PARADYGMATY I JĘZYKI PROGRAMOWANIA

Programowanie funkcyjne (w-9)

Zagadnienia

- Wstęp
- Rachunek *lambda* i programowanie funkcjonalne (Podstawa: *An introduction to functional programming through lambda calculus*. Greg Michaelson.)
- Podsumowanie

Entscheidungsproblem

Historia …

Gottfried Wilhelm Leibniz
(1646—1716)

XX wiek

(1864-1943)

- David Hilbert:

Czy istnieje algorytm, który w ramach danego języka formalnego pozwoli udowodnić prawdziwość lub fałszywość zdań wywiedzionych z tego języka?

(1906-1978)

- Kurt Goedel:

W aksjomatycznej niesprzecznej teorii matematycznej zawierającej pojęcie liczb naturalnych da się sformułować takie zdanie, którego w ramach tej teorii nie da się ani udowodnić, ani obalić.

Entscheidungsproblem

- Dowód...

  W latach 1936 i 1937 Alonzo Church i Alan Turing w niezależnie opublikowanych pracach pokazali, że *problem decyzyjny* Hilberta nie posiada ogólnego rozwiązania.

(1903-1995)

- Church: rachunek lambda (λ)
- Turing: maszyna Turinga
- Obie teorie (Churcha i Turinga) miały i mają dominujący wpływ na rozwój informatyki i teorii obliczania

(1912-1954)

Porównanie języków i/f

- Języki imperatywne
  - oparte na ciągu przypisań
  - ta sama nazwa może być związana z wieloma wartościami
  - ustalony porządek obliczeń
  - wartości związane z nazwą są nadawane przez powtarzanie poleceń
  - ...
- Języki funkcyjne
  - oparte na zagnieżdżonych wywołaniach funkcji
  - nazwa może być związana tylko z jedną wartością
  - porządek obliczeń nie ma znaczenia
  - nowe nazwy wiązane są z nowymi wartościami poprzez rekurencyjne wywołania funkcji
  - dostarczają jawnych reprezentacji struktur danych
  - funkcje są wartościami
  - oparte na logice matematycznej, teorii obliczeń, teorii funkcji rekurencyjnych, rachunku λ

Rachunek λ - podstawy

Rachunek λ – podstawy

- Rachunek lambda jest teorią funkcji jako formuł – system operowania funkcjami, wyrażeniami

- <wyrażenie> := <nazwa> | <funkcja> | <obliczanie>
- <funkcja> := λ<nazwa> . <wyrażenie>

  - <nazwa> - zmienna związana (podobna do parametru formalnego w deklaracji funkcji w językach imperatywnych).
    Kropka "." oddziela zmienną związaną od wyrażenia, w którym nazwa jest używana. Jest to ciało funkcji – może być wyrażeniem *lambda*.
    Funkcja nie ma nazwy!
- <obliczanie> := (<wyrażenie_fun> <wyrażenie_arg>)
  – zastosowanie funkcji do wyrażenia, argumentu.

  Przykład: (λx.x λa.λb.b)

Najmniejszy język funkcjonalny

- Definicja w zapisie BNF

<λ -term> := <variable>
| λ <variable> . <λ -term>
| (<λ -term> <λ -term>)
<variable> := x | y | z | …

Lub w zwarty sposób:

E := V | λ V.E | (E1 E2)
V : = x | y | z | …

gdzie V jest dowolną zmienną a E jest dowolnym λ-wyrażeniem.
λV nazywamy głową λ-wyrażenia a E ciałem.

Rachunek λ – podstawy

- Obliczanie, dwa rodzaje
- W obu wypadkach najpierw obliczane jest wyrażenie funkcyjne w celu zwrócenia funkcji
- Sposoby obliczania
  - zmienna związana w ciele funkcji jest zastępowana przez wartość wyrażenia (argument) – *tryb aplikacyjny* (w Pascalu *wywołanie przez wartość*)
  - zmienna związana w ciele funkcji jest zastępowana przez nieobliczone wyrażenie – *tryb normalny* (w Algolu *wywołanie przez nazwę*)
- Na końcu obliczane jest ciało funkcji
- Tryb normalny jest wygodny, lecz może być mniej efektywny

Funkcje w różnych zapisach

Teoria zbiorów:

$\{(x,y) \mid \forall x,y \in N : y = x^2\}$

Algebra:

$f : N \to N$

$f(x) = x^2;$

λ-notacja bez typów:

```
(λx.x*x)
```

Typowana notacja λ:

```
(λx:int.x * x)
```

Polimorficzna notacja λ:

$(\lambda x:\alpha.x*x)$

LISP:

```
(defun square(x) (* x x)))
```

Scheme:

```
(define square
   (lambda (x) (* x x)))
```

Fortran

```
function square(x)
square=x*x
end
```

K & R C:

```
square(x) int x; { return (x*x); }
```

Algol60:

```
integer procedure square(x); integer x;
begin square := x*x end;
```

Pascal:

```
function square (x:integer) : integer;
begin square := x*x end;
```

ANSI C/C++ & Java:

```
int square(int x) { return (x * x); }
```

ML97:

```
fun square x = x * x;
fun square (x:int) = x * x;
val square = fn x => x * x;
```

Haskell:

```
square :: Integer->Integer
square x = x * x
map (\x -> x * x) [1,2,3,4,5]
[(x,y) | x <- [0..], y <- [x * x]]
```

Funkcje

- Funkcje matematyczne:
  `f: X -> X`; f. identycznościowa; X dowolny zbiór
  `g: x -> x`^{2}; (dziedzina, przeciwdziedzina – zbiór wartości)
  `h: x -> e`^{x}
- `λx.x` – funkcja identycznościowa; `λx.x`^{2}`; λx.e`^{x}; etc.
- Zastosujmy funkcję identycznościową do siebie samej: `(λx.x λx.x)`. Tutaj `λx.x` jest zastosowana do argumentu `λx.x`. Podczas obliczania wyrażenia zmienna związana *x* w wyrażeniu funkcyjnym `λx.x` jest zastępowana przez wyrażenie-argument `λx.x` w ciele funkcji `x`, dając `λx.x`, czyli to samo
- Funkcje „samoobliczalne”. Weźmy funkcję `λs.(s s)` – jej ciało `(s s)` jest obliczaniem funkcji `s` na argumencie `s`.
  Zastosujmy `λx.x` do tej funkcji: `(λx.x λs.(s s)) -> λs.(s s)`.
  Zastosowanie `λs.(s s)` *do* `λx.x: (λs.(s s) λx.x) -> (λx.x λx.x) -> λx.x`.
  Dalej:
  `(λs.(s s) λs.(s s)) -> (λs.(s s) λs.(s s))`
  `->...->(λs.(s s) λs.(s s))`
  – proces nieskończony
- Funkcje te można wykorzystać do budowy funkcji rekurencyjnych

Funkcja obliczania funkcji

- `λfun.λarg.(fun arg)` – zastosowanie tej funkcji zwraca funkcję, która stosuje pierwszy argument do drugiego... Np. użyjemy tej funkcji by zastosować funkcję identycznościową do funkcji samoobliczalnej `λs.(s s)`:

```
((λfun.λarg.(fun arg) λx.x) λs.(s s))
-> (λarg.(λx.x arg) λs.(s s))
-> (λx.x λs.(s s))
-> λs.(s s)
```

- Inny zapis dla ułatwienia redukcji:
  - `def id = λx.x`
  - `def self_apply = λs.(s s)`
  - `def apply = λfun.λarg.(fun arg)`
  - `(<nazwa> <argument>) == (<funkcja> <argument>)`
  - `(<funkcja> argument>) => <wyrażenie>`

Funkcje z funkcji

- Inna funkcja identycznościowa
  - `def id2 = λx.((apply id) x);`

    ```
    (id2 id)
    == (λx.((apply id) x) id)
    => ((apply id) id)
    => ((λfunc.λarg.(func arg) id)
    => (λarg.(id arg) id)
    => (id id)
    => ... => id
    ```

  - `id i id2` są równoważne. Załóżmy, że `arg` oznacza dowolne wyrażenie. Mamy

    ```
    (id2 arg)
    == (λx.((apply id) x) arg)
    =>( (apply id) arg)
    => (id arg) => ...=> arg
    ```

Funkcje z funkcji

- Funkcja obliczania funkcji.... Oznaczmy przez `<funct>` dowolną funkcję

  `(apply <funct>) =`
  `= (λ f.λ a.(f a) <funct>) => λ a.(<funct> a)`

  i zastosujmy do `<arg>`

  `(λ a.(<funct> a) <arg>) => (<funct> <arg>)`

  czyli `(apply <funct>)` działa jak oryginalna `<funct>`

- `def self_apply2 = λ s.((apply s) s)`
- Pokazać, że `self_apply2` jest równoważna `self_apply`.

Wybór pierwszego argumentu

- `def select_first = λfirst.λsecond.first`

Funkcja ta ma związaną zmienną `first` i ciało `λsecond.first`; zastosowanie jej do argumentu zwraca funkcję, która zastosowana do innego, dowolnego argumentu zwraca argument pierwszy, np.

- ```
  ((select_first id) coś)
  == ((λfirst.λsecond.first id) coś)
  => (λsecond id) coś) => id
  ```

- ```
  ((select_first <arg1>) arg2)
  == ((λfirst.λsecond.first <arg1>) <arg2>)
  => (λsecond.<arg1>  <arg2>) => <arg1>
  ```

Wybór drugiego argumentu

- `def select_second = λfirst.λsecond.second`

  Związana zmienna: `first`;
  Ciało: `λsecond.second` – identyczność

- Mamy więc

  - `((select_second id) apply)`
    `== ((λfirst.λsecond.second id) apply)`
    `=> (λsecond.second, apply) => apply`

  - dla *dowolnych* `<arg1>`, `<arg2>`:

    `((select_second <arg1>) <arg2>)`
    `== ((λfirst.λsecond.second <arg1>) <arg2>)`
    `=> (λsecond.second <arg2>)`
    `=> <arg2>`

Funkcje `select`

- `select_second` zwraca jedynkę

```
(select_second <byleco>)
== (λfirst.λsecond.second <byleco>)
=> λsecond.second == id
```

- Dalej

```
(select_first id) => id
```

Pokazać.

Tworzenie pary, wybieranie elementów pary

- Poniższa funkcja zachowuje się jak para:

```
def make_pair =
= λfirst.λsecond.λfunc.((func first) second)
```

Zmienna związana: `first`
Zauważmy, że argumenty `first` i `second` są aplikowane przed `func`!
dając `λfunc.((func first) second)`

Funkcja zastosowana do `select_first` wybiera argument pierwszy,
a zastosowana do `select_second`, argument drugi.

- Mamy więc możliwość wybierania elementów

```
(((make_pair <arg1>) <arg2>) select_first)
=> <arg1>

(((make_pair <arg1>) <arg2>) select_second)
=> <arg2>
```

Wyrażenie logiczne, liczby naturalne

Wyrażenia logiczne

- Logika boolowska opiera sie na wartościach `TRUE` i `FALSE` i operacjach `NOT`, `AND`, `OR` itd.
- W wyrażeniu (np. w języku C)

  `<warunek> ? <wyrażenie> : <wyrażenie>`

  wybierane jest pierwsze wyrażenie gdy warunek jest prawdą i drugie gdy warunek jest fałszem
- Wyrażenia warunkowe można modelować za pomocą wersji `make_pair` - funkcji tworzenia par :
  `def cond = λe1.λe2.λc.((c e1) e2)`
- Łatwo sprawdzić (patrz: następna strona), że
  `((cond <expr1>) <expr2>) => λc.((c <expr1>) <expr2>)`

  Stosując to do `select_first` dostaniemy:
  `(λc.((c <expr1>) <expr2>) select_first) =>...=> <expr1>`

  Zastosowanie `cond` do `select_second` daje:
  `(λc.((c <expr1>) <expr2>) select_second) =>...=> <expr2>`

Wyrażenia logiczne

- Sprawdzenia (z poprzedniej strony; <wyr> = wyrażenie)

```
((cond <wyr1>) <wyr2>)
==((λe1.λe2.λc.((c e1) e2) <wyr1>) <wyr2>)
=>λe2.λc((c <wyr1>) e2) <wyr2>)
=>λc.((c <wyr1>) <wyr2>)
```

- Stosując tę funkcję do `select_first` otrzymamy:

```
(λc.((c <wyr1>) <wyr2>) select_first)
=>((select_first <wyr1>) <wyr2>)
=>...=> <wyr1>
```

Zastosowanie do `select_second` prowadzi do:
`(λc.((c <wyr1>) <wyr2>) select_second) => <wyr2>`

- Operacje logiczne możemy modelować, definiując wartości `true i false:`

```
def true = select_first
def false = select_second
```

Wyrażenia logiczne - NOT

NOT

jest operatorem jednoargumentowym: `NOT <operand>`

- `def not = λx.(((cond false) true) x)`

X	NOT X
FALSE	TRUE
TRUE	FALSE

X ? FALSE : TRUE

Sprawdźmy działanie. Wnętrze:

```
(((cond false) true) x)
== (((cond λe2.λc.((c e1) e2) false) true) x)
=> ((λe2.λc.((c false) e2) true) x)
=> (λc.((c false) true) x) => ((x false) true)
```

- Uproszczona definicja `NOT` jest więc:

```
def not = λx.((x false) true)
```

Sprawdzenie „NOT TRUE”:

```
(not true) == (λx.((x false) true) true) => ((true false) true)
== ((λfirst. λsecond.first false) true) => (λsecond.false true) => false
```

Podobnie „NOT FALSE”

```
((not false) == ((λx.((x false) true) false)
=> (λsecond.false true) => true
```

Wyrażenia logiczne - AND

- AND
  Jeśli X jest TRUE to wynikiem X AND Y jest drugi argument, a jeśli X jest FALSE to wynikiem jest FALSE.

X	Y	X AND Y
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

X ? Y : FALSE

- Reguła działania selektorów: jeśli lewy operand == TRUE to wybieramy prawy operand jako wynik, a jeśli lewy operand == FALSE wybieramy lewy

Wyrażenia logiczne - AND

- Definicja

```
def and = λx.λy.(((cond y) false) y)
```

- Uproszczenie ciała funkcji wewnętrznej

```
(((cond y) false) x)
== (((λe1.λe2.λc.((c e1) e2) y) false) x)
=> ((λe2.λc.((c y) e2) false) x)
=> (λc.((c y) false) x)
=> ((x y) false)
```

- Definicja `AND` po uproszczeniu:

```
def and = λx.λx.((x y) false)
```

- Sprawdzenie, np. „TRUE AND FALSE”:

```
((and true) false) == ((λx.λy.((x y) false) true) false)
=> (λy.((true y) false) false)
=> ((true false) false)
=> ((λfirst. λsecond.first false) false)
=> (λsecond.false) false) => false
```

Wyrażenia logiczne - OR

- OR

  Jeśli pierwszym operandem jest TRUE to wynik jest TRUE; w innym przypadku wynikiem jest drugi operand

  - Definicja

    ```
    def or = λx.λy.(((cond true) y) x)
    ```

  - Uproszczenie

    ```
    (((cond true) y) x)
    == (((λe1.λe2.λc.((c e1) e2) true) y x)
    => ((λe2.λc.((c true) e2) y) x)
    => (λc.((c true) y) x)
    ((x true) y)
    ```

  - Nowa definicja

    ```
    def or = λx.λy.((x true) y)
    ```

  - Sprawdzić poprawność definicji z tabelką prawdy, tzn. obliczyć: `((or false) true), ((or false), false),` itd.

X	Y	X OR Y
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

X ? TRUE : Y

Wyrażenia logiczne, podsumowanie

- NOT

  def not = λx.((x false) true)

- AND

  def and = λx.λy.((x y) false)

- OR

  def or = λx.λy.((x true) y)

Wyrażenia logiczne, liczby całkowite

- Liczby całkowite.
  Jak reprezentować jawnie liczby całkowite?
- Oprzemy się na stwierdzeniu, że są one następnikami zera:
  1 = następnik 0
  2 = następnik 1 = następnik następnika 0
  3 = następnik 2 = nastepnik następnika następnika 0
  itd.
- Potrzebujemy funkcji reprezentującej zero oraz funkcji następnika: `succ` takich, że:

```
def one = (succ zero)
def two = (succ one)
```

itd.

Wyrażenia logiczne, liczby całkowite

- Zero możemy reprezentować np. przez `id`:

```
def zero = id
```

natomiast `succ` przez:

```
def succ = λn.λs.((s false) n)
```

Za każdym razem gdy zastosujemy *succ* do liczby *n* otrzymamy parę funkcji gdzie `false` jest pierwsza i następnie jest liczba `n`

- `one == (succ zero) == λn.λs((s false) n) zero) => λs.((s false) zero)`
- `two == (succ one) == (λn.λs.((s false) n) one) => λs.((s false) one) == λs.((s false) λs.((s false) zero))`
- `three = (succ two) == (λn.λs.((s false) n) two) => λs.((s false) two) == λs.((s false) λs.((s false) one) == λs.((s false) λs.((s false) λs.((s false) zero)))`
- itd. (Zadanie: Wypisać wyrażenia dla `four, five`)

Wyrażenia logiczne, liczby całkowite

- Można zdefiniować funkcję `iszero`, która sprawdza, czy liczba jest zerem i zwraca `true` lub `false`; np.

```
λs.((s false) <number>).
```

Jeśli argumentem jest `select_first` to wybierane jest `false` (Pamiętamy, że liczba jest funkcją z argumentem, który może być użyty jako selektor)

```
(λs.((s false) <number>) select_first)
=> ((select_first false) <number>)
== ((λfirst.λsecond.first false) <number>)
=> (λsecond.false <number>)
=> false
```

Wyrażenia logiczne, liczby całkowite

- Jeśli zastosujemy `zero` (identyczność) do `select_first` to zostanie zwrócona wartość `select_first`, która jest z definicji równa `true`

```
(zero select_first) ==
(λx.x select_first) =>
select_first ==
true
```

- To sugeruje definicję

```
def iszero = λn.(n select_first)
```

Podsumowanie...

def id x = x

def self_apply s = s s

def apply func = λarg.(func arg)

def apply func arg = func arg

def select_first first = λsecond.first

def select_first first second = first

def select_second first = λsecond.second

def select_second first second = second

def make_pair e1 = λe2.λc.(c e1 e2)

def make_pair e1 e2 = λc.(c e1 e2)

def make_pair e1 e2 c = c e1 e2

def cond e1 e2 c = c e1 e2

def true first second = first

def false first second = second

def not x = x false true

def and x y = x y false

def or x y = x true y

Inne funkcje tworzy się podobnie.

Nowy zapis ...

- Dla pewnych funkcji mamy standardowy, równoważny zapis, np. zamiast

```
cond <wyr_prawdy>
<wyr_fałszu> <warunek>
```

piszemy

```
if <warunek>
then <wyr_prawdy>
else <wyr_fałszu>
```

- Zapiszemy więc, np. dla `not`

```
def not x =
  if x
  then false
  else true
```

- dla `and`:

```
def and x y =
   if x
   then y
   else false
```

- i dla `or`:

```
def or x y =
   if x
   then true
   else y
```

- itd

Rekurencja

- Jak używana jest rekurencja w programowaniu funkcyjnym?
- W językach funkcyjnych programy oparte są na strukturalnie zagnieżdżonych wywołaniach funkcji
- Powtarzanie opiera się na rekurencji: „*definiowaniu czegoś przez to samo”*
- Należy rozróżniać prostą rekurencję gdzie znamy liczbę powtórzeń od rekurencji ogólnej gdzie liczba powtórzeń nie jest znana
- Przykład. Dodawanie dwu liczb poprzez zwiększanie jednej z nich i jednoczesne zmniejszanie drugiej do momentu aż stanie się zerem

```
def add x y =
    if iszero y
    then x
    else add (succ x) (pred y)
```

Operacje arytmetyczne

- Potęgowanie (naturalne)

  Potęgi liczb naturalnych oblicza się przez mnożenie liczby potęgowanej przez siebie, zmniejszenie wykładnika o jeden, i dalsze powtarzanie mnożenia otrzymanego wyniku przez liczbę potęgowaną aż do momentu gdy wykładnik będzie zerem, a więc potęga będzie jedynką. Funkcja potęgowania jest postaci

```
rec power x y =
  if iszero y
  then one
  else mult x (power x (pred y))
```

Potęgowanie. Przykład


```
power two three => ... =>
mult two
     (power two (pred three)) -> ... ->
mult two
     (mult two
          (power two (pred (pred three)))) -> ... ->
mult two
     (mult two
          (mult two
               (power two (pred (pred (pred three)))))) -> ... ->
mult two
     (mult two
          (mult two one)) -> ... ->
mult two
     (mult two two) -> ... ->
mult two four => ... =>
eight
```

Odejmowanie

- Różnicę dwu liczb naturalnych znajdziemy, powtarzając obliczenia dla różnicy obu zmniejszonych o jeden liczb. Jeśli w kolejnym kroku pierwsza z liczb stanie się zerem to druga będzie szukaną różnicą

```
rec sub x y =
  if iszero y
  then x
  else sub (pred x) (pred y)
```

- Policzyć dla wprawy `sub four two => 2`
  Uwaga: `sub` zwraca zero dla `x<y`.

Dzielenie

- Problem dzielenia przez zero
- Dla niezerowego dzielnika zliczamy ile razy można go odjąć od dzielnej aż do momentu gdy dzielna będzie mniejsza od dzielnika

```
rec div1 x y =
    if greater y x
    then zero
    else succ (div1 (sub x y) y)

def div x y =
    if iszero y
    then zero
    else div1 x y
```

- Pokazać, że `div seven three => 2`

itd.

- Typy; reprezentacja
- Sprawdzanie typów
- Listy i operacje na listach
- Napisy (strings)
- Struktury bardziej złożone; drzewa
- Leniwe obliczanie
- ...
- Języki
  - SCHEME
  - ML
  - LISP
  - ...

Przykład: przekazanie przez wartość


```
(λx.λf.f(succ x))(λz.(λg.(λy.(add (mul (g y) x)) z)))) (λz.(add z 3)) 5)

   => (λx.λf.f(succ x))(λz.(λg.(λy.(add (mul (g y) x)) z)))) (add 5 3)

   => (λx.λf.f(succ x))(λz.(λg.(λy.(add (mul (g y) x)) z)))) 8

   => (λf.f(succ 8))(λz.(λg.(λy.(add (mul (g y) 8))) z))

   => (λz.(λg.(λy.(add (mul (g y) 8))) z)) (succ 8)

   => (λz.(λg.(λy.(add (mul (g y) 8))) z)) 9

   => (λg.(λy.(add (mul (g y) 8))) 9)
```

Redukcję argumentu `((λz.(add z 3)) 5)` należy traktować jak optymizację ponieważ przekazana została wartość, a nie wyrażenie, które byłoby obliczane dwukrotnie w ciele funkcji. Wyrażenie końcowe (jeszcze nie zredukowane do postaci normalnej) zawiera wywołania przez wartość.

Konstrukcja list

- Funkcja parowania: `pair`

```
def pair = λa.λb.λf.f a b
```

- Funkcje selekcji elementów *head* i *tail*

```
def head = λg.g(λa.λb.a)
def tail = λg.g(λa.λb.b)
```

Konstrukcja list

- Sprawdzenie poprawności funkcji `head (pair p q)`

```
head(pair p q) =>
=> (λg.g (λa.λb.a))((λa.λb.λf.f a b) p q)
=> ((λa.λb.λf.f a b) p q) (λa.λb . a)
=> ((λb.λf.f p b) q)(λa.λb.a)
=> (λf.f p q)(λa.λb.a)
=> (λa.λb.a) p q
=> (λb.p) q
=> p
```

- Zadanie. Sprawdzić poprawność funkcji `tail`

Konstrukcja list

- Funkcja `pair` wystarcza do konstruowania dowolnie długich list (podobnie jak `cons` w LISP). Zdefiniujmy stałą specjalną `nil`

```
def nil = λx.λa.λb.a
```

- Listę `[1,2,3,4]` można zbudować następująco:

```
def [1,2,3,4] =
    pair 1 (pair 2 (pair 3 (pair 4 nil)))
```

- Kombinacje selektorów `head` i `tail` pozwalają wybierać dowolne elementy z listy. Np.

```
head(tail (tail [1,2,3,4])) => 3
```

Języki funkcyjne

- Wszystkie języki funkcyjne są w pewnym sensie syntaktyczną odmianą rachunku *lambda*
- Podstawową operacją we wszystkich językach funkcyjnych jest budowa funkcji – abstrakcji lambda i ich obliczanie
- Nazywanie funkcji ułatwia zapis (czytelność) – *lukier składniowy*
- Podstawą powtarzania obliczeń jest rekurencja
- Większość języków funkcyjnych używa zakresów statycznych, leksykalnych (bloki)
- Wyrażenie `let` lub `where` pozwala zastąpić konstrukcję lambda i jej aplikację. Np.

  `let x=6 in (add x 5)` oznacza `(λx.(add x 5)) 6`

  lub

  `(add x 5) where x=6` też oznacza to samo

Redukcje α, β, η, δ

- red-α: pozwala zamieniać zmienne związane w wyrażeniu lambda na inne zmienne
  $\lambda v.E \Rightarrow_{\alpha} \lambda w.E[v->w]$
- red-β: przekazanie argumentu (wyrażenia) do funkcji
  $(\lambda v.E)E_1 \Rightarrow_{\beta} E[v->E_1]$
- red-η: Jeśli v jest zmienną, a E wyrażeniem i v nie występuje w E to (nie stosuje się do E reprezentujących stałe)
  $\lambda v.(E\ v) \Rightarrow_{\eta} E$
- red-δ: w rachunku *lambda ze stałymi* reguły redukcji związane z tymi stałymi i funkcjami noszą nazwę reguł delta
- Wyrażenie lambda jest w postaci normalnej jeśli nie daje się redukować beta lub delta. Wyrażenie nie zawiera nieobliczonych funkcji

Listy...

- LISTA
  - nil jest listą
  - cons h t jest listą (h – head, t – tail; głowa i ogon)
  - jeśli h jest dowolnym obiektem i t jest listą
- cons jest tradycyjną nazwą konstruktora list
  - cons 4 nil – lista, w której head=4, a tail=nil
  - head cons h t => h;
    head nil => error
- tail cons h t => t;
  tail nil => error

```
def HEAD L =
  if islist L
  then (value L) select_first
  else LIST_ERROR

def TAIL L =
  if islist L
  then (value L) select_second
  else LIST_ERROR
```

Długość listy

- Funkcja obliczania długości listy (rekurencja)

```
LENGTH nil = 0
LENGTH (CONS H T) = SUCC (LENGTH T)
```

```
definicja
rec LENGTH L =
  IF ISNIL L
  THEN 0
  ELSE SUCC (LENGTH (TAIL L))
```

- Przykład

```
LENGTH (CONS 1 (CONS 2 (CONS 3 NIL))) -> ... ->
SUCC (LENGTH (CONS 2 (CONS 3 NIL))) -> ... ->
SUCC (SUCC (LENGTH (CONS 3 NIL))) -> ... ->
SUCC (SUCC (SUCC (LENGTH NIL))) -> ... ->
SUCC (SUCC (SUCC 0))) ==
3
```

Dodawanie elementów – append

- Dodawanie elementów do listy

  APPEND NIL L = L

  APPEND (CONS H T) L = CONS H (APPEND T L)

- Przykład

```
definicja
rec APPEND L1 L2 =
  IF ISNIL L1
  THEN L2
  ELSE CONS (HEAD L1) (APPEND (TAIL L1) L2)
```

APPEND (CONS 1 (CONS 2 NIL)) (CONS 3 (CONS 4 NIL)) -> ... ->

CONS 1 (APPEND (CONS 2 NIL) (CONS 3 (CONS 4 NIL))) -> ... ->

CONS 1 (CONS 2 (APPEND NIL (CONS 3 (CONS 4 NIL)))) -> ... ->

CONS 1 (CONS 2 (CONS 3 (CONS 4 NIL)))

Zapis...

- Inny zapis dla list

```
CONS (CONS 5 (CONS 12 NIL))
        (CONS (CONS 10 (CONS 15 NIL))
                (CONS (CONS 15 (CONS 23 NIL))
                        (CONS (CONS 20 (CONS 45 NIL))
                                NIL)))
```

[[5,12],[10,15],[15,23],[20,45]]

W listach skonstruowanych za pomocą [i] zakłada się istnienie list pustej []

Usuwanie elementów

- Jeśli lista jest pusta zwracana jest lista pusta.
  Jeśli pierwszy element listy jest taki jak usuwany to zwracany jest ogon listy; rekurencja

  DELETE X [] = []
  DELETE X (H::T) = T if <equal> X H

```
definicja
rec DELETE V L =
   IF ISNIL L
   THEN NIL
   ELSE
      IF EQUAL V (HEAD L)
      THEN TAIL L
      ELSE (HEAD L)::(DELETE V (TAIL L))
```

- Przykład

DELETE 10 [5,10,15,20]
(HEAD [5,10,15,20])::(DELETE 10 (TAIL [5,10,15,20])) -> ... ->
5::(DELETE 10 ([10,15,20]))
5::(TAIL [10,15,20]) -> ... ->
5::[15,20] => ... =>
[5,15,20]

Porównywanie list

- porównywanie

```
LIST_EQUAL [] [] = TRUE
LIST_EQUAL [] (H::T) = FALSE
LIST_EQUAL (H::T) [] = FALSE
LIST_EQUAL (H1::T1) (H2::T2) = LIST_EQUAL T1 T2
                               if <equal> H1 H2
LIST_EQUAL (H1::T1) (H2::T2) = FALSE
                               if NOT (<equal> H1 H2)
```

- Napisz definicję rekurencyjną

  rec LIST_EQUAL L1 L2

  korzystając z podanego powyżej przepisu działania funkcji

Porównywanie list

- ...

LIST_EQUAL (TAIL [2,3]) (TAIL [2,4]) -> ... ->

LIST_EQUAL [3] [4] -> ... ->

{ EQUAL (HEAD [3]) (HEAD [4])) -> ... ->

EQUAL 3 4 -> ... ->

FALSE}

FALSE

definicja

```
rec LIST_EQUAL L1 L2 =
  IF AND (ISNIL L1) (ISNIL L2)
  THEN TRUE
  ELSE
    IF OR (ISNIL L1) (ISNIL L2)
    THEN FALSE
    ELSE
      IF EQUAL (HEAD L1) (HEAD L2)
      THEN LIST_EQUAL (TAIL L1) (TAIL L2)
      ELSE FALSE
```

za tydzień ... !?