Fundamentele limbajelor de programare

C03

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Lambda calcul - β-reducții

β-reducții

Convenție. Spunem că doi termeni sunt egali, notat $M = N$, dacă sunt α-echivalenți.

- β-reducție = procesul de a evalua lambda termeni prin "pasarea de argumente funcțiilor"
- β-redex = un termen de forma $(\lambda x.M)\, N$
- redusul unui redex $(\lambda x.M)\, N$ este $M[N/x]$
- reducem lambda termeni prin găsirea unui subtermen care este redex, și apoi înlocuirea acelui redex cu redusul său
- repetăm acest proces de câte ori putem, până nu mai sunt redex-uri
- formă normală = un lambda termen fără redex-uri

β-reducții

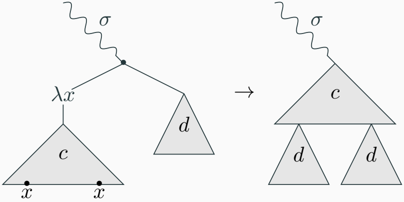
Un pas de β-reducție $\to_\beta$ este cea mai mică relație pe lambda termeni care satisface regulile:

$$(\beta) \qquad \frac{}{(\lambda x.M)N \to_\beta M[N/x]}$$

$$(cong_1) \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$(cong_2) \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}$$

$$(\xi) \qquad \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$



β-reducții

La fiecare pas, subliniem redexul ales în procesul de β-reducție.

$$
\begin{aligned}
(\lambda x.y)\,(\underline{(\lambda z.zz)\,(\lambda w.w)}) &\to_\beta (\lambda x.y)\,((z\,z)[\lambda w.w/z]) \\
&\equiv (\lambda x.y)\,((z[\lambda w.w/z])\,(z[\lambda w.w/z])) \\
&\equiv (\lambda x.y)\,(\underline{(\lambda w.w)\,(\lambda w.w)}) \\
&\to_\beta \underline{(\lambda x.y)\,(\lambda w.w)} \\
&\to_\beta y
\end{aligned}
$$

Ultimul termen nu mai are redex-uri, deci este în formă normală.

β-reducții

$$\begin{aligned}
(\lambda x.y)\,(\underline{(\lambda z.zz\,)\,(\lambda w.w)}) &\to_\beta (\lambda x.y)\,(\underline{(\lambda w.w)\,(\lambda w.w)}) \\
&\to_\beta \underline{(\lambda x.y)\,(\lambda w.w)} \\
&\to_\beta y
\end{aligned}$$

$$\begin{aligned}
\underline{(\lambda x.y)\,((\lambda z.zz\,)\,(\lambda w.w))} &\to_\beta y[(\lambda z.zz\,)\,(\lambda w.w)/x] \\
&\equiv y
\end{aligned}$$

Observăm că:

- reducerea unui redex poate crea noi redex-uri
- reducerea unui redex poate șterge alte redex-uri
- numărul de pași necesari până a atinge o formă normală poate varia, în funcție de ordinea în care sunt reduse redex-urile
- rezultatul final pare că nu a depins de alegerea redex-urilor

β-reducții divergente

Totuși, există lambda termeni care nu pot fi reduși la o β-formă normală (evaluarea nu se termină).

$$\begin{aligned} \omega \equiv \underline{(\lambda x.x\,x)\,(\lambda y.y\,y)} \quad &\rightarrow_\beta \quad (\lambda y.y\,y)\,(\lambda y.y\,y) \equiv \omega \\ &\rightarrow_\beta \quad \ldots \end{aligned}$$

Observați că lungimea unui termen nu trebuie să scadă în procesul de β-reducție; poate crește sau rămâne neschimbată.

β-formă normală

Există lambda termeni care deși pot fi reduși la o formă normală, pot să nu o atingă niciodată.

$$\begin{aligned}
\underline{(\lambda xy.y)\,((\lambda o.o\,o)\,(\lambda p.p\,p))}\,(\lambda z.z) &\to_\beta \underline{(\lambda y.y)\,(\lambda z.z)}\\
&\to_\beta \lambda z.z
\end{aligned}$$

$$\begin{aligned}
(\lambda xy.y)\,(\underline{(\lambda o.o\,o)\,(\lambda p.p\,p)})\,(\lambda z.z) &\to_\beta (\lambda xy.y)\,(\underline{(\lambda p.p\,p)\,(\lambda p.p\,p)})\,(\lambda z.z)\\
&\to_\beta \ldots
\end{aligned}$$

Contează strategia de evaluare.

β-formă normală

Notăm cu $M \twoheadrightarrow_\beta M'$ faptul că M poate fi β-redus până la M' în 0 sau mai mulți pași (închiderea reflexivă și tranzitivă a relației $\rightarrow_\beta$).

M este slab normalizabil *(weakly normalising)* dacă există N în formă normală astfel încât $M \twoheadrightarrow_\beta N$.

M este puternic normalizabil *(strong normalising)* dacă nu există reduceri infinite care încep din M.

Orice termen puternic normalizabil este și slab normalizabil.

Example

$(\lambda x.y)\,((\lambda z.zz\,)\,(\lambda w.w))$ este puternic normalizabil.

$(\lambda xy.y)\,((\lambda o.o\,o)\,(\lambda p.p\,p))\,(\lambda z.z)$ este slab normalizabil, dar nu puternic normalizabil.

Confluența β-reducției

Teorema Church-Rosser. Dacă $M \twoheadrightarrow_\beta M_1$ și $M \twoheadrightarrow_\beta M_2$ atunci există M' astfel încât $M_1 \twoheadrightarrow_\beta M'$ și $M_2 \twoheadrightarrow_\beta M'$.

$$
\begin{tikzcd}
 & M \arrow[dl, ->>] \arrow[dr, ->>] & \\
M_1 \arrow[dr, ->>, dashed] & & M_2 \arrow[dl, ->>, dashed] \\
 & M' &
\end{tikzcd}
$$

Consecință. Un lambda termen are cel mult o β-formă normală (modulo α-echivalență).

Exerciții

Exercițiu. Verificați dacă termenii de mai jos pot fi aduși la o β-formă normală:

1. $(\lambda x.x)\, M$
2. $(\lambda xy.x)\, M\, N$
3. $(\lambda x.x\, x)\, (\lambda y.y\, y\, y)$

Exerciții

Exercițiu. Verificați dacă termenii de mai jos pot fi aduși la o β-formă normală:

1. $(\lambda x.x)\, M$ — Corect: M
2. $(\lambda xy.x)\, M\, N$ — Corect: M
3. $(\lambda x.x\, x)\, (\lambda y.y\, y\, y)$ — Infinit: $(\lambda y.y\, y\, y)\, (\lambda y.y\, y\, y)\, (\lambda y.y\, y\, y) \ldots$

Strategii de evaluare

Strategii de evaluare

De cele mai multe ori, există mai mulți pași de β-reducție care pot fi aplicați unui termen. Cum alegem ordinea? Contează ordinea?

O strategie de evaluare ne spune în ce ordine să facem pașii de reducție.

Lambda calculul nu specifică o strategie de evaluare, fiind nedeterminist. O strategie de evaluare este necesară în limbaje de programare reale pentru a rezolva nedeterminismul.

Strategia normală (normal order)

Strategia normală = *leftmost-outermost*

(alegem redex-ul cel mai din stânga care nu e conținut în alt redex)

- dacă M_1 și M_2 sunt redex-uri și M_1 este un subtermen al lui M_2, atunci M_1 nu va fi următorul redex ales
- printre redex-urile care nu sunt subtermeni ai altor redex-uri (și deci sunt incomparabili față de relația de subtermen), îl alegem pe cel mai din stânga.

Dacă un termen are o formă normală, atunci strategia normală va converge la acea formă normală (știm că e unică).

$$\underline{((\lambda a.a)\,(\lambda xy.y))}\,((\lambda o.o\,o)\,(\lambda p.p\,p))\,(\lambda z.z) \quad \to_\beta$$

Strategia normală (normal order)

Strategia normală = *leftmost-outermost*

(alegem redex-ul cel mai din stânga care nu e conținut în alt redex)

- dacă M_1 și M_2 sunt redex-uri și M_1 este un subtermen al lui M_2, atunci M_1 nu va fi următorul redex ales
- printre redex-urile care nu sunt subtermeni ai altor redex-uri (și deci sunt incomparabili față de relația de subtermen), îl alegem pe cel mai din stânga.

Dacă un termen are o formă normală, atunci strategia normală va converge la acea formă normală (știm că e unică).

$$
\begin{aligned}
(\underline{(\lambda a.a)\,(\lambda xy.y)})\,((\lambda o.o\,o)\,(\lambda p.p\,p))\,(\lambda z.z) &\to_\beta \\
\underline{(\lambda xy.y)\,((\lambda o.o\,o)\,(\lambda p.p\,p))}\,(\lambda z.z) &\to_\beta \underline{(\lambda y.y)\,(\lambda x.x)} \\
&\to_\beta \lambda x.x
\end{aligned}
$$

Strategia aplicativă (applicative order)

Strategia aplicativă = *leftmost-innermost*
(alegem redex-ul cel mai din stânga care nu conține alte redex-uri)

- dacă M_1 și M_2 sunt redex-uri și M_1 este un subtermen al lui M_2, atunci M_2 nu va fi următorul redex ales
- printre redex-urile care nu sunt subtermeni ai altor redex-uri (și deci sunt incomparabili față de relația de subtermen), îl alegem pe cel mai din stânga.

$$(\lambda xy.y)\,(\underline{(\lambda x.x\,x)\,(\lambda x.x\,x)})\,(\lambda z.z) \quad \to_\beta \quad (\lambda xy.y)\,((\lambda x.x\,x)\,(\lambda x.x\,x))\,(\lambda z.z)$$

Strategii în programare funcțională

În limbaje de programare funcțională, în general, reducerile din corpul unei λ-abstractizări nu sunt efectuate (deși anumite compilatoare optimizate pot face astfel de reduceri în unele cazuri).

Strategia *call-by-name* (CBN) = strategia normală fără a face reduceri în corpul unei λ-abstractizări

Strategia *call-by-value* (CBV) = strategia aplicativă fără a face reduceri în corpul unei λ-abstractizări

Majoritatea limbajelor de programare funcțională folosesc CBV, excepție făcând Haskell.

CBN vs CBV

O valoare este un λ-term pentru care nu există β-reducții date de strategia de evaluare considerată.

De exemplu, $\lambda x.x$ este mereu o valoare, dar $(\lambda x.x)\ 1$ nu este.

Sub CBV, funcțille pot fi apelate doar prin valori (argumentele trebuie să fie complet evaluate). Astfel, putem face β-reducția $(\lambda x.M)\ N \twoheadrightarrow_\beta M[N/x]$ doar dacă N este valoare.

Sub CBN, amânăm evaluarea argumentelor cât mai mult posibil, făcând reducții de la stânga la dreapta în expresie. Aceasta este strategia folosită în Haskell.

CBN este o formă de evaluare leneșă *(lazy evaluation)*: argumentele funcțiilor sunt evaluate doar când sunt necesare.

CBN vs CBV

Example

Considerăm 3 și *succ* primitive.

Strategia CBV:

$$
\begin{aligned}
(\lambda x.succ\, x)\,((\lambda y.succ\, y)\,3) &\twoheadrightarrow_\beta (\lambda x.succ\, x)\,(succ\,3)\\
&\rightarrow (\lambda x.succ\, x)\,4\\
&\twoheadrightarrow_\beta succ\,4\\
&\rightarrow 5
\end{aligned}
$$

Strategia CBN:

$$
\begin{aligned}
(\lambda x.succ\, x)\,((\lambda y.succ\, y)\,3) &\twoheadrightarrow_\beta succ\,((\lambda y.succ\, y)\,3)\\
&\twoheadrightarrow_\beta succ\,(succ\,3)\\
&\rightarrow succ\,4\\
&\rightarrow 5
\end{aligned}
$$

Expresivitatea λ-calculului

Expresivitatea λ-calculului

Deși lambda calculul constă doar în λ-termeni, putem reprezenta și manipula tipuri de date comune.

Vom vedea cum putem reprezenta:

- valori booleene (Bool)
- valori opțiune (**Maybe** *a*)
- perechi (**Pair** *a* *b*)
- liste (**List** *a*)
- numere naturale

Bool

Ce este o valoare Bool?
O alegere simplă între două variante

```
data Bool = T | F
```

Ce este o funcție cu domeniu Bool?
O analiză de caz simplă, care produce un rezultat dacă intrarea e **T** și altul dacă intrarea e **F**

```
bool :: a -> a -> Bool -> a
```

Idee: Definim T și F astfel încât

```
bool ifTrue ifFalse b = b ifTrue ifFalse
```

Bool

Ce este o valoare Bool?
O alegere simplă între două variante

```
data Bool = T | F
```

Ce este o funcție cu domeniu Bool?
O analiză de caz simplă, care produce un rezultat dacă intrarea e **T** și altul dacă intrarea e **F**

```
bool :: a -> a -> Bool -> a
```

Idee: Definim T și F astfel încât

```
bool ifTrue ifFalse b = b ifTrue ifFalse
```

- **T** $\triangleq \lambda tf.t$ (dintre cele două alternative o alege pe prima)
- **F** $\triangleq \lambda tf.f$ (dintre cele două alternative o alege pe a doua)

Booleeni

$\mathbf{T} \triangleq \lambda tf.t$ $\mathbf{F} \triangleq \lambda tf.f$ $\mathbf{bool} \triangleq \lambda tfb.btf$

Folosind doar aceste 3 funcții putem defini toate celelalte funcții cu argumente Bool:

```
if :: Bool -> a -> a -> a
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not :: Bool -> Bool
```

Exercițiu
Definiți aceste funcții

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y \qquad \mathbf{bool} \triangleq \lambda tfb.btf$$

$$\begin{aligned} \mathbf{if} &\triangleq \lambda btf.\mathbf{bool}\ t\ f\ b \\ \mathbf{and} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F} \\ \mathbf{or} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2 \\ \mathbf{not} &\triangleq \lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T} \end{aligned}$$

Observați că aceste operații lucrează corect doar dacă primesc ca intrări valori booleene.

Nu există nicio garanție să se comporte rezonabil pe orice alți λ-termeni.

Folosind lambda calcul fără tipuri, avem *garbage in, garbage out*.

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y \qquad \mathbf{bool} \triangleq \lambda tfb.btf$$

$$\begin{aligned}
\mathbf{if} &\triangleq \lambda btf.\mathbf{bool}\ t\ f\ b \\
\mathbf{and} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F} \\
\mathbf{or} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2 \\
\mathbf{not} &\triangleq \lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T}
\end{aligned}$$

Exercițiu. Aduceți la o formă normală următorii termenii:

- **and T F**
- **or F T**
- **not T**

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y \qquad \mathbf{bool} \triangleq \lambda tfb.btf$$

$$\begin{aligned}
\mathbf{if} &\triangleq \lambda btf.\mathbf{bool}\ t\ f\ b \\
\mathbf{and} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F} \\
\mathbf{or} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2 \\
\mathbf{not} &\triangleq \lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T}
\end{aligned}$$

Soluții:

$$\mathbf{and}\,\mathbf{TF} = (\lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F})\,\mathbf{TF} \twoheadrightarrow_\beta \mathbf{if}\ \mathbf{T}\ \mathbf{F}\ \mathbf{F} = (\lambda btf.\mathbf{bool}\ t\ f\ b)\,\mathbf{T}\ \mathbf{F}\ \mathbf{F}$$

$$\twoheadrightarrow_\beta \mathbf{bool}\ \mathbf{F}\ \mathbf{F}\ \mathbf{T} = (\lambda tfb.btf)\,\mathbf{F}\ \mathbf{F}\ \mathbf{T} \twoheadrightarrow_\beta \mathbf{T}\ \mathbf{F}\ \mathbf{F} = (\lambda xy.x)\,\mathbf{F}\ \mathbf{F} \twoheadrightarrow_\beta \mathbf{F}$$

$$\begin{aligned}
\mathbf{or}\,\mathbf{FT} &= (\lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2)\,\mathbf{FT} \twoheadrightarrow_\beta \mathbf{if}\ \mathbf{F}\ \mathbf{T}\ \mathbf{T} \\
&\twoheadrightarrow_\beta \mathbf{F}\ \mathbf{T}\ \mathbf{T} = (\lambda xy.y)\,\mathbf{T}\ \mathbf{T} \twoheadrightarrow_\beta \mathbf{T}
\end{aligned}$$

$$\begin{aligned}
\mathbf{not}\,\mathbf{T} &= (\lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T})\,\mathbf{T} \twoheadrightarrow_\beta \mathbf{if}\ \mathbf{T}\ \mathbf{F}\ \mathbf{T} \\
&\twoheadrightarrow_\beta \mathbf{T}\ \mathbf{F}\ \mathbf{T} = (\lambda xy.x)\,\mathbf{F}\ \mathbf{T} \twoheadrightarrow_\beta \mathbf{F}
\end{aligned}$$

Maybe *a*

Ce este o valoare Maybe ***a*****?**
Două variante din care una încapsulează o valoare de tip *a*

```
data Maybe a = Nothing | Just a
```

Ce este o funcție cu domeniu Maybe ***a*****?**
O funcție (pentru **Just** *a*) sau o valoare implicită (pentru **Nothing**)

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Idee: Definim Nothing și Just astfel încât

```
maybe ifNothing ifJust m = m ifNothing ifJust
```

Maybe *a*

Ce este o valoare Maybe *a*?
Două variante din care una încapsulează o valoare de tip *a*

```
data Maybe a = Nothing | Just a
```

Ce este o funcție cu domeniu Maybe *a*?
O funcție (pentru **Just** *a*) sau o valoare implicită (pentru **Nothing**)

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Idee: Definim Nothing și Just astfel încât

```
maybe ifNothing ifJust m = m ifNothing ifJust
```

- **Nothing** $\triangleq \lambda nj.n$ (dintre cele două alternative o alege pe prima)
- **Just** $\triangleq \lambda anj.ja$ (**Just** *a* aplică al doilea argument valorii *a*)

Maybe *a*

Nothing $\triangleq \lambda nj.n$ **Just** $\triangleq \lambda anj.ja$ **maybe** $\triangleq \lambda njm.mnj$

Folosind doar aceste 3 funcții putem defini toate celelalte funcții cu argumente **Maybe** *a*:

```
fromMaybe :: a -> Maybe a -> a
isNothing :: Maybe a -> Bool
isJust :: Maybe a -> Bool
fmapMaybe :: (a -> b) -> Maybe a -> Maybe b
bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Exercițiu
Definiți aceste funcții (la laborator)

Pair *a b*

Ce este o valoare Pair ***a b*****?**
O valoare care încapsulează o valoare de tip *a* și o valoare de tip *b*

```haskell
data Pair a b = Pair a b
```

Ce este o funcție cu domeniu Pair ***a b*****?**
O funcție care știe ce să facă cu ambele valori

```haskell
unpair :: (a -> b -> c) -> Pair a b -> c
```

Idee: Definim Pair astfel încât

```haskell
unpair f p = p f
```

Pair *a b*

Ce este o valoare Pair ***a b*****?**
O valoare care încapsulează o valoare de tip *a* și o valoare de tip *b*

```
data Pair a b = Pair a b
```

Ce este o funcție cu domeniu Pair ***a b*****?**
O funcție care știe ce să facă cu ambele valori

```
unpair :: (a -> b -> c) -> Pair a b -> c
```

Idee: Definim Pair astfel încât

```
unpair f p = p f
```

- **Pair** $\triangleq \lambda abf.fab$ (**Pair** *a b* aplică funcția valorilor încapsulate)

Pair *a b*

Pair $\triangleq \lambda abf.fab$

uncons $\triangleq \lambda fp.pf$

Folosind doar aceste 2 funcții putem defini alte funcții cu argumente **Pair** *a b*:

```
fst :: Pair a b -> a
snd :: Pair a b -> b
```

Exercițiu
Definiți aceste funcții (la laborator)

List *a*

Ce este o valoare List *a*?
Două variante, una încapsulând o valoare de tip *a* și o altă listă

```
data List a = Nil | Cons a (List a)
```

Ce este o funcție cu domeniu List *a*?
O funcție care știe să agregheze lista

```
foldr :: (a -> b -> b) -> b -> List a -> b
```

Idee: Definim Nil și Cons astfel încât

```
foldr f i l = l f i
```

List a

Ce este o valoare List a?
Două variante, una încapsulând o valoare de tip a și o altă listă

```
data List a = Nil | Cons a (List a)
```

Ce este o funcție cu domeniu List a?
O funcție care știe să agregheze lista

```
foldr :: (a -> b -> b) -> b -> List a -> b
```

Idee: Definim Nil și Cons astfel încât

```
foldr f i l = l f i
```

- **Nil** $\triangleq \lambda fi.i$ (alege valoarea inițială)
- **Cons** $\triangleq \lambda alfi.fa(lfi)$ (**Cons** a l agregează lista, apoi agreghează valorea a în rezultat)

List a

Nil $\triangleq \lambda fi.i$ **Cons** $\triangleq \lambda alfi.fa(lfi)$ **foldr** $\triangleq \lambda fil.lfi$

Folosind doar aceste 3 funcții putem defini alte funcții cu argumente **List** a:

```
(++) :: List a -> List a -> List a
null :: List a -> Bool
map :: (a -> b) -> List a -> List b
filter :: (a -> Bool) -> List a -> List a
foldl :: (b -> a -> b) -> b -> List a -> b  -- greu
reverse :: List a -> List a
uncons :: List a -> Maybe (Pair a (List a)) -- greu
head :: List a -> Maybe a
tail :: List a -> Maybe (List a)
```

Exercițiu
Definiți aceste funcții (la laborator)

Numere naturale

Ce este un număr natural?
Zero sau succesor de un număr natural

```
data Natural = Zero | Succ Natural
```

Ce este o funcție cu domeniu natural?
O funcție care iterează o funcție dată peste o valoare inițială

```
iterate :: (b -> b) -> b -> Natural -> b
```

Idee: Definim Zero și Succ astfel încât

```
iterate f i n = n f i
```

Numere naturale

Ce este un număr natural?
Zero sau succesor de un număr natural

```
data Natural = Zero | Succ Natural
```

Ce este o funcție cu domeniu natural?
O funcție care iterează o funcție dată peste o valoare inițială

```
iterate :: (b -> b) -> b -> Natural -> b
```

Idee: Definim Zero și Succ astfel încât

```
iterate f i n = n f i
```

- **Zero** $\triangleq \lambda fi.i$ (alege valoarea inițială)
- **Succ** $\triangleq \lambda nfi.f(nfi)$ (**Succ** *n* iterează de *n* ori *f* peste *i*, apoi aplică *f* din nou)

Numere naturale

Zero $\triangleq \lambda fi.i$ **Succ** $\triangleq \lambda nfi.f(nfi)$ **iterate** $\triangleq \lambda fin.nfi$

Numeralul Church pentru numărul $n \in \mathbb{N}$ este notat $\overline{n}$.

Numeralul Church $\overline{n}$ este forma normală a λ-termenului **Succ**^{n} **Zero**, adică $\lambda fi.f^n\ i$, unde f^n reprezintă compunerea lui f cu ea însăși de n ori:

$$
\begin{array}{lcl}
\overline{0} & \triangleq & \lambda fi.f^0\ i & = & \lambda fi.i \\
\overline{1} & \triangleq & \lambda fi.f^1\ i & = & \lambda fi.f\ i \\
\overline{2} & \triangleq & \lambda fi.f^2\ i & = & \lambda fi.f\ (f\ i) \\
\overline{3} & \triangleq & \lambda fx.f^3\ i & = & \lambda fi.f\ (f\ (f\ i)) \\
 & \vdots & & & \\
\overline{n} & \triangleq & \lambda fi.f^n\ i & = & \lambda fi.\underbrace{f(f(\ldots(f}_{n}\ i)\ldots))
\end{array}
$$

Numere naturale

Zero $\triangleq \lambda fi.i$ **Succ** $\triangleq \lambda nfi.f(nfi)$ **iterate** $\triangleq \lambda fin.nfi$

Folosind doar aceste 3 funcții putem defini alte funcții:

```
(+), (*) :: Natural -> Natural -> Natural
isZero :: Natural -> Bool
pred :: Natural -> Maybe Natural    -- greu
diff :: Natural -> Natural -> Maybe Natural
(-) :: Natural -> Natural -> Natural -- 0 daca nu se p
(<=), (==) :: Natural -> Natural -> Bool
max :: Natural -> Natural -> Natural
length :: List a -> Natural
sum, product, maximum :: List Natural -> Natural
```

Exercițiu
Definiți aceste funcții (la laborator)

Quiz time!

https://tinyurl.com/C03-Quiz1

Pe săptămâna viitoare!