operațiile aritmetice simple sunt suportate de Haskell, adică: adunarea (+), scăderea (-), înmulțirea (*), împărțirea (/), exponențierea (^) şi rădăcina pătrată (sqrt).

Deschideți interpretorul de Haskell preferat (noi ne vom referi la GHCi) şi încercați câteva evaluări de astfel de expresii. De exemplu:

```
Prelude> 2*5-4
6
Prelude> 3^6+1
730
Prelude> sqrt 5
2.23606797749979
Prelude> sqrt(2*13-8)
4.242640687119285
Prelude> sqrt 2*13-8
10.384776310850238
Prelude> sqrt (2*13)-8
-2.9009804864072155
```

Trebuie să se aibă în vedere modul în care se pun parantezele, întrucât există regulile de *precedență a operatorilor*.

Încercați și ceva de genul 2^5000.

În fapt, GHCi "cunoaşte" foarte multe funcții şi operatori care pot fi utilizați în construcția şi utilizarea expresiilor (predefinite în `Prelude`). În tabelul următor vom încerca să cuprindem majoritatea acestor operatori (funcțiile predefinite fiind deja prezentate în capitolul anterior):

Op	Semnificație	Tipul	Asociativitate	Precedență
!!	extrage din listă	`[a] -> Int -> a`	stânga	9
.	compunere	`(a -> b) -> (c -> a) -> c -> b`	dreapta	9
**	ridicare la putere	`Floating a => a -> a -> a`	dreapta	8
^	ridicare la putere	`(Integral b, Num a) => a -> b -> a`	dreapta	8
*	înmulțire	`Num a => a -> a -> a`	stânga	7
/	împărțire	`Fractional a => a -> a -> a`	stânga	7
+	adunare	`Num a => a -> a -> a`	stânga	6
-	scădere	`Num a => a -> a -> a`	stânga	6
:	constructor de liste	`a -> [a] -> [a]`	dreapta	5
++	concatenare	`[a] -> [a] -> [a]`	dreapta	5
/=	nu este egal	`Eq a => a -> a -> Bool`	nu	4
==	egalitate	`Eq a => a -> a -> Bool`	nu	4
<	mai mic	`Ord a => a -> a -> Bool`	nu	4
<=	mai mic sau egal	`Ord a => a -> a -> Bool`	nu	4
>	mai mare	`Ord a => a -> a -> Bool`	nu	4

>=	mai mare sau egal	`Ord a => a -> a -> Bool`	nu	4
&&	conjuncția logică	`Bool -> Bool -> Bool`	dreapta	3
||	disjuncția logică	`Bool -> Bool -> Bool`	dreapta	3

3. Upluri

Pe lângă o singură valoare, putem sa folosim şi valori multiple. De exemplu, am putea dori să ne referim la poziția unui punct ceea ce implică coordonatele sale. Pentru un punct de coordonate întregi, este simplu. Punem perechea între paranteze rotunde, le despărțim prin virgulă şi ... gata. Încercați:

```
Prelude> (2,4)
(2,4)
```

În Haskell, este admis să avem perechi *eterogene*: adică este permis să avem o pereche formată dintr-un număr întreg şi un cuvânt. Acest lucru contrastează cu *listele* (vezi paragraful următor), listele trebuind să aibă în componență numai elemente de acelaşi tip. Există două funcții predefinite pentru perechi (şi numai pentru perechi !) care permit extragerea primei componente şi, respectiv celei de-a doua.

```
Prelude> fst (12, "hello")
12
Prelude> snd (12, "hello")
"hello"
```

Pe lângă perechi, se pot defini triplete, 4 –upluri ş.a.m.d.. Pentru a defini un triplet sau un 4 – uplu, vom scrie:

```
Prelude> (1,2,3)
(1,2,3)
```

```
Prelude> (1,2,3,4)
(1,2,3,4)
```

Exercițiu.

- Scrieți o combinație a funcțiilor `fst` şi `snd` pentru a extrage caracterul ‘a’ în afara uplului: `(("salut",'a'),1)`.

3. Liste

Conceptul de listă.

În cazul uplurilor există o limitare principală: ele pot reține numai un număr fixat de elemente – perechile rețin două, tripletele rețin trei, ş.a.m.d.. O structură de date care poate avea orice număr de elemente este o *listă.* Ansamblarea listelor este foarte asemănătoare cu cea a uplurilor, însă se utilizează parantezele pătrate. O listă se poate defini astfel:

```
Prelude> [1,2]
[1,2]
Prelude> [1,2,3]
[1,2,3]
```

Listele pot să nu aibă nici un element, lista vidă fiind [ ].

Spre deosebire de upluri, putem adăuga foarte simplu elemente la începutul unei liste utilizând operatorul “cons”, reprezentat prin “:”. Etimologia acestui cuvânt provine de la faptul că noi *cons*truim o nouă listă cu un element şi o listă veche. În plus, operatorul cons se poate aplica, succesiv, de mai multe ori. Să exemplificăm:

```
Prelude> 0:[1,2]
[0,1,2]
Prelude> 5:[1,2,3,4]
[5,1,2,3,4]
Prelude> 5:1:2:3:4:[ ]
[5,1,2,3,4]
```

După cum am mai menționat, listele sunt omogene, adică nu putem avea o listă care să conțină şi intregi şi cuvinte. Orice listă poate să conțină şi upluri şi chiar alte liste şi la fel şi uplurile, pot conține upluri sau liste:

```
Prelude> [(1,1),(2,4),(3,9),(4,16)]
[(1,1),(2,4),(3,9),(4,16)]
Prelude> ([1,2,3,4],[5,6,7])
([1,2,3,4],[5,6,7])
```

În Haskell există două funcții de liste, `head` şi `tail`, asemănătoare cu cele de la perechi Funcția `head` extrage primul element al unei liste, în timp ce funcția `tail` ne dă toată lista fără primul element. Lungimea unei liste se obține cu funcția `length`:

```
Prelude> length [1,2,3,4,10]
5
Prelude> head [1,2,3,4,10]
1
Prelude> length (tail [1,2,3,4,10])
4
```

String-uri.

În Haskell, un `String` este, simplu, o listă de caractere, `Chars`. Putem crea string-ul “Hello” astfel:

```
Prelude> 'H':'e':'l':'l':'o':[ ]
```

```
"Hello"
```

Listele, ca şi string-urile pot fi concatenate utilizând operatorul "++":

```
Prelude> "Hello " ++ "World"
"Hello World"
```

În plus, valorile care nu sunt string-uri pot fi convertite la string-uri utilizand funcția `show` iar string-urile pot fi convertite la alte valori utilizând funcția `read`. De asemenea, dacă se încearcă citirea unei valori greşit concepută, se va raporta eroare de rulare (şi nu de compilare).

```
Prelude> "Seven squared is " ++ show (7*7)
"Seven squared is 49"
Prelude> read "6" + 4
10
Prelude> read "Hello" + 3
Program error: Prelude.read: no parse
```

După cum s-a putut vedea, interpretorul şi-a "dat seama" că dorim să adunăm pe 3 la ceva. Asta înseamnă că atunci când execută `read "Hello"`, ne aşteptăm să obținem un număr. Oricum, "Hello" nu poate fi parsat ca număr şi se raportează eroare.

Funcții simple de liste

Mare parte a calculelor în limbajul Haskell se face prin procesarea listelor. Există patru *funcții primare* pentru procesarea listelor: `map`, `filter`, `foldr` şi `foldl`.

Funcția "map". Ea are ca argument o listă de valori şi o funcție care trebuie aplicată fiecărei valori. De exemplu, există o funcție predefinită `Char.toUpper` care are drept intrare un caracter (din `Char`) şi produce

acelaşi caracter însă scris cu literă mare. Astfel, se poate converti un şir (care-i listă de caractere) intr-unul scris cu caractere mari:

```
Prelude> map Char.toUpper "Hello World"
"HELLO WORLD"
```

Să observăm că funcția `map` nu schimbă lungimea listei de intrare.

Funcția "`filter`". Se poate folosi atunci când se doreşte inlăturarea unor elemente dintr-o listă , depinzând de valorile pe care le au şi nu de context. De exemplu, funcția `Char.isLower` ne indică dacă un caracter este mic. Putem filtra într-un string toate caracterele mici:

```
 Prelude> filter Char.isLower "Hello World"
"elloorld"
```

Funcția "`foldr`". Este o funcție având trei argumente: o funcție, o valoare inițială şi o listă. Modul cel mai bun de a o gândi este că ea inlocuieşte toate aparițiile operatorului `cons` (:) cu parametrul funcției şi că înlocuieşte lista vidă ([ ]) cu valoarea inițială. De exemplu, dacă avem lista [5,3,4,2,9] şi vrem să-i aplicăm `foldr (+) 5`, privim lista ca fiind 5:3:4:2:9:[ ] şi vom obține 5 + 3 + 4 + 2 + 9 + 5 care adună 5 la suma elementelor din listă. Să încercăm:

```
Prelude> foldr (+) 5 [5,3,4,2,9]
28
Prelude> foldr (*) 3 [2,4,10]
240
Prelude> foldr (*) 0 [4,8,5]
0
Prelude> foldr (/) 3 [36,2,3]
```

```
18.0
```

Atenție la ceea ce se dorește să se calculeze. Modul de asociere este de la dreapta, adică

foldr (/) 3 [36,2,3]

==> 36 / (foldr (/) 3 [2,3])

==> 36 / (2 / foldr (/) 3 [3])

==> 36 / (2 / (3/ foldr (3) [ ]))

==> 36 / (2 / (3 / 3))

==> 36 / (2 / 1)

==> 36 / 2

==> 18

În fond, şi numele funcției ne spune asta, `foldr` este prescurtare de la "împachetare de la dreapta"

Funcția "`foldl`". Ea, după cum este şi prescurtarea, este "împachetare la stânga".

```
Prelude> foldl (/) 3 [36,2,3]
1.38888888888888e-2
```

Observații.

1. `foldl` este de obicei mai eficientă decât `foldr`
2. `foldr` poate să lucreze cu liste infinite în timp ce `foldl` nu poate să facă asta. Motivul este simplu, înainte de a face orice `foldl` trebuie să "meargă" în coada listei.

   De exemplu, `foldr (:) [ ] [1,2,3,4,5]` ne returnează aceeaşi listă, iar foldl nu poate să o facă.

Exerciții.

1. Utilizați funcția `map` pentru convertirea unui string într-o listă de valori booleene, astfel încât noua listă să reprezinte dacă elementul inițial a fost sau nu literă mică. De exemplu, dacă se consideră şirul "abCdE" să ni se dea lista `[True,True,False,True,False].`
2. Utilizând funcțiile învățate, calculați numărul de litere mici dintr-un string.
3. Ştiind că funcția `max` aplicată unei perechi de numere ne oferă cel mai mare, scrieți o funcție care să extragă maximul dintr-o listă de numere pozitive (utilizând `foldr` sau `foldl`)
4. Scrieți o funcție care se aplică unei liste (de lungime minim 2) de perechi şi oferă ca ieşire prima componentă a celei de-a doua perechi.

4. Crearea fişierelor proprii.

Ca şi programatori, nu dorim să evaluăm numai expresii mici, simple, ca acelea pe care le-am utilizat până acum.

De la bun început trebuie să subliniem faptul că, în Haskell, **spațierea** şi **alinierea la stânga** sunt foarte importante. Acest lucru permite în Haskell să se poată scrie fără a pune în mod explicit ";" sau paranteze.

Cu riscul de a plictisi, insistăm asupra faptului că, în Haskell, toate definițiile se supun **regulii offside - ului**:

o definiție se termină o dată cu prima apariție
a unui text care se află la aceeaşi aliniere la stânga

sau la stânga startului definiției

Noi vom încerca să folosim şi următoarea regulă, **regula alinierii:**

Partea dreaptă a oricărei ecuații

trebuie să fie, ca aliniere, la dreapta semnului "="

În plus, Haskell impune următoarele reguli privind denumirile:

- numele funcțiilor trebuie să înceapă cu literă mică (exemplu: `square, ratio, min`)
- numele argumentelor trebuie să înceapă cu literă mică (`x, y, n`)
- numele tipurilor trebuie să înceapă cu literă mare (`Int, Float`)
- numele constantelor trebuie să înceapă cu literă mare(`True, False`)
- respectând regulile precedente,numele pot conține orice combinație de litere cifre, şi alte semne ("_", " ' ", etc)

Să încercăm acum să definim funcții în cod-sursă şi apoi să le utilizăm în mediul interactiv. Să creăm un fişier numit Test.hs introducând următorul cod:

```
module Test
     where
x = 5
y = (6, "Hello")
z = x * fst y
```

Acesta este un program foarte simplu scris în Haskell. El defineşte un modul numit "`Test`" (în general, numele modulelor trebuie să coincidă cu numele fişierelor create). În acest modul sunt trei definiții: x, y şi z. Odată ce

ați salvat acest fişier, mergeți în directorul în care se află şi îl puteți încărca în două moduri:

```
% ghci Test.hs
```

sau, în cazul în care interpretorul GHCi este deja încărcat, puteți utiliza comanda ":load" sau ":l" pentru încărcarea modulului:

```
Prelude> :l Test.hs
...
Test>
```

Se observă că interpretorul, acolo unde scria "`Prelude`" scrie "`Test`". Acest lucru înseamnă că `Test` este modulul curent. Modulul `Prelude` este întotdeauna încărcat şi conține definițiile standard. Odată încărcat, putem folosi lucrurile definite în modul:

```
Test> x
5
Test> y
(6,"Hello")
Test> z
30
```

Pentru a reveni la modulul Prelude, singurul lucru de făcut este:

```
Test>:module Prelude
Prelude>
```

Un singur lucru mai avem de lămurit în acest moment. În linile de cod scrise, nu am definit nici o funcție şi am dori să transformăm prin compilare programul nostru într-un executabil de sine stătător.

Vom edita fişierul nostru de lucru, Test.hs, şi schimbăm linia de declarare a modulului din `module Test` în `module Main` şi vom adăuga definiția funcției `main`. Să adăugăm linia:

```
main = putStrLn "Hello World"
```

Salvăm fişierul modificat şi-l compilăm cu comanda:

```
% ghc --make Test.hs -o test.exe
```

Se crează fişierul `test.exe` care se poate executa. Ghiciți rezultatul?

5. Funcții

După ce am văzut cum se pot scrie linii de cod într-un fişier, putem începe să scriem funcții. Să vedem *funcția de ridicare la pătrat*. Mai întâi, deschidem din nou editorul de text şi adăugăm următoarea linie, de definiție a funcției de ridicare la pătrat:

```
square x = x * x
```

Această funcție, `square`, are un singur argument , `x`, şi valoarea ei este egală cu `x * x`.

Putem, de asemenea, să definim şi funcția semn, `signum`, care ne arată că limbajul Haskell suportă şi expresii condiționale:

```
signum x =
        if x < 0
          then -1
          else if x > 0
              then 1
              else 0
```

Ca şi alte limbaje de programare, limbajul Haskell suportă şi construcțiile **`case`**. Ele se utilizează atunci când definiția este "pe cazuri",

adică există mai multe valori pe care le avem de verificat. Să presupunem că dorim să definim o funcție `f` care are valoarea 2 în argumentul 0, 3 în argumentul 1, 4 în argumentul 2 şi -2 în rest. Scrierea unei astfel de funcții cu **`if`** ar fi suficient de lungă şi greu de citit.

```
f x =
     case x of
          0 -> 2
          1 -> 3
          2 -> 4
          _ -> -2
```

În acest program, l-am definit pe `f` ca funcție de argument `x`.

Dacă se doreşte să se scrie cu ";" şi "{}", funcția anterioară se poate scrie:

```
f x = case x of
{ 0 -> 1 ; 1 -> 5 ; 2 -> 2 ; _ -> 1 }
```

Bineînțeles, dacă se utilizează ";" sau "{}", liniile de cod se pot structura oricum se doreşte.

Mai mult, funcția se poate da şi componentă cu componentă:

```
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

Aici însă, ordinea este importantă, întrucât, dacă se pune ultima linie mai întâi, funcția va lua valoarea -1 pentru orice argument.

Se pot defini şi funcții mai complicate utilizând *compunerea funcțiilor:*

```
Test> square (f 1)
25
Test> square (f 2)
4
Test> f (square 1)
5
Test> f (square 2)
-1
```

Utilizând "." pentru compunere, testul precedent se mai putea scrie:

```
Test> (square . f) 1
25
Test> (square . f) 2
4
Test> (f . square) 1
5
Test> (f . square) 2
-1
```

Trebuie să punem compunerea între paranteze pentru ca nu cumva compilatorul să înțeleagă că dorim să compunem square cu valoarea lui f în 1, de exemplu, întrucât o astfel de operație nu are sens.

Funcții Infix

Funcțiile infix sunt acele funcții care sunt compuse, de obicei, din simboluri, şi apar, de obicei între diferite argumente ale funcției pe care o reprezintă. De exemplu, `+, *, ++,` sunt toate funcții de acest tip. În fapt,

toți operatorii descrişi în primul paragraf al acestui capitol sunt funcții infix. Funcțiile infix pot fi văzute ca fiind non-infix dacă le scriem între paranteze:

```
Prelude> 5 + 10
15
Prelude> (+) 5 10
15
```

Pe de altă parte, chiar şi funcțiile care nu sunt infix (cum ar fi `map`) pot fi făcute infix dacă se scriu cu apostrof invers (backquote) simbolul de pe tastaturile americane aflat sub tilda "~":

```
Prelude> map Char.toUpper "Hello World"
"HELLO WORLD"
Prelude> Char.toUpper `map` "Hello World"
"HELLO WORLD"
```

6. Declarații locale

Deseori este nevoie de utilizarea locală a unor funcții.

De exemplu, vrem să determinăm rădăcinile unei ecuații de gradul al doilea, $ax^2 + bx + c = 0$. Am putea folosi următoarea funcție pentru calculul acestor rădăcini:

```
roots a b c =
     ((-b + sqrt(b*b - 4*a*c)) / (2*a),
     (-b - sqrt(b*b - 4*a*c)) / (2*a))
```

În loc de a scrie atâtea litere şi cifre, Haskell oferă posibilitatea legăturilor locale, denumind în cadrul programului o nouă funcție care este chemată ori de câte ori este nevoie.

Definiția cu `let ... in`

```
roots a b c =
     let det = sqrt (b*b - 4*a*c)
     in  ((-b + det) / (2*a),
           (-b - det) / (2*a))
```

Mai mult, putem realiza multiple legături:

```
roots a b c =
     let det = sqrt (b*b - 4*a*c)
         twice_a = 2*a
     in ((-b + det) / twice_a,
          (-b - det) / twice_a)
```

Definiția cu `where`

Aceeaşi funcție, `roots`, poate fi definită şi cu ajutorul clauzei `where` Această clauză defineşte, în cadrul unei funcții, valori şi aceste valori dețin prioritate față de alte valori cu acelaşi nume. De exemplu, pentru următorul cod scris în Haskell,

```
det = "Abracadabra"
roots a b c =
     ((-b + det) / (2*a), (-b - det) / (2*a))
     where det = sqrt(b*b-4*a*c)
f _ = det
```

Valoarea funcției `roots` nu ține cont de declarația aflată înaintea sa pentru `det`, atâta timp cât este "ascunsă" de definiția locala.

Mai mult, nici funcția `f` nu poate "să vadă în interiorul" lui `roots`, singurul lucru pe care-l ştie despre `det` este ceea ce apare la nivelul de

definire cu aceeaşi aliniere, adică şirul de caractere “Abracadabra”. De aceea, `f` este o functie care ia orice argument a acestui sir.

Clauzele pot conține orice număr de subexpresii, dar atenție la aliniere când scriem definițiile. La fel ca mai sus, putem elimina calculul multiplu al lui `2*a` prin introducerea codului următor:

```
roots a b c =
      ((-b + det) / (a2), (-b - det) / (a2))
      where det = sqrt(b*b-4*a*c)
            a2 = 2*a
```

Expresiile din clauzele **`where`** apar după definirea funcției în timp ce, atunci când este mai convenabil, pentru a da mai întâi definiția locală se utilizează clauza **`let...in`**

Clauzele `where` sunt de dorit, întrucât ele permit cititorului să vadă imediat ce funcție a fost definită. În practica de zi cu zi, însă, valorile primesc, de multe ori, nume criptice. Pentru aceste cazuri, clauza `let...in` pare mult mai bună. Atenție la clauza `let` fără a avea şi `in`. Ea apare în blocurile **`do`**, din tipul I/O (vezi pentru detalii şi capitolul următor, paragraful 4).

7. Recursivitate

Într-un limbaj imperativ, structura de bază pentru control este *bucla* (de exemplu bucla `for`).

În Haskell, buclele nu-si au locul, întrucât ele necesită reînnoire distructivă (destructive update). În locul buclelor, Haskell-ul utilizează *recursia.*

O funcție este *recursivă* dacă se "cheamă" pe ea însăşi. Ele există şi în limbajele imperative, însă sunt mult mai puțin utilizate.

Un prototip de funcție recursivă este *funcția factorial*. Într-un limbaj imperativ, ea se poate scrie

```
int factorial(int n) {
    int fact = 1;
    for (int i=2; i <= n; i++)
       fact = fact * i;
     return fact;
}
```

sau

```
int factorial(int n) {
    if (n == 1)
       return 1;
    else
       return n * factorial(n-1);
}
```

Primul cod calculează cu succes factorialul oricărui număr natural, însă ignoră întrucâtva definiția factorialului. Cel de-al doilea cod este versiune recursivă imperativă, însă codul tinde să fie mult mai lent decât versiunea cu buclă.

Definiția factorialului este :

$$n! = \begin{cases} 1 & n = 1 \\ n * (n-1)! & \text{altfel} \end{cases}$$

Această definiție se transcrie aproape cuvânt cu cuvânt în cod Haskell:

```
factorial 1 = 1
factorial n = n * factorial (n-1)
```

În general, recursia este destul de greu de "stăpânit", însă este analogă inducției matematice. În orice caz, se pare că cea mai mare problemă este atunci când avem una sau mai multe definiții inițiale şi una sau mai multe definiții recursive. În cazul funcției factorial este o singură definiție de bază şi o singură definiție recursivă.

Alt exemplu: *Exponențierea numerelor naturale*.

$$a^b = \begin{cases} a & b = 1 \\ a * a^{(b-1)} & \text{altfel} \end{cases}$$

Putem "transcrie" totul în Haskell:

```
exponent a 1 = a
exponent a b = a * exponent a (b-1)
```

La fel cum am definit funcțiile recursive pe numere întregi, le putem defini şi pe liste. De obicei, definiția de bază este pentru lista vidă [ ], iar definiția recursivă este prin utilizarea operatorului `cons`. De exemplu, *funcția de lungime de listă*:

```
my_length [] = 0
my_length (x:xs) = 1 + my_length xs
```

După cum cred că s-a observat, am dat o definiție alternativă a unei funcții deja existente în `Prelude`, motiv pentru care i-am ataşat prefixul `my_`, pentru ca nu cumva să "confuzăm" compilatorul.

În mod asemănător, se poate defini şi funcția de filtrare, `filter`. Din nou, definiția de bază este lista vidă, iar definiția recursivă este dată cu operatorul `cons`.

```
my_filter p [] = []
my_filter p (x:xs) =
     if p x
     then x : my_filter p xs
     else my_filter p xs
```

Să explicăm puțin codul. Odată prezentată lista de forma `(x:xs)`, trebuie să decidem dacă se păstrează sau nu valoarea `x`. Pentru aceasta se utilizează construcția `if then else` şi predicatul `p`. Dacă `p x` este adevărată, atunci se returnează lista care începe cu `x` şi continuă cu rezultatul filtrării cozii listei. Dacă `p x` este falsă, atunci se exclude `x` şi se returnează rezultatul filtrării cozii listei.

Să mai observăm că şi pentru `map` sau `fold` se pot da definiții recursive (pentru `fold` o vom da ceva mai târziu, în capitolele următoare).

Important

Una din principalele critici aduse limbajelor de programare funcționale este că recursia tinde să fie mult mai lentă decât bucla. Chiar dacă partea hardware a computerelor actuale a evoluat foarte mult față de momentul în care se "bătea monedă" pe această critică şi diferențele de performanță şi viteză nu mai sunt atât de evidente, modul de "concepere" a recursiei poate, şi el, să influențeze foarte mult performanța codului pe care l-am scris. Să exemplificăm acest lucru cu ajutorul **şirului lui Fibonacci:**

Şirul Fibonacci este definit prin : $F_n = \begin{cases} 1 & n = 1 \text{ sau } n = 2 \\ F_{n-2} + F_{n-1} & \text{altfel} \end{cases}$.

Vom descrie două funcţii recursive `fib1` şi `fib2` care, în esenţă, fac acelaşi lucru, iau numărul natural *n* ca parametru şi calculează F_n.

În primă variantă, *definiţia funcţiei* `fib1` este cea care urmăreşte, în mod fidel, definiţia dată mai sus:

```
fib1 :: Int -> Int
-- fib1 n returns the n-th Fibonacci number
fib1 1 = 1
fib1 2 = 1
fib1 n = fib1 (n - 1) + fib1 (n - 2)
```

Sa vedem câte evaluări trebuie făcute pentru diverse valori ale `fib1`

```
fib1 5 = fib1 4 + fib 3
       = fib 3 + fib 2 + fib 3
       = fib 2 + fib 1 + fib 2 + fib 3
       = fib 2 + fib 1 + fib 2 + fib 2 + fib 1
```

Aşadar avem `fib1 5 =` 1 + 1 + 1 + 1 + 1 = 5

Pentru `fib1 5`:

– `fib1 3` este evaluat de două ori

– `fib1 2` este evaluat de trei ori

Pentru `fib1 7`:

– `fib1 5` este evaluat de două ori

– `fib1 4` este evaluat de trei ori

– `fib1 3` este evaluat de cinci ori

– `fib1 2` este evaluat de opt ori

Evaluarea lui `fib1 n` creşte rapid odată cu creşterea lui `n`

În cea de a doua variantă, ne vom gândi la numerele lui Fibonacci cu ajutorul perechilor:

Perechea Fibonacci de rang `n` pair este dată prin:

```
fibpair n = (fib n, fib (n + 1))
```

Considerăm funcția `fibpair` care, pentru orice număr natural `n`, ne oferă perechea de rang `n`.

Date două perechi,

```
fibpair (n – 1) = (fib (n - 1), fib n)
fibpair n = (fib n, fib (n + 1))
= (fib n, fib n + fib (n - 1))
```

avem că

```
fibpair n = (m,m + k)
```

unde `(k,m) = fibpair (n - 1)`

Astfel, fib2 este o funcție (de ordin înalt) care ia prima componentă a lui `fibpair:`

```
fib2 = fst . fibpair
```

Pentru a evita orice confuzie, dăm, în continuare şi codul sursă, pe care-l salvați într-un fişier, să spunem Fibo.hs, şi-l încărcați în interpretorul GHCi:

```
module Fibo
     where

{- this is an example how we can improve the
performance of recursion. Compare the definitions of
fib1 with fib2 for the same purpose: compute the nth
Fibonacci number.
Note the performance for large n’s -}
```

```
fib1 1 = 1
fib1 2 = 1
fib1 n = fib1 (n-1) + fib1 (n-2)

fibpair :: Int -> (Int, Int)
-- pre: n >= 1
-- fibpair n returns the nth Fib. pair
fibpair 1 = (1, 1)
fibpair n = (m,m + k)
          where (k,m) = fibpair (n - 1)
fib2 = fst . fibpair
```

După ce ați încărcat fișierul, o sesiune în GHCi va arăta astfel:

```
*Fibo> fib1 40
102334155
```

Unui procesor destul de performant (Athlon64 3200+, 512 MbRAM), i-au trebuit 5 minute, lucrând 100%, pentru a obține `fib1 40`, în schimb `fib2 40` a fost obținut instantaneu.

```
*Fibo> fib2 40
102334155
```

Exerciții.

1. Definiți o funcție recursivă `mult` care înmulțește două numere naturale utilizând numai adunarea.
2. Definiți o funcție recursivă `my_map` care se comportă la fel cu funcția standard `map`.

8. Interacțiune.

Poate că v-ați întrebat deja de ce nu ați întâlnit deja unul din programele standard în alte limbaje de programare de gen interactiv (de exemplu un mic program care să ceară utilizatorului numele şi să-i răspundă cu un salut). Fiind un limbaj funcțional pur, nu este foarte clar cum putem stăpâni operații de gen "input utilizator". Dacă am presupune că am modelat o funcție care citeşte un cuvânt scris de utilizator, şi chemăm funcția de două ori, iar utilizatorul scrie prima oară ceva şi a doua oară altceva, nu vom mai avea o funcție, întrucât ar returna două valori distincte. Soluția pentru aşa ceva s-a găsit în teoria categoriilor, de către Philip Wadler, prin aşa numitul concept de *monadă*. În acest moment nu suntem încă pregătiți să abordăm acest concept, dar ne vom gândi la monade ca la un mod convenabil de exprimare a operațiilor de gen input/output.

Să presupunem că dorim să scriem o funcție care este interactivă. Pentru aceasta vom folosi cuvântul cheie **`do`**. Acest cuvânt permite să se specifice ordinea operațiilor (Haskell, fiind limbaj întârziat, nu ține cont în mod specific de o anumită ordine de evaluare).

Scrieți următorul cod în editorul de texte după care salvați fişierul ca "`Name.hs`":

```
module Main
     where

import IO
```

```
main = do
   hSetBuffering stdin LineBuffering
putStrLn "Please enter your name: "
name <- getLine
putStrLn ("Hello, " ++ name ++ ", how are you?")
```

Puteți sau să încărcați modulul în interpretor şi să se execute `main` numai scriind "`main`", sau să compilați fişierul şi să-l rulați. Din punct de vedere interactiv, rezultatele ar fi de genul:

```
Main> main
Please enter your name:
Mihai
Hello, Mihai, how are you?
Main>
```

La linia a 4-a am importat biblioteca IO pentru a putea accesa funcțiile I/O. Începând cu linia a 7-a, prin instrucțiunea **do** spunem Haskell-ului că executăm o secvență de comenzi.

Prima comandă este `hSetBuffering stdin LineBuffering`, (să menționăm aici că această comandă este cerută doar de GHC, în Hugs putând să nu o punem). Această comandă este necesară întrucât, atunci când GHC citeşte intrarea, se aşteaptă să o citească în blocuri mari. Un nume de persoană poate să "umple" aceste blocuri. Atunci când se încearcă să se citească din `stdin` se aşteaptă până când se obține un întreg bloc. Pentru a depăşi acest lucru, spunem programului să utilizeze `LineBuffering` în loc de `BlockBuffering`.

Următoarea comandă este `putStrLn` care afişează un sir de caractere pe ecran. Pe linia a noua linie avem "`name <- getLine`" Aceasta ar fi trebuit, în mod normal, să fie scrisă "`name = getLine`", însă am utilizat

săgeata în loc de egalitate pentru a arăta că `getLine` nu este efectiv o funcție (ea este o *acțiune*) şi că poate returna diferite valori. Această comandă înseamnă "rulează acțiunea `getLine` şi păstrează rezultatul în `name`".

Ultima linie construieşte un şir utilizând ceea ce s-a citit în linia precedentă şi apoi îl afişează pe ecran.

Un alt exemplu al unei funcții care nu este funcție în mod real este una care returnează o valoare aleatoare. În acest context, una din funcțiile care fac acest lucru este `randomRio`. Putem scrie, cu ajutorul acestei funcții, un program "ghiceşte numărul". Vom scrie într-un fişier "`Guess.hs`" următorul cod:

```
module Main
     where

import IO
import Random

main = do
   hSetBuffering stdin LineBuffering
   num <- randomRIO (1::Int, 100)
   putStrLn "I'm thinking of a number between 1 and
     100"
   doGuessing num

doGuessing num = do
   putStrLn "Enter your guess:"
   guess <- getLine
   let guessNum = read guess
```

```
if guessNum < num
  then do putStrLn "Too low!"
          doGuessing num
  else if read guess > num
          then do putStrLn "Too high!"
            doGuessing num
          else do putStrLn "You Win!"
```

Sa examinăm puțin aceste linii de cod. Linia a 5-a spune compilatorului că se va importa "`Random`", adică vom utiliza una dintre funcțiile "random" (acestea nu sunt funcții în Prelude). În prima linie a funcției `main`, se cere un număr oarecare între 1 şi 100. Trebuie să scriem `::Int` pentru a spune compilatorului că utilizăm numai numere întregi. Pe linia următoare se spune utilizatorului ceea ce se întâmplă, după care, în ultima linie a funcției `main`, cerem compilatorului să execute comanda `doGuessing`.

Funcția `doGuessing` ia numărul pe care utilizatorul încearcă să-l ghicească ca argument. Mai întâi cere utilizatorului să ghicească şi apoi acceptă ceea ce dă utilizatorul în formă `String`. Testul **`if`** verifică mai întâi dacă numărul introdus este prea mic. Întrucât `guess` este de tip string iar `num` este de tip integer, mai întâi trebuie să efectuăm o conversie, adică să convertim `guess` la tipul integer prin citirea lui. Mai mult, deoarece "`read guess`" este o funcție pură (şi nu e o acțiune de tip IO), nu se utilizează notația `<-`; pur şi simplu se leagă valoarea la `guessNum`.

Am vorbit ceva mai sus despre tipuri. Este destul de uşor de înțeles despre ce este vorba în acest caz. Oricum, capitolul următor ne va introduce în studiul tipurilor din Haskell.

În rest, liniile de cod sunt clare. Trebuie totuşi să mai observăm că odată obținută valoarea de ghicit, se spune că s-a câştigat şi se iese din program, fără a fi nevoie de o afirmație de tip `return ( )`.

Fişierul se poate sau compila sau încărca direct în interpretor. Se pot obține ceva de genul:

```
Main> main
I'm thinking of a number between 1 and 100
Enter your guess:
50
Too low!
Enter your guess:
75
Too low!
Enter your guess:
85
Too high!
Enter your guess:
80
Too high!
Enter your guess:
78
Too low!
Enter your guess:
79
You Win!
```

În acest momentar trebui să aveți o înțelegere bună a modului de scriere a unor funcții simple, compilarea lor, testarea funcțiilor şi programelor într-un mediu interactiv şi manipularea listelor.

Exerciții

1. Încercăm să scriem un program care cere în mod repetat utilizatorului să introducă cuvinte. Dacă utilizatorul introduce în orice moment un cuvânt vid, programul afişează tot ceea ce s-a introdus până în acel moment şi apoi iese. Se presupune că următoarele linii de cod ar îndeplini această cerință:

```
askForWords = do
       putStrLn "Please enter a word:"
       word <- getLine
       if word == ""
          then return []
          else return (word : askForWords)
```

Din păcate acest program va da eroare. Unde este greşeala? Corectați programul.

2. Scrieți un program care cere în mod repetat utilizatorului numere până când utilizatorull introduce un 0, moment în care programul va afişa suma şi produsul tuturor numerelor şi, pentru fiecare număr în parte, factorialul său. În fapt, o sesiune ar trebui să arate astfel:

```
Give me a number (or 0 to stop):
5
Give me a number (or 0 to stop):
8
Give me a number (or 0 to stop):
2
```

```
Give me a number (or 0 to stop):
0
The sum is 15
The product is 80
5 factorial is 120
8 factorial is 40320
2 factorial is 2
```

Capitolul IV

Tipuri de bază

Limbajul Haskell utilizează un sistem *static de verificare a tipurilor*. Acest lucru înseamnă că oricărei expresii din Haskell i se asociază un tip. De exemplu, `'a'` este de tip `Char`. În aceste condiții, dacă aveți o funcție care se aşteaptă la un argument de un anumit tip şi i se dă un argument de un tip greşit, nu vom putea să compilăm programul. Acest lucru reduce, în cazul Hakell-ului, în mod drastic numărul bug-urilor care ar putea apărea.

Limbajul Haskell utilizează şi un sistem *infererențial de tipuri*, adică nu este nevoie să specificăm de fiecare dată tipul unei expresii. Ca şi comparație, în C, atunci când se defineşte o variabilă, trebuie să se specifice şi tipul său. În Haskell, tipul variabilei se subînțelege din context (totuşi, pentru o mai bună înțelegere a liniilor de cod, este bine să scriem în mod explicit tipul unei expresii - ajută la corectarea programelor).

Interpretoarele GHCi şi Hugs permit aflarea tipului unei expresii. De exemplu:

```
Prelude> :t 'c'
'c' :: Char
```

Acesta ne spune că expresia `'c'` are tipul `Char`, semnul "`::`" utilizându-se în Haskell pentru specificarea tipurilor

1. Tipuri simple

Există tipuri simple predefinite în Haskell, cum ar fi `Int` (pentru numere întregi), `Double` (pentru numere cu virgulă mobilă), `Char` (pentru caractere simple), `String` (pentru cuvinte) ş.a.m.d. Exemple:

```
Prelude> :t "Hello"
"Hello" :: String
Prelude> :t 'a' == 'b'
'a' == 'b' :: Bool
Prelude> :t 'a' == "a"
ERROR - Type error in application
*** Expression : 'a' == "a"
*** Term : 'a'
*** Type : Char
*** Does not match : [Char]
```

Prima linie a erorii ne spune care este expresia în care apare eroarea de tip. A doua linie ne indică partea din expresie care este prost tipizată. A treia linie ne spune tipul implicit al expresiei, iar cea de a patra că tipul `Char` nu este de tip `[Char]` (adică listă de caractere sau şir de caractere în Haskell).

Observație. Procesul de control a tipurilor încearcă să găsească un tip chiar şi atunci când o expresie este greşit introdusă. În exemplul dat operatorul de egalitate necesită ca ambele argumente să aibă acelaşi tip.

După cum am mai menționat, putem specifica în mod explicit tipul unei expresii utilizând operatorul " `::` ". În cazul numerelor, putem face declarații duble. De exemplu:

```
Prelude> :t 5 :: Int
5 :: Int
Prelude> :t 5 :: Double
```

```
5 :: Double
```

În acest fel, numărul 5 poate fi privit sau ca `Int` sau ca `Double`. Ce se întâmplă când nu se specifică tipul:

```
Prelude> :t 5
5 :: Num a => a
```

Nu este chiar ceea ce ne-am fi aşteptat. Acest lucru înseamnă că dacă un anumit tip `a` este o instanță a clasei `Num`, atunci tipul expresiei 5 poate fi `a`. Vom vorbi în secțiunile următoare despre **clase**.

Să prezentăm, pe scurt, câteva dintre tipurile predefinite:

Numere întregi: tipurile `Int` şi `Integer`:

Tipul `Int` conține numerele întregi dintre -2^{31} şi $2^{31}-1$

Tipul `Integer` este, în esență, acelaşi ca şi tipul `Int`, exceptând faptul că, în acest tip, se poate reprezenta orice număr întreg. Este bine, însă, să fie utilizat numai în cazurile în care ştim sigur că se depăşeşte intervalul lui `Int`

Numerele reale: tipurile `Float` şi `Double`:

Tipul `Float` conține numerele reale. Ele se pot scrie în două moduri

- notație zecimală: `265.65`
- notație stiințifică: `2.6565e2`

Observație. Calculatoarele reprezintă numerele reale prin aproximare cu numere raționale, motiv pentru care trebuie să se evite testarea egalității! În acest sens, nu încercați să verificați faptul că $(\sqrt{2})^2 = 2$.

```
Prelude> sqrt 2 ^ 2 == 2
False
```

Tipul `Double` este, în esență, acelaşi ca şi tipul `Float`, exceptând faptul că el reprezintă numerele cu o mai bună precizie.

Tipul boolean: `Bool`

Tipul `Bool` conține numai două valori `True` şi `False`

- este utilizat pentru reprezentarea rezultatelor unor decizii.
- să notăm faptul că şi `True` şi `False` se scriu cu literă mare.

Tipul Char

Tipul Char conține caracterele, şi anume:

- caracterele alfabetice, `'a' .. 'z'` şi `'A' .. 'Z'`
- caracterele numerice, `'0' .. '9'`
- caracterele simbolice, cum ar fi. `'!', '(', ' ', '+'`
- caracterele speciale,cum ar fi `'\n'` (linie nouă), `'\t'` (tab)

Ghilimelele între care se află un caracter sunt cele **simple (apostrof)** care se găsesc pe tastatură pe aceeaşi tastă cu cele duble, “.

Tipul String

Tipul `String` conține valori constituite dintr-un şir de caractere din `Char`. Ele se dau între ghilimele duble. De exemplu:

- `"Romania"` este un string alfabetic
- `"42"` un string de cifre
- `"99(!) mantie gri"` este un string amestecat
- `"  "` un string de trei spații
- `""` string-ul vid

Observații. Atragem atenția asupra diferenții dintre:

`'q'` este un `Char`

`"q"` este un `String` care conține un caracter

`""` un String vid

`" "` un String care coține un spațiu (caracter)

`"  "` un String care coține două spații (caractere)

`42` este un `Int`

`"42"` este un `String` care coține două cifre privite ca fiind caractere

`7` este un `Int`

`'7'` este un `Char`

`"7"` este un `String` care coține o cifră privită ca şi caracter

Exercițiu.

Încercați să determinați tipul expresiilor următoare (dacă există), fără a utiliza Haskell-ul. Verificați. Care dintre expresii dau eroare de tip?

```
1. 'h':'e':'l':'l':'o':[ ]
2. [5,'a']
3. (5,'a')
4. (5::Int) + 10
5. (5::Int) + (10::Double)
```

2. Tipuri polimorfe

Haskell utilizează in sistem de tipuri polimorf. Acest lucru înseamnă că putem avea tipuri variabile. De exemplu, o funcție cum este `tail` nu ține seamă de felul elementelor dintr-o listă:

```
Prelude> tail [5,6,7,8,9]
```

```
[6,7,8,9]
Prelude> tail "hello"
"ello"
Prelude> tail ["the","man","is","happy"]
["man","is","happy"]
```

Acest lucru este posibil datorită faptului că `tail` are tip polimorf : `[•]->[•]`. În acest fel, se poate lua ca argument orice listă şi să se returneze ca valoare o listă de acelaşi tip.

În mod asemănător putem explica şi tipul funcției `fst`,

```
Prelude> :t fst
forall a b . (a,b) -> a
```

În acest caz, GHCi a explicat în mod amănunțit tipul, adică a menționat faptul că pentru orice tip `a` şi `b`, `fst` este o funcție de la `(a,b)` la `a`.

Exercițiu. Încercați să determinați tipul expresiilor următoare (dacă există), fără a utiliza Haskell-ul. Verificați.

1. `snd`
2. `head`
3. `null`
4. `head . tail`
5. `head . head`

3. Clase de tipuri

Motivație.

În multe limbaje de programare există un sistem de "overloading", adică o funcție poate fi scrisă ca luând drept parametru diferite tipuri. Un exemplu în acest sens este funcția egalitate, care face o comparație între două lucruri de acelaşi tip •, • putând avea diferite tipuri (`Int`, `Double`, `Char`, etc). Vom spune că • este *variabilă tip*, şi vom nota variabilele tip cu litere din alfabetul grec.

Din păcate, acest lucru prezintă unele probleme atunci când se face verificarea statică a tipurilor, întrucât verificatorul de tip nu ştie, atunci când vom defini o operație, pentru ce tipuri va fi definită. În Haskell există soluția sistemului de *Clase de tipuri*.

Testarea egalității

Vrem să putem defini o funcție = = (operatorul egalitate) care ia doi parametri, ambii de acelaşi tip, şi returnează un parametru boolean. Această funcție nu poate fi definită pentru orice tip; atunci va trebui să asociem această funcție cu o anumită clasă de tipuri pe care o vom numi **`Eq`**.

În cazul în care un anumit tip • aparține unei anumite clase de tipuri (adică toate funcțiile asociate acelei clase conțin implementări pentru •) vom spune că • este o *instanță* a acelei clase. De exemplu, **`Int`** este o instanță a clasei **`Eq`**.

Clasa `Num`.

Haskell - ul are încărcate şi constante numerice ca şi funcţii. Acest lucru este făcut în aşa fel încât, atunci când vom scrie 3, de exemplu, compilatorul este liber să aleagă tipul lui 3 (întreg sau cu virgulă mobilă, de exemplu). Se defineşte clasa **`Num`** ca fiind clasa care conţine toate numerele şi principalele operaţii peste ele. Tipurile numerice de bază sunt, aşadar, instanţe ale clasei **`Num`**. Bineînţeles `Int` este instanţă şi a acestei clase

Clasa `Show`.

Tipurile din clasa **`Show`** au funcţii care convertesc valori ale acelor tipuri într-un şir. Această funcţie este numită `show` . De exemplu, când aplicăm `show` numărului întreg `4` se obţine şirul `"4"`. Când se aplică unui caracter `'a'` se obţine un şir de trei caractere `"'a'"` (semnele apostrof sunt şi ele privite ca şi caractere).

```
Prelude> show 5
"5"
Prelude> show 'a'
"'a'"
Prelude> show "Hello World"
"\"Hello World\""
```

"Backslash"-urile apar deoarece ghilimelele interioare sunt "scăpate", adică ele sunt parte a şirului şi nu parte a interpretorului care afişează valoarea.

Vom reveni în capitolul următor asupra claselor de tipuri, dând şi o listă a tuturor claselor predefinite în `Prelude`.

4.Tipuri funcționale

În Haskell, funcțiile sunt *valori de primă clasă*, adică aşa cum `1` sau `'a'` sunt valori care au un tip, şi funcțiile `square` sau `++` au un tip. Înainte de a merge direct la funcții, să ne ocupăm puțin de ceva ce am studiat deja, adică lambda calcul.

Lambda Calcul

Să reamintim că lambda calculul era, în mare, o modalitate de a scrie funcțiile. De exemplu, când doream să evaluăm funcția $(\lambda x.x*x)5$, se "scoate" lambda şi se înlocuieşte fiecare apariție a lui *x* cu 5.

În fapt, Haskell-ul este bazat pe o extensie a lambda calculului. Expresia de mai înainte se scrie direct în Haskell înlocuind λ cu `\` şi . cu săgeată; bine-înțeles, în Haskell trebuie să dăm nume funcțiilor:

```
square = \x -> x*x
f = \x y -> 2*x + y
```

Putem să evaluăm şi lambda expresii într-o celulă interactivă:

```
Prelude> (\x -> x*x) 5
25
Prelude> (\x y -> 2*x + y) 5 4
14
```

După cum ați observat în al doilea exemplu, a trebuit să dăm abstractizării două argumente, una corespunzătoare lui `x` şi una lui `y`.

Tipuri de ordin înalt

"Tipuri de ordin înalt" (*higher order types*) constituie numele dat funcțiilor. Tipul dat funcțiilor mimează reprezentarea dată funcțiilor în lambda calcul.

De exemplu, definiția funcției square am dat-o prin $\lambda x.x*x$. Pentru a da şi tipul acesteia, trebuie mai întâi să ne întrebăm asupra tipului variabilei x. Să presupunem că ne-am hatărât ca x să fie `Int`. Apoi ne uităm la funcția noastră şi observăm că ea se defineşte pe întregi şi produce valori întregi. În acest mod vom spune că `square` este de tip `Int -> Int`.

Cea de a doua funcție prin care am exemplificat scrierea cu "lambda" poate fi văzută ca fiind de tipul `Int -> (Int -> Int)`.

Observații.

1. Parantezele puse nu sunt necesare, după cum vă puteți reaminti din lambda calcul. Adică dacă aveți $\alpha \to \beta \to \gamma$, se subînțelege că $\beta \to \gamma$ este grupată. Dacă dorim să avem tipul care are $\alpha \to \beta$ grupat, trebuie să puneți paranteze.

2. Numerele, după cum am văzut deja, nu sunt chiar de tip `Int`, ele sunt privite ca fiind de tipul `Num a => a`

Se poate afla foarte uşor tipul funcțiilor predefinite din Prelude utilizând "`:t`",

```
Prelude> :t head
head :: [a] -> a
Prelude> :t tail
tail :: [a] -> [a]
Prelude> :t null
null :: [a] -> Bool
Prelude> :t fst
fst :: (a,b) -> a
```

```
Prelude> :t snd
snd :: (a,b) -> b
```

Cum se interpretează aceste lucruri: de exemplu, "`head`" este o funcție care se aplică unei liste care conține valori de tipul "`a`" şi ne dă o valoare de tipul "`a`".

De asemenea, putem afla tipul operatorilor cum ar fi `+` , `*` sau `:`. În orice caz, trebuie să îi punem între paranteze atunci când vrem să le aflăm tipul, deoarece sunt funcții infix:

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (*)
(*) :: Num a => a -> a -> a
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

Tipul funcțiilor `+` şi `*` este acelaşi, adică ele sunt funcții care, pentru un anumit tip `a` care este instanță a clasei `Num`, ia o valoare de tip `a` şi produce o altă funcție care se aplică unei valori de tip `a` şi returnează o valoare de tip `a`. Am fi putut spune că +, de exemplu, se aplică unei perechi de valori de tip `a` şi returnează o valoare de tip `a`, însă ar fi imprecisă o astfel de exprimare.

Tipul IO

Am putea fi tentați să încercăm să vedem tipul unor funcții cum ar fi `putStrLn` sau `readFile`:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ( )
Prelude> :t readFile
```

`readFile :: FilePath -> IO String`

Ce înseamnă IO? Este modul de bază utilizat de Haskell pentru reprezentarea unor, să le spunem, *pseudofuncții* (adică funcții care nu sunt în mod real funcții). Ele se numesc "*IO Actions*". Următoarea întrebare care ne-o putem pune ar fi: "şi cum reuşim să stăpânim IO?".

Din păcate, nu putem să o stăpânim în mod direct, în sensul că nu putem scrie funcții având tipul `IO String -> String`. Singura posibilitate de a utiliza un tip IO este să fie combinat cu alte funcții utilizând, de exemplu, notația **`do`.**

De exemplu, dacă se citeşte un fişier utilizând `readFile`, se presupune că dorim să facem ceva cu şirul care-l returnează (altfel, de ce am citi fişierul mai întâi ?). Să presupunem că avem o funcție `f` care transformă un `String` într-un `Int`. Nu putem aplica în mod direct `f` rezultatului lui `readFile` întrucât intrarea lui `f` este `String` pe când ieşirea lui `readFile` este `IOString`, deci nu se potrivesc. Însă le putem combina astfel:

```
main = do
  s <- readFile "somefile"
  let i = f s
  putStrLn (show i)
```

Aici am utilizat convenția de săgeată "scoate şirul de sub acțiunea IO" şi apoi am aplicat `f` şirului. După aceasta putem, de exemplu, să scriem rezultatul pe ecran.

Atenție: `let` nu are un `in` corespunzător, şi asta pentru că suntem într-un bloc `do`. De asemenea, nu vom scrie `i <- f s`, deoarece `f` este o funcție "normală", şi nu o acțiune IO.

Declararea explicită a tipurilor

De cele mai multe ori este de dorit să se specifice în mod explicit tipurile unor elemente sau funcții, şi asta din unul (sau mai multe) dintre motivele următoare:

- claritate
- viteză
- corectare

Există părerea că este foarte bine ca măcar toate funcțiile de top să aibă tipul specificat. În acest caz, atunci când se obțin erori de compilare pe care nu le putem înțelege, este mai uşor de observat unde ar fi eroarea.

Declararea tipului unei funcții se face separat de definiția funcției. Declararea explicită a tipului se numeşte *semnătura tipului*. Ca exemplu, putem da în mod explicit tipul funcției `square` ca în liniile de cod următoare:

```
square :: Num a => a -> a
square x = x*x
```

Aceste două linii nu trebuie să fie neapărat una după cealaltă. După definiție, este clar că se poate aplica `square` oricărui tip care este instanță a clasei `Num`: `Int`, `Double`, etc...

Dacă totuşi suntem siguri că vom aplica funcția `square` numai pentru valori întregi, se poate rafina tipul ei:

```
square :: Int -> Int
square x = x*x
```

Ce avantaj avem de aici: compilatorul nu trebuie să genereze un cod general după cum era specificat în prima declarație; acest lucru conduce la posibilitatea generării unui cod mai rapid.

În cazul în care extensiile GHCi sunt deschise (adică "`-fglasgow-exts`") se poate adăuga semnătura tipului şi expresiilor, nu numai funcțiilor. De exemplu, am putea scrie:

```
square (x :: Int) = x*x
```

care spune compilatorului că `x` este un întreg; acest lucru lasă la latitudinea compilatorului să infereze tipul restului expresiei. Încercați să deduceți tipul funcției square din exemplul prezentat şi verificați-l în Prelude, scriind:

```
Prelude> :t (\(x :: Int) -> x*x)
```

Nu am dat şi răspunsul GHCi -ului.

Argumente funcționale

În capitolul precedent, am văzut cum am aplicat unele funcții (`map`, `filter`, `foldr`, `foldl`, ...) altor funcții.

Să ne oprim pentru început la funcția `map`. Ea trebuie să ia o listă de elemente şi să producă o altă listă de elemente. Cele două liste nu sunt neapărat de acelaşi tip. Astfel, `map` are ca argument o listă de tip `[a]` şi produce o listă de tip `[b]`. Cum realizează acest lucru? Ea utilizează o altă funcție, care este furnizată de utilizator, pentru a face conversia. Pentru a converti tipul `a` în tipul `b`, această funcție trebuie să fie de tipul `a->b`. Din

acest motiv, tipul funcției `map` trebuie să fie `(a->b)->[a]->[b]`. Putem verifica acest lucru în interpretorul nostru.

În mod asemănător se poate analiza şi funcția `filter` şi să observăm că ea este de tipul `(a->Bool)->[a]->[a]`.

După cum am prezentat funcția `foldl`, am fi tentați să spunem că tipul ei este `(a -> a  -> a) -> a ->[a] ->a`, adică avem o funcție care combină două funcții într-o altă funcție, are o valoare inițială de tip `a`, o listă de `a` - uri, şi produce în final o valoare de tip `a`.

De fapt, `foldl` are un tip mai general, `(a -> b  -> a) -> a ->[b] ->a`, adică este o funcție care transformă un tip `a` şi un `b` în tipul `a`, o valoare inițială de tipul `a` şi o listă de valori de tip `b`, şi produce, în final, o valoare de tipul `a`.

Pentru a vedea toate astea, putem scrie o funcție `count` care numără câți membri dintr-o listă satisfac o anumită condiție. Am fi putut utiliza, de exemplu, `filter` şi `length` pentru asta, însă o putem face şi prin utilizarea lui `foldr`:

```
module Count
     where

import Char

count1 p l = length (filter p l)
count2 p l = foldr (\x c -> if p x then c+1 else c)
0 l
```

Funcționarea lui `count1` este simplă. Ea filtrează după predicatul `p` lista `l` şi apoi ia lungimea listei obținute.

Pe de altă parte, `count2` utilizează valoarea inițială (care este întreagă) pentru a "ține" numărătoarea curentă. Pentru fiecare element din lista `l`, îi aplică lambda expresia din paranteza. Ea are două argumente, `c` care ține numărătoarea curentă şi `x` care este elementul curent din listă. Ea verifică dacă predicatul `p` are loc pentru `x`. Dacă da, ea măreşte pe `c` la `c+1`, iar dacă nu, ea returnează vechea valoare a lui `c`.

Exerciții. Încercați să determinați tipul expresiilor următoare (dacă există), fără a utiliza Haskell-ul. Verificați.

```
1.   x -> [x]
2.   x y z -> (x,y:z:[])
3.   x -> x + 5
4.   x -> "hello, world"
5.   x -> x 'a'
6.   x -> x x
7.   x -> x + x
```

5. Tipuri de date utilizator

Uplurile şi listele sunt modalități obişnuite pentru definirea unor structuri de date. În orice caz, este de dorit să putem să ne definim tipurile noastre de date şi funcțiile peste ele. Aşa numitele *tipuri de date* se definesc utilizând cuvântul cheie **data**. Pentru început (în acest paragraf) ne vom limita la a da un exemplu pentru definirea perechilor, cititorului revenindu-i "misiunea" de a rezolva exercițiile de la sfârşitul acestui paragraf pentru definirea altor tipuri simple de date. În ultimul capitol vom reveni la acest subiect.

Perechi.

Definiția unei perechi de elemente (la fel ca şi cea standard, tipul `Pair`) ar putea fi :

```
data Pair a b = Pair a b
```

Să comentăm această linie de cod. Când declarăm "`data`" se înțelege că definim un tip de date. După aceasta dăm numele tipului de date. `a` -ul şi `b` -ul care urmează sunt tipurile parametrilor. După semnul egal, specificăm *constructorii* acestui tip de date. În cazul nostru avem numai un singur constructor, "`Pair`" (el nu trebuie să aibă neapărat acelaşi nume ca şi tipul). După constructor scriem din nou "`a b`" care ne arată că pentru construcția lui `Pair` avem nevoie de două valori, una de tip `a` şi una de tip `b`. Această definiție a introdus o funcție `Pair :: a -> b -> Pair a b`, pe care o utilizăm să construim perechi. Dacă scriem acest cod într-un fişier şi apoi îl încărcăm, putem vedea cum sunt construite:

```
Datatypes> :t Pair
Pair :: a -> b -> Pair a b
Datatypes> :t Pair 'a'
Pair 'a' :: a -> Pair Char a
Datatypes> :t Pair 'a' "Hello"
Pair 'a' "Hello" :: Pair Char [Char]
```

Astfel, dând lui `Pair` două valori, am construit o valoare a tipului `Pair`. Putem scrie funcții care utilizează perechi:

```
pairFst (Pair x y) = x
pairSnd (Pair x y) = y
```

În aceste linii de cod am utilizat posibilitățile Haskell-ului de *potrivire al şablonului* (pattern matching – vezi şi capitol V), pentru a extrage valori dintr-o pereche. În definirea lui `pairFst` luăm o întreagă pereche şi extragem primul element şi, analog, pentru `pairSnd`.

Exerciții.

1. Scrieți un tip de date pentru `Triple`, un tip ce conține trei elemente, fiecare de tip diferit. Scrieți funcțiile `tripleFst`, `tripleSnd` şi `tripleThr` care extrag respectiv primul, al doilea şi al treilea element.
2. Scrieți un tip de date pentru `Quadruple` un tip ce conține patru elemente: primele două elemente trebuie să fie de acelaşi tip şi ultimele două elemente trebuie să fie de acelaşi tip. Scrieți funcțiile `firstTwo`, care extrage o listă cu primele două elemente, şi o funcție `lastTwo`, care extrage o listă cu ultimele două elemente. Scrieți semnătura de tip a acestor funcții.

Capitolul V

Alte proprietăți şi tehnici utile

În acest ultim capitol al introducerii în programarea funcțională dorim să fixăm o parte din cunoştințele dobândite până acum, îmbunătățind şi "oferta" posibilităților pe care le oferă Haskell.

1. Module

În Haskell, subcomponentele unui program mai mare sunt divizate in **module**. Fiecare modul este situat în propriul fişier, denumirea lui trebuie să coincidă cu denumirea fişierului (fără extensia "hs" bineînțeles), şi asta întrucât există posibilitatea să mai folosiți acel modul şi într-un alt program. Această modularitate este foarte asemănătoare cu metodele utilizate în programarea orientată obiect.

Să presupunem că dorim sa desenăm o parabolă cu ajutorul unui program scris în Haskell. Întrucât este posibil să dorim să scriem şi un program care să deseneze şi alte figuri geometrice, definirea punctelor, liniilor, culorilor, ş.a.m.d., ar trebui să fie comună. Suntem conduşi la ideea considerării scrierii, mai întâi, a unui modul separat, pe care-l vom numi `Graphics`. Ce ar trebui să conțină un astfel de modul ? În primul rând ar trebui să definească punctele şi liniile, să "ştie" să rotească figurile, să le mărească/micşoreze, să facă translații (adică să mute) figurile. După care, pentru a utiliza modulul definit în cadrul programului nostru, vom folosi aşa numitele **importuri.**

Există câteva chestiuni mai sensibile în sistemul de import al modulelor, dar atâta timp cât nu vom încerca cine ştie ştie ce forţări, nu sunt probleme. Chemarea unui modul în cadrul unui alt modul, (l-am numit în cazul nostru `Geometry`) se face simplu:

```
module Geometry
     where
import Graphics
.....
```

Am văzut deja, în exemplele prezentate când am dorit să creăm programe interactive, cum se face un astfel de import (importând modulul IO sau modulul `Random`).

2. Aplicarea parțială

Prin aplicare parțiala vom înțelege este cazurile în care, considerând o funcție de *n* variabile, trebuie să o aplicăm numai într-un număr mai mic de valori (decât *n*). Un exemplu în acest sens:

```
map (+1) [0,2,4]
```

Partea `(+1)` este o aplicare parțiala a funcției `+` şi aceasta deoarece operatorul `+` este definit ca având două argumente, noi dându-i numai unul singur.

Aplicarea parțială este foarte răspândită în definițiile date unor funcții, fiind cunoscută în teorie sub numele de η- conversie (vezi capitolul I).

Un alt exemplu:

Să presupunem că dorim să scriem o funcție `ucaseString` care converteşte un string de caractere alfabetice într-un string scris numai cu caractere alfabetice mari (majuscule). Putem scrie astfel:

```
ucaseString s = map toUpper s
```

În acest moment nu este vorba de nici o aplicare parțială. În schimb, utilizând o η- conversie, putem scrie:

```
ucaseString = map toUpper
```

iar de data aceasta este o aplicare parțială a lui `map`.

3. Potrivirea şabloanelor sau "Pattern Matching"

Potrivirea şabloanelor este una dintre cele mai puternice proprietăți caracteristice a majorității limbajelor de programare funcționale şi, în particular, ale Haskell - ului. Este destul de des folosită în expresia **case**, pe care am vazut-o deja în capitolele precedente (când am vorbit despre diverse modalități de definire a funcțiilor).

Pentru definirea unor funcții pe perechi, de exemplu, se utilizează *şabloanele*. De exemplu, să considerăm o funcție de adunare a unei perechi de numere întregi:

```
addPair :: (Int, Int) -> Int
addPair (x,y) = x + y
```

Şablonul `(x,y)` este "potrivit" oricărei perechi, şi defineşte faptul ca `x` este prima componentă a perechii şi `y` este a doua componentă. De exemplu: `(0.0, a)` se potriveşte oricărei perechi care are primul element valoarea `Float 0.0` iar cel de-al doilea element legat de tipul `a`.

Putem să facem potriviri numai după tipuri de date şi **nu** putem face potriviri pentru funcții. De exemplu, urmatoarea definiție nu este validă:

```
f x = x + 1
g (f x) = x
```

Chiar dacă intenția noastră privind g este clară, adică faptul că `g x = x - 1`, compilatorul nu are de unde să ştie că funcția f are o inversă, deci nu poate face astfel de potriviri.

Atenție:

- şabloanele nu sunt expresii arbitrare
- variabilele nu pot fi repetate în partea stângă a unei definiții
- potrivirea şabloanelor este abordată în manieră **strict secvențială**
  - de la stânga la dreapta într-o definiție
  - de sus în jos într-o secvență de clauze ale definiției.
- ori de câte ori potrivirea şablonului *nu reuşeşte*, ecuația respectivă este respinsă şi se încearcă următoarea.

Exemplu:

Definim o funcție

```
sumPairs :: [(Int,Int)] -> [Int]
```

care sumează elementele fiecărei perechi dintr-o listă de perechi, şi ne oferă o listă cu sumele perechilor. Cu alte cuvinte,
`sumPairs [(1, 9),(37, 8)]` trebuie să fie lista `[10, 45]`
Cazul inițial este standard,

```
sumPairs [] = []
```

Putem alege mai multe şabloane pentru partea inductivă. În general este bine să se aleagă *şablonul cel mai precis*.

Care ar fi alternativele:

```
sumPairs1 (x:xs) = (fst x + snd x) : sumPairs1 xs
```

şablonul `(x:xs)` se potriveşte oricărei liste nevide. Componentele sunt extrase utilizând funcțiile de perechi `fst` şi `snd`.
Cea de a doua alternativă se realizează utilizând o clauză `where` :

```
sumPairs2 (x:xs) = (m + n) : sumPairs2 xs
                    where (m, n) = x
```

Şi aici şablonul `(x:xs)` se potriveşte oricărei liste nevide, însă componentele capului de listă se obțin prin potrivirea şabloanelor din definiția clauzei `where`, `(m, n) = x.`

Ultima alternativă pe care o prezentăm este bazată pe alegerea celui mai precis şablon, ceea ce ne conduce la cea mai simplă şi mai clară definiție:

```
sumPairs ((m,n):xs) = m+n : sumPairs xs
```

Atragem atenția că următoarea linie de cod este **ilegală:**

```
find code ((code, name, price) : rest)= (name, price)
```

Acest lucru se datorează faptului că variabilele se repetă în partea stângă a definiției.

Soluția acestei probleme este ca în loc să utilizăm potrivirea şabloanelor pentru testarea egalității să utilizăm **gărzile**.

```
find barcode ((code, name, price) : rest) = (name, price)
           | barcode == code
           | . . .
```

4. Gărzi

Gărzile pot fi interpretate ca o extensie a facilităților de potrivire a şabloanelor. Ele se utilizează atunci când dorim să definim o funcție pe bază de alternative, depinzând de anumite condiții care pot fi adevărate sau false.

Gărzile apar după ce au fost date argumentele funcției, însă înaintea semnului egal; ele încep cu o bara verticala.

De exemplu, să scriem o funcție care compară două numere întregi, rezultatul fiind dat printr-un mesaj care ne va spune care este mai mare:

```
compare x y :: Int -> Int -> String
compare x y
     | x > y = "The first is greater"
     | x < y = "The second is greater"
     | otherwise = "They are equal"
```

Să observăm că a apărut o gardă specială, `otherwise`, care este definită (chiar în Prelude) ca fiind întotdeauna adevărată. Ea este utilizată la *sfârşitul* unei secvențe de alternative, pentru a putea "prinde" toate alternativele care nu au fost luate în considerare. Scrieți aceste linii de cod într-un fişier `Guards.hs`, bineînțeles dupa ce ați pus numele modulului, încărcați în GHCi şi verificați funcționarea :

```
Guards> compare 5 3
"The first is greater"
Guards> compare 2 5
"The second is greater"
Guards> compare 3 3
"They are equal"
```

Un alt exemplu al definiției cu gărzi pentru funcția factorial:

```
fact' :: Integer -> Integer
fact' n
     | n==0      = 1
     | otherwise = n * fact' (n-1)
```

care, totuşi, nu este atât de clară cu cea dată, prin şabloane, în capitolul precedent.

Să mai observăm că, în definiția unei funcții, pot apărea, pe lângă gărzi, şi definiții locale. În capitolul precedent am dat o funcție roots, despre care nu am spus nimic ce va face în cazul în care rădăcinile ecuației de grad 2 nu sunt reale. Să revenim la ea şi să o completăm:

```
roots a b c
```

```
  |det >=0   = ((-b + det)/(2*a),(-b - det)/(2*a))
  |otherwise = error "No real roots"
     where det = sqrt(b*b-4*a*c)
```

5. Clase

Pentru a declara faptul că un anumit tip de date este o instanţă a unei anumite clase, trebuie să-i dăm o *declaraţie de instanţă*. Marea majoritate a claselor conţin o aşa numită "definiţie completă minimală". Această definiţie minimală conţine obligatoriu funcţiile care trebuie să poată fi implementate în oricare dintre tipurile care sunt instanţe ale acelei clase. Odată ce se reuşeşte această implementare, tipul de date respectiv poate fi declarat ca instanţă a acelei clase. Vom încerca să prezentăm clasele predefinite în `Prelude` împreună cu ierarhia lor.

Pentru aceasta, să definim un tip de date propriu, cum ar fi tipul culoare, şi să vedem dacă este instanţă a unei clase din `Prelude`.

```
data Color
     = Red
     | Orange
     | Yellow
     | Green
     | Blue
     | Purple
     | White
     | Black
```

Dacă dorim ca acest tip de date să poată fi utilizat în modulul Graphics de la începutul capitolului, trebuie să scriem şi o funcție de conversie de la `Color` la un triplu `RGB`.

```
colorToRGB Red = (255,0,0)
colorToRGB Orange = (255,128,0)
colorToRGB Yellow = (255,255,0)
colorToRGB Green = (0,255,0)
colorToRGB Blue = (0,0,255)
colorToRGB Purple = (255,0,255)
colorToRGB White = (255,255,255)
colorToRGB Black = (0,0,0)
```

Dacă dorim ca utilizatorul să poată să-şi definească culori, putem să adăugăm la definiția tipului de date linia:

```
| Custom Int Int Int -- R G B components
```

iar la funcția de conversie linia:

```
colorToRGB (Custom r g b) = (r,g,b)
```

Clasa `Eq`

Clasa `Eq` are două funcții, cu următoarele semnături:

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

Prima dintre aceste semnături de tip indică faptul că funcția = = este o funcție care are argumentele de tip a care face parte din clasa Eq şi produce o valoare de tip Bool. Semnătura de tip pentru funcția /= (diferit) este identică.

O definiție completă minimală a clasei Eq cere ca una dintre funcțiile menționate să poată fi definită pentru tipul de data ce ar fi membru al clasei (în mod evident, daca una dintre funcții este definită cealaltă se obține prin negare). Aceste declaratii trebuie sa fie furnizate in interiorul instanței de declarare a tipului de date avut în vedere.

Revenind la tipul de date Color, putem să-l vedem ca instanță a clasei Eq astfel:

```
instance Eq Color where
     Red == Red = True
     Orange == Orange = True
     Yellow == Yellow = True
     Green == Green = True
     Blue == Blue = True
     Purple == Purple = True
     White == White = True
     Black == Black = True
     (Custom r g b) == (Custom r' g' b') =
          r == r' && g == g' && b == b'
     _ == _ = False
```

Prima linie din exemplu începe cu cuvântul cheie **instance** care spune compilatorului că vom face o instanță de declarație. Aceasta specifică clasa, care este Eq, apoi tipul de date, Color, care va trebui să fie o instanță a acestei clase. După cuvântul cheie **where** este declararea egalității.

Clasa `Show`

Clasa Show conține tipurile care au o reprezentare printabilă, adică toate elementele unui tip din această clasă pot fi argumente ale funcției `show`:

```
show :: Show a => a -> String
```

Putem defini tipul nostru de date `Color` sa fie o instanta a lui `Show`, definind-o astfel:

```
instance Show Color where
     show Red = "Red"
     show Orange = "Orange"
     show Yellow = "Yellow"
     show Green = "Green"
     show Blue = "Blue"
     show Purple = "Purple"
     show White = "White"
     show Black = "Black"
     show (Custom r g b) =
          "Custom " ++ show r ++ " " ++
          show g ++ " " ++ show b
```

În fond, prin această declarare de instanță, specificăm modul în care se convertesc valorile de tip Color la String.

Fără a mai insista, vom face o rapidă trecere în revistă a principalelor clase din Prelude:

Eq

- *descriere*: Tipurile care sunt instanțe ale acestei clase au egalitatea definită pentru ele. Acest lucru înseamnă că toate elementele din astfel de tipuri se pot testa pentru egalitate.
- *instanțe*: Toate tipurile din `Prelude` cu excepția `IO` și a funcțiilor.

Ord

- *descriere*: Tipurile acestei clase au definită o ordine completă pentru ele.
- *instanțe*: Toate tipurile din `Prelude` cu excepția `IO`, a funcțiilor și a `IOError.`

Enum

- *descriere*: Tipurile acestei clase pot fi enumerate. Acest lucru înseamnă că pentru orice element de acest tip există o funcție la un număr natural asociat în mod unic.
- *instanțe*:
    - `Bool`
    - `Char`
    - `Int`
    - `Integer`
    - `Float`
    - `Double`

Show

- *descriere*: conține tipurile care au o reprezentare printabilă, adică toate elementele unui tip din această clasă pot fi argumente ale funcției `show`

- *instanţe*: toate tipurile din Prelude

`Read`

- *descriere*: Tipurile care sunt instanţe ale acestei clase permit o reprezentare în formă string a tuturor.
- *instanţe*: Toate tipurile din `Prelude` cu excepţia `IO` şi a funcţiilor.

`Num`

- *descriere*: Această clasă este "mama" tuturor claselor numerice. Orice element din clasă trebuie să aibă operaţiile numerice de bază definite în ele şi, totodată trebuie posede o conversie de la un `Int` sau `Integer` la un element din acel tip
- *instanţe*:
    - `Int`
    - `Integer`
    - `Float`
    - `Double`

`Real`

- *descriere*: Clasa acoperă toate tipurile numerice ale căror elemente se pot exprima ca o fracţie.
- *instanţe*:
    - `Int`
    - `Integer`

- `Float`
- `Double`

`Fractional`

- *descriere*: Clasa acoperă toate tipurile numerice ale căror elemente sunt fracționare. Orice element din clasă trebuie să aibă definită în el o împărțire, orice element din tip trebuie să aibă un invers şi trebuie să poată fi convertit din numere raționale ca şi din numere cu virgulă mobilă în dublă precizie.
- *instanțe*:
    - `Float`
    - `Double`

`Integral`

- *descriere*: Clasa acoperă toate tipurile numerice ale căror elemente sunt întregi.
- *instanțe*:
    - `Int`
    - `Integer`

`Floating`

- *descriere*: Clasa acoperă toate tipurile numerice ale căror elemente sunt numere cu virgulă mobilă.
- *instanțe*:
    - `Float`
    - `Double`

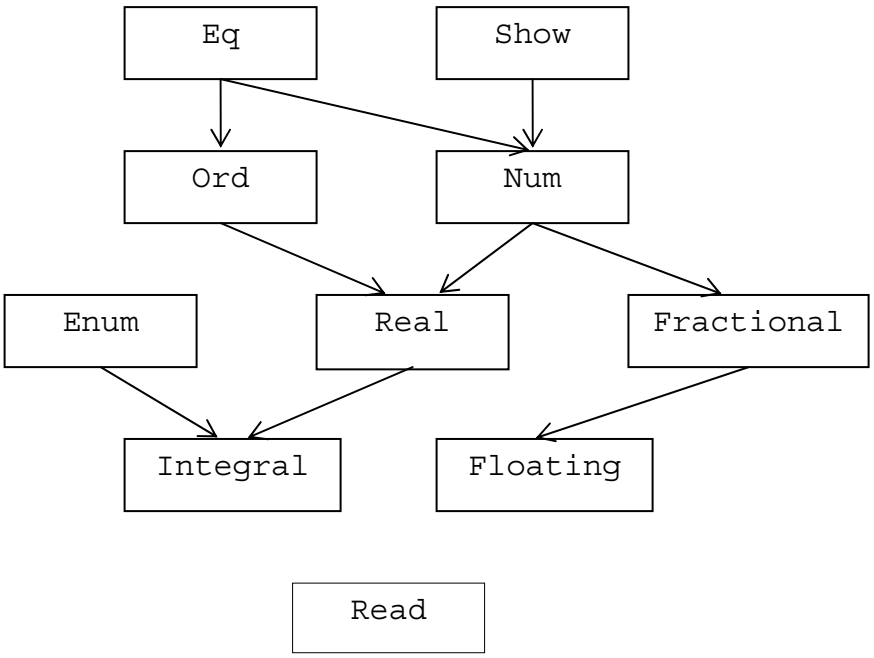
Ierarhia claselor

După cum am observat, un tip de date poate fi comun mai multor clase, şi asta întrucât pentru el se pot defini funcţiile care determină o anumită clasă. Scrierea acestui lucru poate fi făcută şi prin:

```
data Color
     = Red
     | ...
     | Custom Int Int Int -- R G B components
     deriving (Eq, Ord, Show, Read)
```

ducând la crearea automată a instanţelor pentru tipul `Color` în clasele respective.

Pe de altă parte, există şi o ierarhie a claselor în `Prelude`. Un exemplu în acest sens îl figurăm în diagrama următoare. Săgeţile diagramei reprezintă ordonarea claselor în ierarhie. De exemplu, pentru ca un tip să fie în clasa `Num`, el trebuie să fie mai întâi în clasa `Eq`:



Să mai observăm că clasa Read este separată de restul ierarhiei.

6. Comprehensiunea listelor

Progresii aritmetice

De multe ori se întâmplă să dorim o listă de numere aflate în progresie aritmetică. Haskell furnizează, pentru aceasta, aşa numita sintaxă “dot-dot”. De exemplu:

`[1..5]` înseamnă lista `[1, 2, 3, 4, 5]`

Pentru a specifica paşii de mărime diferită de 1, se dau primele două elemente din listă, şi astfel:

`[1, 3..10]` înseamnă `[1, 3, 5, 7, 9]` sau

`[0, 10..50]` înseamnă `[0, 10, 20, 30, 40, 50]`

Atenție, pentru progresiile geometrice această notație este "ilegală":

`[2, 4, 8..128]` este, evident, **invalidă**.

Notația "dot-dot" se utilizează şi pentru alte tipuri, în particular pentru `Float` şi `Char`.

`[1.0 .. 3.5]` înseamnă `[1.0, 2.0, 3.0]`

`[0, 1.5 .. 5]` înseamnă `[0.0, 1.5, 3.0, 4.5]`

Conform sistemului `ASCII`, `['a' .. 'z']` este lista literelor mici, `['A' .. 'Z']` este lista majusculelor şi `['0' ..'9']` este lista cifrelor Putem da, cu această notație şi liste *infinite*. De exemplu:

`[1..]` înseamnă `[1, 2, 3, 4, 5, 6, ...`

`[1, 3..]` este lista tuturor numerelor naturale impare.

Revenind la liste, dorim să punem în evidență un mod de definire al lor asemănător cu cel folosit în matematică atunci când se vorbeşte despre mulțimi. Este vorba de *definirea implicită* a mulțimilor, adică definirea printr-o proprietate comună tuturor elementelor.

Pornind de aici, putem folosi o notație asemănătoare pentru definirea listelor în Haskell. De exemplu, lista valorilor `x^2`, unde `x` este din lista `[1..10]` şi `x` este par se poate defini prin:

```
[x^2 | x <- [1..10], even x]
```

unde `even` este definită în `Prelude`.

Aşadar, *comprehensiunea unei liste* este un mod convenabil de a defini o listă în termeni de alte liste. Avem, în acest sens, trei elemente: *expresia*, *generatorul* şi *filtrul*. Pe exemplul anterior avem:

```
[x^2       | x <- [1..10], even x]
expression   generator    filter
```

Ne gândim la generator ca punând `x` fiecărui element din lista sa:
`x` = 1, `x` = 2, . . . `x` = 10.
Pentru fiecare din aceste elemente, în cazul în care filtrul este adevărat, valoarea este utilizată în expresie. Astfel, vom obține lista:
`[2^2, 4^2, 6^2, 8^2, 10^2]`

Să mai observăm că putem avea *orice număr* de generatori şi de filtre.

Observație:

- generatorii şi gărzile se evaluează de la *stânga la dreapta*. Din acest motiv, ordonarea lor în definirea unei liste este importantă.

Exemplu:
Având 2 generatori, se obțin toate perechile posibile de valori. Astfel, pentru prima valoare din primul generator, obținem *toate* va lorile celui de-al doilea generator. Cu alte cuvinte, lista:

```
[(x,y) | x <- [1..3], y <- ['a'..'b']]
```

este lista:

```
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Această metodă poate fi utilizată şi atunci când executăm o procesare a unui string. Să presupunem că, având un string, vrem să-i scoatem toate

caracterele mici iar majusculele să le convertim în litere mici. Putem face acest lucru fără a utiliza comprehensiunea listelor:

```
Prelude> map Char.toLower (filter Char.isUpper "Hello World")
"hw"
```

sau, utilizând comprehensiunea, putem scrie:

```
Prelude> [Char.toLower x | x <- "Hello World", Char.isUpper x]
"hw"
```

În această situație, din punct de vedere al înțelegerii codului, putem spune că aceste două exprimări sunt echivalente.

Nu este însă cazul pentru exemplul următor. Să presupunem că dorim să obținem lista tuturor punctelor (date ca perechi) de coordonate numere natura-le, aflate între punctul de coordonate (1,1) şi punctul de coordonate (5,7), deasupra diagonalei principale (inclusiv punctele de pe diagonală). Abordările obişnuite ar face codul foarte greu de citit, în schimb, utilizând comprehensiunea listelor putem scrie:

```
Prelude> [(x,y) | x <- [1..5], y <- [x..7]]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(2,2),
(2,3),(2,4),(2,5),(2,6),(2,7),(3,3),(3,4),(3,5),
(3,6),(3,7),(4,4),(4,5),(4,6),(4,7),(5,5),(5,6),
(5,7)]
```

Acest exemplu este util şi pentru a sublinia ordinea de evaluare a generatorilor. Scriind:

```
Prelude> [(x,y) | y <- [x..7], x <- [1..5]]
```

vom primi o **eroare** drept răspuns**.**

7. Din nou despre tipuri

Tipul Maybe

Să încercăm să scriem o funcție similară funcției `head` care ne oferă primul element al unei liste date. Dacă o vom nota cu `fstEl`, definiția tipului ar fi :

```
fstEl :: [a] -> a
```

După cum se poate remarca, totuşi, dacă lista de intrare este vidă, programul care ar utiliza această funcție ar "muri". O soluție ar fi utilizarea constructorului 0-ar, `Nothing`, punând:

```
fstEl [] = Nothing
```

Problema este că, în acest caz, va apare eroare de tip, `Nothing` nefiind de tip `a`.

Soluția dată a fost introducerea unui nou tip de date, **`Maybe`**, definit astfel:

```
data Maybe a = Nothing
             | Just a
```

Acest tip este predefinit în Prelude. Definiția sa ne spune că pentru a crea ceva de tip `Maybe a`, avem două posibilități: prima este utilizarea constructorului 0-ar, `Nothing`, care nu are nici un argument, sau să se utilizeze constructorul `Just`, împreună cu o valoare de tip `a`.
Putem acum să revenim la funcția noastră şi să o definim:

```
fstEl :: [a] -> Maybe a
fstEl [] = Nothing
fstEl (x:xs) = Just x
```

Tot ca un exemplu de utilizare a tipului de date `Maybe`, dorim să definim o funcție `findEl` care să extregă dintr-o listă toate elementele care satisfac un anumit predicat:

```
findEl :: (a -> Bool) -> [a] -> Maybe a
findEl p [] = Nothing
findEl p (x:xs) =
               if p x then Just x
               else findElement p xs
```

Tipuri sinonime

Haskell-ul ne permite să dăm propriile noastre nume tipurilor pe care le-am construit. Ca şi exemplu în acest sens să presupunem că dorim să reprezentăm o parabolă $ax^2+ bx + c$ printr-un triplet:

```
(a, b, c) :: (Float, Float, Float)
```

Codul scris poate fi mult mai uşor de înțeles dacă am da un nume sugestiv tipului `(Float, Float, Float)`. Acest lucru îl realizăm prin declarația:

```
type Quadratic = (Float,Float,Float)
```

(a se observa utilizarea majusculei `Q`).

Pentru ce ar fi utilă o astfel de declarație? Un exemplu ar fi utilizarea sa în construcția unei funcții care obține, de exemplu, rădăcinile ecuației $ax^2+ bx +$

c=0, adică abscisele intersecției parabolei cu axa *Ox*. În loc să scriem definiția funcției:

```
abscise :: (Float,Float,Float) -> [(Float,Float)]
```

putem scrie :

```
abscise :: Quadratic -> [(Float,Float)]
abscise (a, b, c)
     |det >= 0   =[((-b - det1)/(2*a),0),((-b +
          det) / (2*a),0)]
     |otherwise = error "No intersection with Ox"
          where det = sqrt (b*b - 4*a*c)
```

Ca şi observație de final, atunci când se utilizează tipuri sinonime, GHCi-ul utilizează tipul efectiv şi nu numele nou pe care l-am dat noi. Acest lucru se poate vedea cu uşurință în mesajul de eroare următor:

```
*Abscise> abscise('1', '2', '1')
<interactive>:1 :9
     Couldn't match 'Float' against 'Char'
        Expected type: Float
        Inferred type: Char
     In the first argument of 'abscise', namely
             '('1', '2', '1')'
     In the definition of 'it': it = abscise('1',
'2', '1')
*Abscise>
```

Nu a fost utilizat noul nume Quadratic ci tipul real pe care-l reprezintă

Tipuri definite de utilizator

Revenim, în încheiere, la a prezenta câteva modalități de a ne defini tipurile personale.

Înainte de orice, să ne amintim că:

Oamenii fac greşeli.
Computerele nu fac greşeli:
computerele fac exact ceea ce le spun oamenii!

Aşadar, este de dorit să încercăm să facem în aşa fel încât computerele să ne ajute să evităm erorile menționate. Acesta este motivul pentru care este foarte importantă evitarea ambiguității definițiilor şi reprezentărilor. Să dăm un exemplu:

Reprezentări

Să presupunem că dorim să implementăm o funcție care să utilizeze zilele săptămânii. În primul rând trebuie să reprezentăm zilele săptămânii. Utilizând sinonimia de tipuri putem defini:

```
type WeekDay = Int
```

Funcția respectivă, să-i spunem `f` va fi de următorul tip:

```
f :: WeekDay -> String
```

Zilele săptămânii pot fi reprezentate, de exemplu, prin numerele 1, …, 7 şi, corespunzător, ar trebui ca fiecărei funcții care utilizează tipul `WeekDay` să-i punem o precondiție, măcar ca şi comentariu,

```
-- pre-condition: 1 <= d <= 7, where d represents a
```

```
-- day
```

Apare însă o problemă: software-ul se scrie, sau cel puțin așa este etic, pentru a fi înțeles de cat mai mulți utilizatori.

În cazul prezentat, modul de concepere a liniilor de cod poate duce foarte uşor la interpretări greşite sau greşeli de genul următor:

- utilizarea argumentelor de tip `Int` pot fi utilizate în mod eronat: se poate uita că `d` reprezintă o zi, şi i se dă o valoare care reprezintă, în cadrul programului, cu totul altceva (o cantitate de produse, o temperatură, ...)
- chiar dacă cineva nu ar greşi tipul lui `d`, el poate fi confuzat de interpretare. Ce înseamnă că valoarea lui `d` este ? Un român ar spune că `d` reprezintă ziua de Marți, în timp ce un englez ar spune că reprezintă ziua de Luni.

Soluția ar fi să utilizăm un alt mod de definirea tipului de date:

Tipuri de date enumerate

Haskell – ul oferă o facilitate mult mai potrivită pentru reprezentarea datelor. Am văzut, atunci când am vorbit despre clase de tipuri, cum putem reprezenta culorile ca o înşiruire, utilizând cuvântul cheie `data` şi definind propriul nostru tip de date. Acelaşi lucru îl vom face şi pentru problema prezentată mai înainte. În loc să reprezentăm zilele prin `Int`, definim propriul nostru tip de date:

```
data WeekDay = Sun | Mon | Tue | Wed
             | Thu | Fri | Sat
             deriving (Eq,Ord,Show,Read,Enum)
```

În acest fel am creat un **nou tip** numit `WeekDay` şi şapte **valori noi**, numite, de obicei **constructori**. Mai mult am specificat şi că trebuie create automat şi instanțe pentru tipul `WeekDay` în clasele `Eq,Ord,Show,Read,`şi `Enum`. După această definiție, putem verifica:

```
Prelude> Sun
Sun :: WeekDay
Prelude> [Sun, Tue]
[Sun, Tue] :: [WeekDay]
Prelude> (Thu, " is ", 4)
(Thu, " is ", 4) :: (WeekDay, [Char], Int)
```

Aceşti constructori se pot utiliza în acelaşi mod ca în cazul valorilor de orice alt tip.

Câteva *funcții* le avem *pre-definite* pentru tipul nostru `WeekDay`, în mod automat, prin utilizarea clauzei `deriving`. Astfel:

- clasa `Eq` defineşte `==` şi `/=`
- clasa `Ord` defineşte `<, <=, >, >=,` etc.
- clasa Show defineşte funcția `show`
- clasa `Read` defineşte funcția `read`
- clasa `Enum` ne permite utilizarea tipului `WeekDay` în şiruri aritmetice

Verificăm şi aceste chestiuni

```
Prelude> Sun == Tue
False :: Bool
Prelude> Sun < Thu
True :: Bool
Prelude> show Wed
"Wed" :: [Char]
Prelude> read " Wed "
```

```
Wed :: WeekDay
Prelude> [Mon .. Fri]
[Mon, Tue, Wed, Thu, Fri] :: [WeekDay]
```

Ordinea utilizată în funcțiile <, >, etc. este bazată pe ordinea în care apar constructorii în declarația tipului de date. Astfel,

```
Sun < Mon < Tue < …
```

Funcții definite de utilizator peste tipuri enumerate.

Putem defini diferite funcții, ca în cazul oricărui tip de date, şi pentru tipul introdus. Un exemplu ar fi funcțiile de testare - ele trebuie să aibă următoarea formă

```
f :: TipulDefinit -> Bool
```

Pentru `WeekDay`, să considerăm o funcție `isWeekend` care are ca argument o zi `d` şi ne spune dacă este zi de weekend:

```
isWeekend :: WeekDay -> Bool
-- isWeekend d returns True iff d is a
-- weekend day
isWeekend d = d `elem` [Sat, Sun]
```

Tot aşa, putem defini o funcție `isWorkday` care are ca argument o zi `d` şi ne spune dacă este zi de lucru:

```
isWorkday :: WeekDay -> Bool
-- isWorkday d returns True iff d is a
-- working day
```

Există mai multe variante prin care putem defini imaginile lui d prin această funcție:

```
isWorkday d = Mon <= d && d <= Fri
```

sau

```
isWorkday d = d `elem` [Mon .. Fri]
```

sau

```
isWorkday Sun = False
isWorkday Mon = True
isWorkday Tue = True
isWorkday Wed = True
isWorkday Thu = True
isWorkday Fri = True
isWorkday Sat = False
```

Tipuri de date compuse

Există situații în care dorim să asociem valorile (constructorii) unui tip de date cu valorile altui tip de date.

Acest lucru se realizează cu ajutorul **tag**-urilor. Să vedem şi un exemplu.

```
data Temperature = Fahrenheit Float
                 | Celsius Float
                 | Kelvin Float
          deriving (Eq, Ord, Show, Read)
```

Am creat astfel un nou tip, `Temperature`, având trei constructori care sunt **funcții** având drept argumente valori din `Float` şi au valori în `Temperature`. Verificăm:

```
Prelude> Fahrenheit 873
Fahrenheit 873 :: Temperature
Prelude> :t Fahrenheit
Fahrenheit :: Float -> Temperature
Prelude> Fahrenheit 25 == Fahrenheit 34
False :: Bool
Prelude> Fahrenheit 4 == Kelvin 4
False :: Bool
Prelude> Fahrenheit 8 < Kelvin 4
True :: Bool
```

Funcții peste tipuri de date compuse

Ele se definesc în acelaşi mod în care definim orice funcție pentru alte tipuri de date. Pentru tipul considerat, să dăm, din nou, exemple.

Un prim exemplu ar fi să definim o funcție de testare, adică o funcție cu valori booleene. În acest sens, definim o funcție Frost care să testeze dacă o anumită temperatură se află sub pragul de îngheț al apei:

```
frost :: Temperature -> Bool
-- frost t returns True iff for t, the water will
-- freeze
frost (Fahrenheit t) = t <= 32
frost (Celsius t) = t < =0
frost (Kelvin t) = t <= 273
```

Cel de al doilea exemplu va fi o funcție de transformare între constructori, cu alte cuvinte o astfel de funcție va avea ca argument o valoare din `Temperature` şi va returna tot o valoare din `Temperature`. Astfel, definim o funcție `toCelsius`, care, pentru o temperatură t ne oferă valoarea ei în grade Celsius.

```
toCelsius :: Temperature -> Temperature
-- toCelsius t returns t in Celsius
toCelsius (Fahrenheit t)
= Celsius (5 * (t – 32) / 9)
toCelsius (Kelvin t)
= Celsius (t - 273)
toCelsius (Celsius t)
= Celsius t
```

Anexă (Exemple de programe în Haskell)

În cadrul acestei anexe vom încerca să exemplificăm, prin câteva programe, tehnicile învățate până în momentul de față. Totodată, vom găsi şi răspuns la câteva dintre exercițiile propuse pe parcurs.

Aceste programe au constituit teme şi proiecte de laborator la cursul opțional de "Programare funcțională" pe care l-am ținut studenților de la anul III secția Matematică-Informatică.

Ca şi observație, programele pot conține şi mici erori (misiunea de a le corecta revenind cititorului) şi, evident, se pot îmbunătăți. Din acest motiv... atenție la liniile de cod!

I. Primul exemplu poate fi utilizat pentru a testa cunoştințele elevilor din clasele primare privind cele 4 operații cu numere până la 100.

Subliniem faptul că se importă două module, Random care ne oferă numere în mod aleator, şi IO cu care putem obține un mediu interactiv:

```
module Main
       where

  import Random
  import IO

  imp a 0 = 0
  imp a b = 1 + imp (a-b) b


  main = do
    hSetBuffering stdin LineBuffering
```

```
  putStrLn "Buna! Eu te voi ajuta sa socotesti! Cum te
    numesti?"
  name<-getLine
  putStrLn " "
  putStrLn("Buna, "++name++"! Eu voi verifica o parte din
    cunostintele tale de matematica !")
  putStrLn " "
  operatii


operatii = do
  putStrLn "Apasa tasta:"
  putStrLn " "
  putStrLn " 1 pentru a aduna"
  putStrLn " 2 pentru a scadea"
  putStrLn " 3 pentru a inmulti"
  putStrLn " 4 pentru a imparti"
  putStrLn " alta tasta pentru a iesi"
  putStrLn " "
  ans<-getLine
  case ans of
       "1" -> aduna
       "2" -> scade
       "3" -> inmultire
     "4" -> impartire
       _   -> return ()
  putStrLn " "


aduna = do
  fstnum<-randomRIO(1::Int,100)
  secnum<-randomRIO(1::Int,100)
  putStrLn "Trebuie doar sa introduci rezultatul corect !"
  putStrLn " "
  putStr(show fstnum++"+"++show secnum++"=")
  answer1<-getLine
```

```
    let good=fstnum+secnum
    if good==read answer1
       then do putStrLn " "
             putStrLn "BRAVO!!!"
             putStrLn " "
               operatii
       else do putStrLn ":( Mai incearca !"
               putStr(show fstnum++"+"++show secnum++"=")
               answer2<-getLine
               if good==read answer2
                  then do putStrLn "Trebuie doar sa fii mai
                        atent/a ;). BRAVO!"
                          putStrLn " "
                          operatii
                  else do putStrLn "Mai ai dreptul la o
                        incercare !"
                           putStr(show fstnum++"+"++show
                               secnum++"=")
                          answer3<-getLine
                          if good==read answer3
                             then do putStrLn "Nu-i rau. Tot ai
                                    reusit ! BRAVO!"
                                     putStrLn " "
                                     operatii
                             else do putStrLn "Trebuie sa mergi
sa mai inveti ! "
                                     return ()

  scade = do
    fstnum<-randomRIO(1::Int,100)
    secnum<-randomRIO(1::Int,100)
    putStrLn "Trebuie doar sa introduci rezultatul corect !"
    putStrLn " "
    putStr(show fstnum++"-"++show secnum++"=")
    answer1<-getLine
    let good=fstnum-secnum
```

```
  if good==read answer1
     then do putStrLn " "
           putStrLn "BRAVO!!!"
           putStrLn " "
             operatii
     else do putStrLn ":( Mai incearca !"
             putStr(show fstnum++"-"++show secnum++"=")
             answer2<-getLine
             if good==read answer2
                then do putStrLn "Trebuie doar sa fii mai
                   atent/a ;). BRAVO!"
                        putStrLn " "
                        operatii
                else do putStrLn "Mai ai dreptul la o
                      incercare !"
                        putStr(show fstnum++"-"++show
                            secnum++"=")
                        answer3<-getLine
                        if good==read answer3
                           then do putStrLn "Nu-i rau. Tot ai
                                     reusit ! BRAVO!"
                                   putStrLn " "
                                   operatii
                           else do putStrLn "Trebuie sa mergi
                               sa mai inveti ! Mai incercam
                               cand vei fi mai bine
                               pregatit/a !"
                                   return ()
inmultire = do
  fstnum<-randomRIO(1::Int,50)
  secnum<-randomRIO(1::Int,10)
  putStrLn "Trebuie doar sa introduci rezultatul corect !"
  putStrLn " "
  putStr(show fstnum++"*"++show secnum++"=")
  answer1<-getLine
  let good=fstnum*secnum
```

```
  if good==read answer1
     then do putStrLn " "
           putStrLn "BRAVO!!!"
           putStrLn " "
             operatii
     else do putStrLn ":( Mai incearca !"
             putStr(show fstnum++"*"++show secnum++"=")
             answer2<-getLine
             if good==read answer2
                then do putStrLn "Trebuie doar sa fii mai
                       atent/a ;). BRAVO!"
                         putStrLn " "
                         operatii
                else do putStrLn "Mai ai dreptul la o
                           incercare !"
                         putStr(show fstnum++"*"++show
                              secnum++"=")
                         answer3<-getLine
                         if good==read answer3
                            then do putStrLn "Nu-i rau. Tot ai
                                        reusit ! BRAVO!"
                                    putStrLn " "
                                    operatii
                            else do putStrLn "Trebuie sa mergi
                                      sa mai inveti ! Mai
                                      incercam cand vei fi mai
                                      bine pregatit/a !"
                                    return ()
impartire = do
  fstnum<-randomRIO(10::Int,50)
  secnum<-randomRIO(1::Int,10)
  putStrLn "Trebuie doar sa introduci rezultatul corect !"
  putStrLn " "
  putStr "Catul si restul impartirii "
  putStrLn(show fstnum++"/"++show secnum++" ")
  putStr "Catul:"
```

```
cat1<-getLine
putStr "Restul:"
rest1<-getLine
let good=fst(divMod fstnum secnum)
let good1=snd(divMod fstnum secnum)
if (good==read cat1) && (good1==read rest1)
   then do putStrLn " "
         putStrLn "BRAVO!!!"
         putStrLn " "
           operatii
   else do putStrLn ":( Mai incearca !"
           putStrLn "Catul (enter) si Restul (enter):"
           cat2<-getLine
           rest2<-getLine
           if (good==read cat2) && (good1==read rest2)
              then do putStrLn "Trebuie doar sa fii mai
                         atent/a ;). BRAVO!"
                      putStrLn " "
                      operatii
              else do putStrLn "Mai ai dreptul la o
                         incercare !"
                      putStrLn "Catul (enter) si Restul
                            (enter):"
                      cat3<-getLine
                      rest3<-getLine
                      if (good==read cat3) && (good1==read
                            rest3)
                         then do putStrLn "Nu-i rau. Tot ai
                                   reusit ! BRAVO!"
                                 putStrLn " "
                                 operatii
                         else do putStrLn "Trebuie sa mergi
                                   sa mai inveti ! Mai
                                   incercam cand vei fi mai
                                   bine pregatit/a !"
                                 return ()
```

II Următorul proiect a fost conceput pentru a fi testate cunoştinţele din algebră referitor la progresii:

```
module Main
   where

import IO
import Random

main = do
      putStrLn ("Alege una din optiuni");
      putStrLn ("1)Progresie aritmetica.");
      putStrLn ("2)Progresie geometrica.");
      optiune <- getLine
      let var = read optiune
      if var == 1
        then do
             putStrLn "S-a ales pr aritmetica"
             putStrLn "Formula termenului general: a(n+1)
= r + a(n)"
             putStrLn "Dati raza progesiei:"
             pas <- getLine
             let raza=read pas
             hSetBuffering stdin LineBuffering
             a <- randomRIO (1::Int, 100)
             let a1 = sir 1 raza a
             putStrLn ("Primul termen al progresiei a(1)
=" ++ show(a1))
```

```
            putStrLn "Dati rangul termenului dorit:"
            rang <- getLine
            let na = read rang
            putStrLn "Calculeaza termenul:"
            term <- getLine
            let t=read term
            if t == (sir na raza a)
               then do putStrLn "Corect"
               else do putStrLn "Gresit"
                    putStrLn("afisez termenul corect de
rang: " ++ rang)
                      putStrLn(show (sir na raza a) )

            putStrLn "Formula pt calculul sumei
progresiei este:"
            putStrLn "S(n) = (2*a(1)+(n-1)*r)*n/2"
            putStrLn "Calculeaza suma termenilor:"
            sa <- getLine
            let s=read sa
            putStrLn "."
            if s*2 == (suma na raza a)
              then do putStrLn "Corect"
              else do putStrLn "Gresit"
                       putStrLn("afisez suma corecta a
celor " ++ rang ++ " termeni")
                    putStrLn(show (suma na raza a) ++
"/2")
     else if var == 2
           then do putStrLn "S-a ales pr geometrica "

                    putStrLn "Formula termenului
general: a(n+1) = r * a(n)"
                putStrLn "Dati raza progresiei:"
```

```
                rg <- getLine
                let razag=read rg
                hSetBuffering stdin LineBuffering
                d <- randomRIO (1::Int, 100)
                let b1 = prgeo 1 razag d
                putStrLn ("Primul termen al progresiei
b(1) =" ++ show(b1))
                putStrLn "dati rangul termenului dorit:"
                ng <- getLine
                let rangg = read ng
                putStrLn "Calculeaza termenul:"
                termg <- getLine
                let tg=read termg
                if tg == prgeo rangg razag d
               then do putStrLn "Corect"
               else do putStrLn "Gresit"
                     putStrLn ("afisez termenul corect de
                     rang: " ++ rg)
                     putStrLn (show (prgeo rangg razag
d))
                      putStrLn "Formula pt calculul sumei
unei progresii geometrice este:"
                putStrLn "S(n) = (a(1)*(q^n - 1))/(q-1)"
                putStrLn "Calculeaza suma termenilor:"
                putStrLn "Formula pt calculul sumei unei
progresii geometrice este:"
                sg <- getLine
                let s2=read sg
                let s3 = s2 * (razag-1)
                if s3 == suma2 rangg razag d
                   then do putStrLn "Corect"
                   else do putStrLn "Gresit"
```

```
                              putStrLn("afisez suma corecta
a celor " ++ show(rangg) ++ " termeni")
                            putStrLn(show (suma2 rangg razag
d) ++ "/(" ++ show (razag) ++ "- 1)")

            else do putStrLn "Nu s-a ales optiunea
corecta"

--am calculat termenul de rang n pt prog arit
sir 1 r c= c
sir (n+1) r c= r + sir n r c

primt = do
 hSetBuffering stdin LineBuffering
 a <- randomRIO (1::Int, 100)
 putStrLn "."
   --return ()
-- calculez suma termenilor prog arit
suma n r c= ((2 * sir 1 r c + (n - 1) * r) * n)

--am calculat termenul de rang n pt prog geometrica
prgeo 1 r d =d
prgeo (n+1) r d = r * prgeo n r d

--calculez suma termenilor prog geometrice

suma2 n r d=((prgeo 1 r d) * (r ^ n - 1))
```

III. Programul este conceput pentru a testa/calcula diferite noțiuni din combinatorică: permutări, aranjamente, combinări, binomul lui Newton. Ca şi observație, acest program conține şi un “help”.

```
module Project
       where
import      IO

--The factorial of n
perm :: Double -> Double
perm n
  | n==0        = 1
  | n>0         = perm (n-1) * n
  --Generating an exception
  | n<0         = error "Perm only defined on natural numbers"

perm1 :: Int -> Int
perm1 n
  | n==0        = 1
  | n>0         = perm1 (n-1) * n
  --Generating an exception
  | n<0         = error "Perm only defined on natural numbers"

--The arangements of n k
arange :: Double -> Double -> Double
arange n k
  | k==0         = 1
  | n>0          = perm n / perm (n-k)
  | k<0          = error "Arange only defined on natural
numbers"
  | n==0         = error "Arange only defined for n>k"
  | n<0          = error "Arange only defined on natural
numbers"

--The combination of n k
```

```
comb :: Double -> Double -> Double
comb n k
  | k==0          = 1
  | n==0          = error "Comb only defined on natural numbers
: n must be grater than 0 "
  |(n<0)||(k<0)  = error "Comb only defined on natural numbers
: n>0 or k<0 in this case"
  | n>0           = arange n k / perm k
  | n<k           = error "Comb only defined on natural numbers
: n-k < 0 in this case"

--Raising the base a to power n
power :: Int -> Int -> Int
power a n
  | n==0        = 1
  | n>0         = a * power a (n-1)

--Division function
divide    :: Int -> Int -> Int
divide m n
  | m<n         = 0
  | otherwise   = 1 + divide (m-n) n

--Reading an Int
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)

--Calculation of the general term of the Newton's binom
start = do
           putStrLn "\t"
           putStrLn "\t\t\t\tBinomul lui Newton"
           putStrLn "Se considera : (x-a)^n"
           putStrLn "\t"
           putStrLn "Introduceti puterea (n):"
         n <- getInt
```

```
        putStrLn "Introduceti termenul liber (a):"
        a <- getInt
        putStrLn "\t"
        putStrLn ("Am obtinut binomul: (x-
"++show(a)++")^"++show(n))
        putStrLn "Termenul general al binomului este:"
        putStrLn "\t"
        putStrLn ("T(k+1)=(-1)^k * C(n,k) * x^("++show(n)++"-
k) * "++show(a)++"^k")
        putStrLn "\t"
        putStrLn " Doriti sa calculati un anumit termen pentru
binomul introdus? Da|D|da|d"
        putStrLn " Sau, doriti sa calculati toti termenii
pentru acest caz ? All|all|A|a"
        putStrLn " Pentru revenire apasati orice alta tasta."
        putStrLn " Nota : Raspunsul se va da numai pentru una
din intrebari !"
        putStrLn " \t"
        line <- getLine
        if line == "Da" || line == "D" || line == "da" || line
== "d"
         then
                term n a
         else do
                if line == "All" || line=="all" || line == "a"
|| line == "A"
                  then do
                         let k = 0
                         term1 n a k
                  else do
                         putStrLn "\t"
                         putStrLn "Bye!"
                         putStrLn "\t"
term n a = do
            putStrLn "Introduceti rangul pentru care va fi
calculat termenul :"
```

```
            m <- getInt
            let k = m - 1
            if ( k > n )
             then do
                   putStrLn ("Ati introdus o valoare pentru rang
mai mare decat cea permisa !")
                  putStrLn "\t"
                  term n a
             else do
                    putStrLn "\t"
                    let a1 = perm1 n
                    let a2 = perm1 (n-k) * perm1 k
                    if (k `mod` 2 == 0)
                     then
                       putStrLn (" T(" ++ show(k+1) ++ ") = "
++ show (divide a1 a2) ++ " * x^" ++ show(n-k) ++ " * " ++
show(a)++"^"++show(k))
                     else
                       putStrLn (" T(" ++ show(k+1) ++ ") = (-
1)* " ++ show (divide a1 a2) ++ " * x^" ++ show(n-k) ++ " * "
++ show(a)++"^"++show(k))
term1 n a k = do
                let a1 = perm1 n
              let a2 = perm1 (n-k) * perm1 k
                if ( k > n)
                       then do
                             putStrLn "\t"
                             putStrLn "Acestia sunt toti
termenii"
                             putStrLn "\t"
                       else do
                             if (k `mod` 2 == 0)
                               then do
                                      putStrLn (" T(" ++
show(k+1) ++ ") = " ++ show (divide a1 a2) ++ " * x^" ++
show(n-k) ++ " * " ++ show(a)++"^"++show(k))
```

```
                              else do
                                     putStrLn (" T(" ++
show(k+1) ++ ") = (-1)* " ++ show (divide a1 a2) ++ " * x^" ++
show(n-k) ++ " * " ++ show(a)++"^"++show(k))
                             term1 n a (k+1)
help = do
         putStrLn "\t"
         putStrLn "Modulul contine urmatoarele functii: "
         putStrLn "\t"
         putStrLn "\tperm :: Double -> Double"
         putStrLn "\tsintaxa : perm n, unde n este o valoare
intreaga"
         putStrLn "\tinfo:calculeaza permutari de n = n!"
         putStrLn "\t"
         putStrLn "\tarange :: Double -> Double -> Double"
       putStrLn "\tsintaxa:arange n k, unde n si k sunt valori
intregi"
       putStrLn "\tinfo : calculeaza aranjamente de n luate
cate k"
       putStrLn "\t"
         putStrLn "\tcomb :: Double -> Double -> Double"
         putStrLn "\tsintaxa :comb n k , unde n si k sunt
valori intregi"
         putStrLn "\tinfo:calculeaza combinari de n luate cate
k"
         putStrLn "\t"
       putStrLn "\tpower :: Int -> Int -> Int"
         putStrLn "\tsintaxa:power a n, unde n si k valori de
tip Int"
         putStrLn "\tinfo:ridica baza a la puterea n"
         putStrLn "\t"
         putStrLn "\tdivide :: Int -> Int -> Int"
         putStrLn "\tsintaxa:divide m n, unde m si n valori de
tip Int"
         putStrLn "\tinfo:calculeaza partea intrega a impartiri
lui m la n"
```

```
         putStrLn "\t"
         putStrLn "\tstart : functia principala"
         putStrLn "\tinfo: calculeaza termenii generali pentru
un binom oarecare"
         putStrLn "\t"
```

IV. În programul următor se consolidează primele noțiuni pe care le dobândeşte un elev din clasele primare/gimnaziale privind aria şi perimetrul unor triunghiuri particulare:

```
module Main
       where

import IO

main = do
    hSetBuffering stdin LineBuffering
    putStrLn "Eu sunt Calculatorul!Tu cine esti?"
    name <- getLine
    putStrLn("Buna, "++name++"! Acest joc de va ajuta sa-ti
fixezi cunostintele de calcul ale perimetrului si ariei!")
    putStrLn "Trebuie doar sa incerci!"
    doLoop

doLoop = do
    putStrLn "Introdu unul din cele 2 numere:"
    putStrLn "1.Test de geometrie"
    putStrLn "2.Iesire"
    command <- getLine
    if command == "1"
        then doJoc
       else if command == "2"
               then return []
            else doLoop
```

```
doJoc = do
    putStrLn "Introdu coordonatele primului punct"
    x1 <- getLine
    let p1x = read x1
    y1 <- getLine
    let p1y = read y1
    putStrLn "Introdu coordonatele celui de al doilea punct"
    x2 <- getLine
    let p2x = read x2
    y2 <- getLine
    let p2y = read y2
    putStrLn "Introdu coordonatele celui de al treilea punct"
    x3 <- getLine
    let p3x = read x3
    y3 <- getLine
    let p3y = read y3
    let lat1 = sqrt ((p2x-p1x) * (p2x-p1x) + (p2y-p1y) * (p2y-
p1y))
    let lat2 = sqrt ((p3x-p2x) * (p3x-p2x) + (p3y-p2y) * (p3y-
p2y))
    let lat3 = sqrt ((p1x-p3x) * (p1x-p3x) + (p1y-p3y) * (p1y-
p3y))
    if lat3 > lat1 + lat2
         then do putStrLn "Datele introduse nu alcatuiersc un
trinunghi. Atentie!!! Suma celor 2 laturi trebuie sa fie mai
mare decat baza!!"
                 doLoop
       else if lat2 > lat1 + lat3
                  then do putStrLn "Datele introduse nu
alcatuiersc un trinunghi. Atentie!!! Suma celor 2 laturi
trebuie sa fie mai mare decat baza!!"
                        doLoop
                  else if lat1 > lat3 + lat2
                            then do putStrLn "Datele introduse
nu alcatuiersc un trinunghi. Atentie!!! Suma celor 2 laturi
trebuie sa fie mai mare decat baza!!"
```

```
                                    doLoop
                          else if lat1 == lat2
                                                then do putStrLn
"Coordonatele introduse reprezinta un triunghi isoscel"
                                                       putStrLn
("Care este perimetrul unui triunghi isoscel de latura
"++show(lat1)++" si baza "++show(lat3)++"")
                                                let perimetru = 2 *
lat1 + lat3
                                                --doGuessingper
perimetru
                                                doGuessingper1
perimetru
                                                putStrLn "Care este
aria acestui triunghi?"
                                                let p = perimetru /
2
                                                let aria = sqrt
(p*(p-lat1)*(p-lat2)*(p-lat3))
                                                doGuessing1 aria
                                                      --doGuessing
aria
                                                doLoop
                                       else if lat1 == lat3
                                                then do putStrLn
"Coordonatele introduse reprezinta un triunghi isoscel"
                                                       putStrLn
("Care este perimetrul unui triunghi isoscel de latura
"++show(lat1)++" si baza "++show(lat2)++"")
                                                let perimetru = 2 *
lat1 + lat2
                                                  --doGuessingper
perimetru
                                                  doGuessingper1
perimetru
```

```
                                                    putStrLn
"Care este aria acestui triunghi?"
                                               let p = perimetru /
2
                                                 let aria = sqrt
(p*(p-lat1)*(p-lat2)*(p-lat3))
                                                 doGuessing1 aria
                                                     --doGuessing
aria
                                                 doLoop
                                           else if lat3 == lat2
                                                    then do
putStrLn "Coordonatele introduse reprezinta un triunghi
isoscel"

      putStrLn ("Care este perimetrul unui triunghi isoscel de
latura "++show(lat3)++" si baza "++show(lat1)++"")
                                                    let perimetru =
2 * lat2 + lat1
                                                    --doGuessingper
perimetru
                                                    doGuessingper1
perimetru
                                                    putStrLn "Care
este aria acestui triunghi?"
                                                    let p =
perimetru / 2
                                                    let aria = sqrt
(p*(p-lat1)*(p-lat2)*(p-lat3))
                                                    doGuessing1
aria
                                                                --
doGuessing aria
                                                    doLoop
                                               else if lat1 * lat1 +
lat2 * lat2 == lat3 * lat3
```

```
                                                        then do
putStrLn "Coordonatele introduse reprezinta un triunghi
dreputnunghic"

      putStrLn ("Care este perimetrul unui triunghi dreptunghic
determinat de catetele " ++show(lat1)++" si "++show(lat2)++ "
si ipotenuza "++show(lat3)++"")
                                                        let
perimetru = lat1 + lat2 + lat3
                                                        --
doGuessingper perimetru

      doGuessingper1 perimetru
                                                        putStrLn
"Care este aria acestui triunghi?"
                                                        let aria
= (lat1 * lat2) / 2
                                                        --
doGuessing aria

      doGuessing1 aria
                                                        doLoop
                                              else if lat1 * lat1 +
lat3 * lat3 == lat2 * lat2
                                                             then
do putStrLn "Coordonatele introduse reprezinta un triunghi
dreputnunghic"

putStrLn ("Care este perimetrul unui triunghi dreptunghic
determinat de catetele " ++show(lat1)++" si "++show(lat3)++ "
si ipotenuza "++show(lat2)++"")
                                                             let
perimetru = lat1 + lat2 + lat3
                                                             --
doGuessingper perimetru
```

```
doGuessingper1 perimetru

putStrLn "Care este aria acestui triunghi?"
                                                                    let
aria = (lat1 * lat3) / 2
                                                                    --
doGuessing aria

doGuessing1 aria
                                                  else if lat3 *
lat3 + lat2 * lat2 == lat1 * lat1

      then do putStrLn "Coordonatele introduse reprezinta un
triunghi dreputnunghic"

            putStrLn ("Care este perimetrul unui triunghi
dreptunghic determinat de catetele " ++show(lat3)++" si
"++show(lat2)++ " si ipotenuza "++show(lat1)++"")

      let perimetru = lat1 + lat2 + lat3

      --doGuessingper perimetru

      doGuessingper1 perimetru

      putStrLn "Care este aria acestui triunghi?"

      let aria = (lat3 * lat2) / 2

      --doGuessing aria

      doGuessing1 aria

      doLoop
```

```
                                                    else if lat1
== lat2 && lat2 == lat3

      then do putStrLn "Coordonatele introduse reprezinta un
triunghi echilateral"

            putStrLn ("Care este perimetrul unui triunghi
echilateral cu latura egala cu " ++show(lat1)++"?")

      let perimetru = 3 * lat1

      --doGuessingper perimetru

      doGuessingper1 perimetru

      putStrLn "Care este aria acestui triunghi?"

      let p = perimetru / 2

      let aria = sqrt (p*(p-lat1)*(p-lat2)*(p-lat3))

      --doGuessing aria

      doGuessing1 aria

      doLoop
                                                          else do
putStrLn "Coordonatele introduse reprezinta un triunghi
oarecare"

putStrLn ("Care este perimetrul unui triunghi oarecare
determinat de urmatoarele laturi: " ++show(lat1)++" , "
++show(lat2)++ " si "++show(lat3)++"")
                                                                let
perimetru = lat1 + lat2 + lat3
```

```
                                                                    --
doGuessingper perimetru

doGuessingper1 perimetru

putStrLn "Care este aria acestui triunghi?"
                                                                    let
p = perimetru / 2
                                                                    let
aria = sqrt (p*(p-lat1)*(p-lat2)*(p-lat3))
                                                                    --
doGuessing aria

doGuessing1 aria

doLoop


{-doGuessingper perimetru = do
 guess1 <- getLine
 let guessNum1 = read guess1
 if guessNum1 < perimetru
       then do putStrLn "Imi pare rau, dar rezultatul nu este
cel corect! Mai incearca! "
               doGuessingper perimetru
       else if guessNum1 > perimetru
            then do putStrLn "Imi pare rau, dar rezultatul nu
este cel corect! Mai incearca! "
                   doGuessingper perimetru
            else do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale perimetrului"
-}

doGuessing1 aria = do
 guess2 <- getLine
 let guessNum2 = read guess2
```

```
 if guessNum2 < aria
       then do putStrLn ("Imi pare rau, ai gresit, aria
este:"++show(aria)++"")
               doLoop
       else if guessNum2 > aria
            then do putStrLn ("Imi pare rau, ai gresit, aria
este:"++show(aria)++"")
                    doLoop
            else do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale ariei"
                    doLoop
{-
doGuessing aria = do
 guess <- getLine
 let guessNum = read guess
 if guessNum < aria
       then do putStrLn "Imi pare rau, dar rezultatul nu este
cel corect! Mai incearca! "
               doGuessing aria
       else if guessNum > aria
            then do putStrLn "Imi pare rau, dar rezultatul nu
este cel corect! "
                    doGuessing aria
            else do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale ariei"
-}
doGuessingper1 perimetru = do
 guess1 <- getLine
 let guessNum1 = read guess1
 if guessNum1 == perimetru
       then do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale perimetrului"
       else do putStrLn ("Imi pare rau, ai gresit, perimetrul
este:"++show(perimetru)++"")
```

V. Varianta următoare a lui IV utilizează coordonatele în plan şi este o îmbunătățire, în sensul că se surprinde şi cazul triunghiului oarecare:

```
module Main
       where

import IO

main = do
    hSetBuffering stdin LineBuffering
    putStrLn "Eu sunt Calculatorul!Tu cine esti?"
    name <- getLine
    putStrLn("Buna, "++name++"! Acest joc de va ajuta sa-ti
fixezi cunostintele de calcul ale perimetrului si ariei!")
    putStrLn "Trebuie doar sa incerci!"
    doLoop

doLoop = do
    putStrLn "Introdu unul din cele 2 numere:"
    putStrLn "1.Test de geometrie"
    putStrLn "2.Iesire"
    command <- getLine
    if command == "1"
        then doJoc
       else if command == "2"
               then return []
            else doLoop

doJoc = do
    putStrLn "Introdu coordonatele primului punct"
    x1 <- getLine
    let p1x = read x1
    y1 <- getLine
    let p1y = read y1
    putStrLn "Introdu coordonatele celui de al doilea punct"
    x2 <- getLine
```

```
    let p2x = read x2
    y2 <- getLine
    let p2y = read y2
    putStrLn "Introdu coordonatele celui de al treilea punct"
    x3 <- getLine
    let p3x = read x3
    y3 <- getLine
    let p3y = read y3
    let lat1 = sqrt ((p2x-p1x) * (p2x-p1x) + (p2y-p1y) * (p2y-
p1y))
    let lat2 = sqrt ((p3x-p2x) * (p3x-p2x) + (p3y-p2y) * (p3y-
p2y))
    let lat3 = sqrt ((p1x-p3x) * (p1x-p3x) + (p1y-p3y) * (p1y-
p3y))
    if lat3 > lat1 + lat2
         then do putStrLn "Datele introduse nu alcatuiersc un
trinunghi. Atentie!!! Suma celor 2 laturi trebuie sa fie mai
mare decat baza!!"
                 doLoop
       else if lat2 > lat1 + lat3
                  then do putStrLn "Datele introduse nu
alcatuiersc un trinunghi. Atentie!!! Suma celor 2 laturi
trebuie sa fie mai mare decat baza!!"
                        doLoop
                  else if lat1 > lat3 + lat2
                            then do putStrLn "Datele introduse
nu alcatuiersc un trinunghi. Atentie!!! Suma celor 2 laturi
trebuie sa fie mai mare decat baza!!"
                                  doLoop
                       else if lat1 == lat2
                                           then do putStrLn
"Coordonatele introduse reprezinta un triunghi isoscel"
                                                 putStrLn
("Care este perimetrul unui triunghi isoscel de latura
"++show(lat1)++" si baza "++show(lat3)++"")
```

```
                                        let perimetru = 2 *
lat1 + lat3
                                        --doGuessingper
perimetru
                                        doGuessingper1
perimetru
                                        putStrLn "Care este
aria acestui triunghi?"
                                        let p = perimetru /
2
                                        let aria = sqrt
(p*(p-lat1)*(p-lat2)*(p-lat3))
                                        doGuessing1 aria
                                              --doGuessing
aria
                                        doLoop
                                  else if lat1 == lat3
                                        then do putStrLn
"Coordonatele introduse reprezinta un triunghi isoscel"
                                                putStrLn
("Care este perimetrul unui triunghi isoscel de latura
"++show(lat1)++" si baza "++show(lat2)++"")
                                        let perimetru = 2 *
lat1 + lat2
                                          --doGuessingper
perimetru
                                          doGuessingper1
perimetru
                                                putStrLn
"Care este aria acestui triunghi?"
                                        let p = perimetru /
2
                                          let aria = sqrt
(p*(p-lat1)*(p-lat2)*(p-lat3))
                                          doGuessing1 aria
```

```
                                                    --doGuessing
aria
                                                doLoop
                                          else if lat3 == lat2
                                                    then do
putStrLn "Coordonatele introduse reprezinta un triunghi
isoscel"

      putStrLn ("Care este perimetrul unui triunghi isoscel de
latura "++show(lat3)++" si baza "++show(lat1)++"")
                                                    let perimetru =
2 * lat2 + lat1
                                                    --doGuessingper
perimetru
                                                    doGuessingper1
perimetru
                                                    putStrLn "Care
este aria acestui triunghi?"
                                                    let p =
perimetru / 2
                                                    let aria = sqrt
(p*(p-lat1)*(p-lat2)*(p-lat3))
                                                    doGuessing1
aria
                                                                --
doGuessing aria
                                                    doLoop
                                             else if lat1 * lat1 +
lat2 * lat2 == lat3 * lat3
                                                          then do
putStrLn "Coordonatele introduse reprezinta un triunghi
dreputnunghic"

      putStrLn ("Care este perimetrul unui triunghi dreptunghic
determinat de catetele " ++show(lat1)++" si "++show(lat2)++ "
si ipotenuza "++show(lat3)++"")
```

```
                                                        let
perimetru = lat1 + lat2 + lat3
                                                        --
doGuessingper perimetru

      doGuessingper1 perimetru
                                                        putStrLn
"Care este aria acestui triunghi?"
                                                        let aria
= (lat1 * lat2) / 2
                                                        --
doGuessing aria

      doGuessing1 aria
                                                        doLoop
                                              else if lat1 * lat1 +
lat3 * lat3 == lat2 * lat2
                                                              then
do putStrLn "Coordonatele introduse reprezinta un triunghi
dreputnunghic"

putStrLn ("Care este perimetrul unui triunghi dreptunghic
determinat de catetele " ++show(lat1)++" si "++show(lat3)++ "
si ipotenuza "++show(lat2)++"")
                                                              let
perimetru = lat1 + lat2 + lat3
                                                              --
doGuessingper perimetru

doGuessingper1 perimetru

putStrLn "Care este aria acestui triunghi?"
                                                              let
aria = (lat1 * lat3) / 2
                                                              --
doGuessing aria
```

```
doGuessing1 aria
                                                  else if lat3 *
lat3 + lat2 * lat2 == lat1 * lat1

      then do putStrLn "Coordonatele introduse reprezinta un
triunghi dreputnunghic"

            putStrLn ("Care este perimetrul unui triunghi
dreptunghic determinat de catetele " ++show(lat3)++" si
"++show(lat2)++ " si ipotenuza "++show(lat1)++"")

      let perimetru = lat1 + lat2 + lat3

      --doGuessingper perimetru

      doGuessingper1 perimetru

      putStrLn "Care este aria acestui triunghi?"

      let aria = (lat3 * lat2) / 2

      --doGuessing aria

      doGuessing1 aria

      doLoop

                                                     else if lat1
== lat2 && lat2 == lat3

      then do putStrLn "Coordonatele introduse reprezinta un
triunghi echilateral"

            putStrLn ("Care este perimetrul unui triunghi
echilateral cu latura egala cu " ++show(lat1)++"?")
```

```
      let perimetru = 3 * lat1

      --doGuessingper perimetru

      doGuessingper1 perimetru

      putStrLn "Care este aria acestui triunghi?"

      let p = perimetru / 2

      let aria = sqrt (p*(p-lat1)*(p-lat2)*(p-lat3))

      --doGuessing aria

      doGuessing1 aria

      doLoop
                                                          else do
putStrLn "Coordonatele introduse reprezinta un triunghi
oarecare"

putStrLn ("Care este perimetrul unui triunghi oarecare
determinat de urmatoarele laturi: " ++show(lat1)++" , "
++show(lat2)++ " si "++show(lat3)++"")
                                                                  let
perimetru = lat1 + lat2 + lat3
                                                                  --
doGuessingper perimetru

doGuessingper1 perimetru

putStrLn "Care este aria acestui triunghi?"
                                                                  let
p = perimetru / 2
```

```
                                                                    let
aria = sqrt (p*(p-lat1)*(p-lat2)*(p-lat3))
                                                                    --
doGuessing aria

doGuessing1 aria

doLoop


{-doGuessingper perimetru = do
 guess1 <- getLine
 let guessNum1 = read guess1
 if guessNum1 < perimetru
       then do putStrLn "Imi pare rau, dar rezultatul nu este
cel corect! Mai incearca! "
               doGuessingper perimetru
       else if guessNum1 > perimetru
            then do putStrLn "Imi pare rau, dar rezultatul nu
este cel corect! Mai incearca! "
                    doGuessingper perimetru
            else do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale perimetrului"
-}

doGuessing1 aria = do
 guess2 <- getLine
 let guessNum2 = read guess2
 if guessNum2 < aria
       then do putStrLn ("Imi pare rau, ai gresit, aria
este:"++show(aria)++"")
               doLoop
       else if guessNum2 > aria
            then do putStrLn ("Imi pare rau, ai gresit, aria
este:"++show(aria)++"")
                    doLoop
```

```
            else do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale ariei"
                    doLoop
{-
doGuessing aria = do
 guess <- getLine
 let guessNum = read guess
 if guessNum < aria
       then do putStrLn "Imi pare rau, dar rezultatul nu este
cel corect! Mai incearca! "
               doGuessing aria
       else if guessNum > aria
            then do putStrLn "Imi pare rau, dar rezultatul nu
este cel corect! "
                    doGuessing aria
            else do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale ariei"
-}
doGuessingper1 perimetru = do
 guess1 <- getLine
 let guessNum1 = read guess1
 if guessNum1 == perimetru
       then do putStrLn "Felicitari!!!! Se pare ca ti-ai
insusit destul de bine cunostintele de calcul ale perimetrului"
       else do putStrLn ("Imi pare rau, ai gresit, perimetrul
este:"++show(perimetru)++"")
```

VI. Acest din urmă exemplu, neterminat, afişează distribuția pe orbitali a electronilor unui element chimic şi are câteva mici elemente de grafică. Vă invit să-l continuați.

```
module Main
       where

import IO

main = do
    hSetBuffering stdin LineBuffering
    putStrLn "Bine v-am gasit :)"
    putStrLn "In cele ce urmeaza veti vedea o afisare pe
orbitali a electronilor unui element chimic "
    doLoop

doLoop = do
    putStrLn "Introdu numarul corespunzator pentru a alege
optiunea dorita:"
    putStrLn " 1 ---- Sistemul periodic al elementelor"
    putStrLn " 2 ---- Afisare pe orbitali"
    putStrLn " 3 ---- Iesire"
    putStrLn " "
    optiune <- getLine
    if optiune == "1"
        then doelement
        else if optiune == "2"
                then doorbitali
                else if optiune == "3"
                        then return []
                        else doLoop


doelement = do
     putStrLn "Pana in acest moment afisarea se poate realiza
numai pentru urmatoarele elemente:"
```

```
     putStrLn "ELEMENT ------- NUMAR ATOMIC"
     putStrLn "-----------------------------"
     putStrLn "  H     -------      1"
     putStrLn "  He    -------      2"
     putStrLn "  Li    -------      3"
     putStrLn "  Be    -------      4"
     putStrLn "  B     -------      5"
     putStrLn "  C     -------      6"
     putStrLn "  N     -------      7"
     putStrLn "  O     -------      8"
     putStrLn "  F     -------      9"
     putStrLn "  Ne    -------     10"
     putStrLn "-----------------------------"
     doLoop

doorbitali = do
    putStrLn "Introduceti numarul atomic al elementului dorit:"
    nratom <- getLine
    let nratomic = read nratom
    doasezare nratomic
    doLoop

doasezare nratomic = do
    putStrLn "  Pentru a intelege mai bine acestea sunt
semnificatiile elementelor grafice:"
    putStrLn "-------------------------------------------------
---------"
    putStrLn "  K , L , M , N , O , P    ====   straturile de
electroni"
    putStrLn "  -------- o ----------    ====   un singur
electron "
    putStrLn "  -------- @ ----------    ====   doi electroni
grupati"
    putStrLn "-------------------------------------------------
---------"
    putStrLn ""
```

```
    dos1 nratomic

dos1 nratomic = do
   let nrs1 = nratomic - 2
   if nrs1 == (-1)
         then do putStrLn "Stratul K de 2 electroni -- O ------
------------------"
                 doLoop
         else if nrs1 == 0
                   then do putStrLn "Stratul K de 2 electroni -
- @ ------------------------"
                           putStrLn "-------------------------
----------------------------"
                           doLoop
                   else do putStrLn "Stratul K de 2 electroni -
- @ ------------------------"
                           putStrLn "-------------------------
----------------------------"
                           let nrlucrus1 = nratomic - 2
                           dos2 nrlucrus1

dos2 nrlucrus1 = do
   let nrs2 =  nrlucrus1 - 2
   if nrs2 == -1
         then do putStrLn "Stratul L de 8 electroni -- O ------
------------------"
                 putStrLn "-------------------------------------
------------------"
                 doLoop
         else if nrs2 == 0
                   then do putStrLn "Stratul L de 8 electroni -
- @ ------------------------"
                           putStrLn "-------------------------
----------------------------"
                           doLoop
```

```
                  else do putStrLn "Stratul L de 8 electroni -
- @ ------------------------"
                           putStrLn ""
                           let nrlucrus2 = nrlucrus1 - 2
                           dop2 nrlucrus2

dop2 nrlucrus2 = do
   let nrp2 = nrlucrus2 - 6
   if nrp2 == -5
         then do putStrLn "                          ----- O ---
------------------"
                 putStrLn "----------------------------------
------------------"
                 doLoop
         else if nrp2 == -4
                    then do putStrLn "
----- @ ----------------------"
                             putStrLn "-------------------------
-----------------------------"
                             doLoop
                    else if nrp2 == -3
                               then do putStrLn "
----- @o --------------------"
                                       putStrLn "--------------
----------------------------------------"
                                       doLoop
                               else if nrp2 == -2
                                          then do putStrLn "
----- @@ --------------------"
                                                  putStrLn "---
--------------------------------------------------"
                                                  doLoop
                                          else if nrp2 == -1
                                                     then do
putStrLn "                          ----- @@o ------------------
-"
```

```
putStrLn "-------------------------------------------------------
-"

doLoop
                                                        else if
nrp2 == 0

then do putStrLn "                              ----- @@@ ----------
---------"

putStrLn "-------------------------------------------------------
-"

doLoop

else do putStrLn "                              ----- @@@ ----------
---------"

putStrLn "-------------------------------------------------------
-"

let nrlucrup2 = nrlucrus2 - 6

doLoop
```

Bibliografie

[1] H. Barendregt, *The lambda calculus, its syntax and semantics*, North-Holland (1984)

[2] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, New York, 1998.

[3] J. Bokowski: *Computational Oriented Matroids*, Cambridge University Press, 2005

[4] A.Davie. *Introduction to Functional Programming System Using Haskell*. Cambridge University Press, 1992.

[5] H. Daumé III, *Yet Another Haskell Tutorial*, carte on-line la adresa http://darcs.haskell.org/yaht/yaht.pdf

[6] K. Doets, J. van Eijck: *The Haskell Road to Logic, Maths and Programming*. King's College Publications, London, 2004

[7] J. Gibbons, O. de Moor (eds.): *The Fun of Programming*,Palgrave, 2002,

[8] C. Hall, J. O'Donnell: *Discrete Mathematics Using a Computer*, Springer, 2000

[9] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359--411, 1989.

[10] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, *The History of Haskell*, in the Third ACM SIGPLAN History of Programming Languages Conference (HOPL III), San Diego, ACM Press, 2007.

[11] P. Hudak: *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, New York, 2000

[12] J. Hughes, *Why Functional Programming Matters*, The Computer Journal, Vol. 32, No. 2, 1989, pp. 98 - 107. Also in: David A. Turner (ed.): Research Topics in Functional Programming, Addison-Wesley, 1990, pp. 17 - 42.

[13] G. Hutton: *Programming in Haskell*, Cambridge University Press (January 31, 2007)

[14] G. Michaelson, *An Introduction to Functional Programming through Lambda Calculus*, Addison-Wesley, Wokingham, 1988

[16] S. Peyton Jones: *Implementation of Functional Programming Language*, Prentice-Hall, 1987

[17] S. Peyton Jones, D. Lester: *Implementing Functional Languages*, Prentice Hall,1992

[18] S. Peyton Jones (editor). *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*. Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106, Feb 1999.

[19] S. Peyton Jones (editor) *The Haskell 98 Library Report*. Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1105, Feb 1999.

[20] S. Peyton Jones: "Haskell 98 language and libraries: the Revised Report", Cambridge University Press, 2003

[21] F. Rabhi, G. Lapalme: *Algorithms: A functional programming approach*, Addison-Wesley, 1999

[22] J. Rees and W. Clinger (eds.). *The revised$_3$ report on the algorithmic language Scheme*. SIGPLAN Notices, 21(12):37--79, December 1986.

[23] G.L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, Mass., 1984.

[24] S. Thompson: *Type Theory and Functional Programming*, Addison-Wesley, 1991

[25] S. Thompson: *Haskell: The Craft of Functional Programming*, Second Edition, Addison-Wesley, 507 pages, paperback, 1999

[26] P. Wadler. *How to replace failure by a list of successes*. In Proceedings of Conference on Functional Programming Languages and Computer Architecture, LNCS Vol. 201, pages 113--128. Springer Verlag, 1985.

[27] P. Wadler. *Monads for Functional Programming* In Advanced Functional Programming , Springer Verlag, LNCS 925, 1995.

[28] http://haskell.org/haskellwiki/Why_Haskell_matters

[29] http://haskell.org/haskellwiki/Books_and_tutorials

[30] http://haskell.org/haskellwiki/Learning_Haskell

[31] http://ling.ucsd.edu/~barker/Lambda/

[32] http://en.wikipedia.org/wiki/Lambda_calculus

[33] http://www.haskell.org/haskell-history.html

[34] http://haskell.org/definition/haskell98-report.pdf

[35] http://haskell.org/onlinereport/

[36] http://haskell.org/haskellwiki/Future

[37] http://www.cs.mu.oz.au/~bjpop/timeline/timeline.6.pdf

[38] http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html