lazy evaluation

Chapter 15

In Haskell computation is the application of functions to arguments.

```
inc:: Int -> Int
inc n = n + 1
```

inc (2 * 3) can be evaluated in this order:

```
inc (2 * 3)                 but also     inc (2 * 3)
= applying inc                            =applying *
(2 *3) +1                                 inc 6
= applying *                             =applying inc
6 + 1  applying +                         6 + 1 applying +
7                                          7
```

in Haskell any two different ways of evaluating the same expression will always produce the same final value, provided tht they both terminate

this property does not apply to imperative languages: suppose n=0,

n + (n = 1)	n + (n = 1)
=applying n	= applying =
0 + (n = 1)	n + 1
= applying =	= applying n
0 + 1	1 + 1
=applying +	= applying +
1	2

evaluation stategies
an expression that has the form of a function applied to one or more arguments is a reducible expression = redex

```
mult :: (Int, It) -> Int
mult (x,y) = x * y
```

mult (1+2, 2+3) has 3 redexes
1+2 ➔ mult (3, 2+3)
2+3 ➔ mult (1 + 2, 5)
and mult (1+2, 2+3) ➔ (1 +2) * (2 + 3)
which redex should we pick ?

innermost evaluation : pick a redex that does not contain other redexes

in our example 1 + 2 or 2 + 3 when there are ties, like here, we pick the leftmost, i.e., 1 + 2

mult (1 +2, 2 + 3)
= applying first +
mult (3, 2 + 3)
=applying +
mult (3, 5)
=applying mult
3 * 5 applying * = 15

in innermost computation arguments are passed to functions by value

dual to innermost is the outermost computation: in which one always chooses a redex that is not contained in any other redex. Again, in case of ties, the leftmost outermost redex is chosen

mult (1 + 2, 2 + 3)
= applying mult
(1 + 2) * (2 + 3)
=applying first +
 3 * (2 + 3)
= applying + = 3 * 5 = 15

with outermost computation arguments are passed to functions as not evaluated
this is called «by name»

many built-in functions require their arguments to be evaluated before being applied

for instance +, * etc

such functions are called strict

exercise 1 identify the redexes

1 + (2 * 3)

(1 + 2) * (2 + 3)

fst (1 + 2, 2 + 3)

(\x -> 1 + x) (2 * 3)

2) show why outermost evaluation is preferable to innermost for

fst (1 + 2, 2 + 3)

Lambda expressions

```
mult :: Int -> Int ->Int
mult  x = \y -> x * y
```

innermost

```
mult (1 + 2) (2 + 3)
= first +
mult 3 (2 + 3)
= applying mult
(\y -> 3 * y) (2 + 3)
= +
(\y -> 3 * y) 5  applying lambda = 3 * 5  = 15
```

the arguments are passed (by value) one at the time

Notice: the selection of redexes within the body of lambda expressions is forbidden

functions are viewed as black boxes

(\x -> 1 + 2) is fully evaluated

(\x -> 1 + 2) 0
= lambda
1 + 2
= +
3

innermost and outermost evaluation, but not within lambda expressions are called
call by value and call by name evaluation, respectively

Termination:

```
inf :: Int
inf = 1 + inf
```

evaluation of inf does not terminate, but consider

fst (0, inf) , with call by value

```
fst (0, inf)
= inf
fst (0, 1 + inf)
= inf
fst (0, 1 + (1 + inf))
=....
```

whereas, with call by name:
fst (0, inf)
= fst
0

call ny name may produce a result for expressions for which call by value does not terminate

Theorem
if there is any evaluation sequence that terminates for a given expression, then call by name evaluation will also terminate for that expression and with the same final result

Number of reductions

```
square :: Int -> Int
square n = n * n
```

with call by value

```
square (1 + 2)
= +
square 3
= square
3 * 3
= *
9
```

In contrast with call by name:

square (1 + 2)

= square

(1 + 2) * (1 + 2)

= first +

3 * (1 + 2)

= +

3 * 3

= *

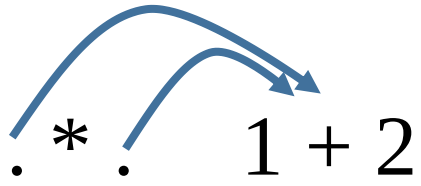
9

(1 + 2) is evaluated twice

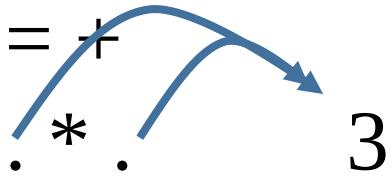
this duplicated computation can be avoided by using pointers to indicate sharing of expressions during evaluation

square (1 + 2)

= square



= +



= *

9

call by name + sharing of expressions = lazy evaluation

Infinite structures
ones :: [Int]
ones = 1 : ones

the computation of ones does not terminate,
head ones with call by value also does not terminate:
head (1 : ones)
head (1 : (1 : ones))
head (1 : (1 : (1: ones)))
.....

however, with call by name

head ones
= applying ones
head 1 : ones
=applying head
1

lazy evaluation evaluates ones, only as much as it is strictly necessary in order to produce result

Property: with lazy evaluation expressions are only evaluated as much as required by the context in which they are used

with this view ones is only a potentially infinite list which is only evaluated as much as required by the context

This does not apply only to lists, for instance, it applies also to trees

We consider Exercise 5 : consider

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
        deriving Show
```

Define for Tree a, appropriate versions of the following library functions for lists:

```
repeat :: a -> [a]
repeat x = xs where xs = x: xs

take :: Int -> [a] -> [a]
take 0 _ =  []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

replicate :: Int -> a -> [a]
replicate n = take n . repeat
```

```
repeatTree :: a -> Tree a
repeatTree x = Node t x t
where t = repeatTree x

takeTree :: Int -> Tree a -> Tree a
takeTree 0 _ = Leaf
takeTree _ Leaf = Leaf
takeTree n (Node l x r) = Node (takeTree (n-1) l) x (takeTree (n-1) r)

replicateTree :: Int -> a -> Tree a
replicateTree n = takeTree n . repeatTree
```

Modular Programmig (recall John Hughes)
Lazy evaluation allows to separate control from data

for example a list o 3 ones (control) can be produced from an infinite list of ones (data), using take of Exercise 5:

take 3 ones
[1,1,1]

but it’s interesting to look at this evaluation in detail

take 3 ones
= ones
take 3 (1: ones)
=take
1 : take 2 ones
=ones
1: take 2 (1 : ones)
=take
1: 1: take 1 ones
=ones
1 : 1 : take 1 (1 : ones)
=take
1 : 1 : 1: take 0 ones = take 1 : 1 : 1 : []

```
take :: Int -> [a] -> [a]
take 0 _ =  []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

ones is only evaluated when take needs it
take and ones evaluate in turns

without lazy evaluation a single function does both control and list

```
 replicate:: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

be aware:

filter (<= 5) [1..] loops forever

takeWhile (<=5) [1..]
[1,2,3,4,5]

strict application

```
sumwith :: Int -> [Int] ->Int
sumwith v []       = v
sumwith v (x:xs) = sumwith (v+x) xs

sumwith 0 [1,2,3]
=sumwith
sumwith (0+1) [2,3]
=sumwith
sumwith ((0+1)+2) [3]
=sumwith
sumwith(((0+1)+2)+3) []
```

this is the point

.....= 6

we want to compute the sum as soon as possible:

```
sumwith' v []     = v
sumwith' v (x:xs)= (sumwith' $! (v+x)) xs
```

$! = strict application

```
sumwith' 0 [1,2,3]
=(sumwith' $! (0+1)) [2,3]
=(sumwith' $! 1) [2,3]
= sumwith' 1 [2,3]
=(sumwith' $! (1+2)) [3]
=(sumwith' $! 3) [3]
=sumwith' 3 [3]
=....
```

more steps, but less space used

$! forces the top-level evaluation of the argument

if the argument is Int/Bool/.. => the constant
if it is a pair then the top-level evaluation stops as soon as it has shown that the argument has a value (_,_)
if it is a list then it stops as soon as the value is either [] or (_:_)

esempio: square x = x * x

square $! (1+2)
= +
square $! 3
= $!
square 3
=square
3 * 3
= *
9

```
in Data.Foldable
foldl’ :: (a -> b -> a) -> a -> [b] -> a
foldl’  f v []      = v
foldl’ f v (x:xs) = ((foldl’ f) $! (f v x)) xs

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []       = v
foldl f v (x:xs) = foldl f  (f v x)  xs


sumwith’ = foldl’ (+)
sumwith = foldl (+)
```

for functions with several parameters, like f x y, we can choose for each argument whether we want it to be strict or not,

```
(f $! x) y
(f  x) $! y
(f $! x) $!  y
```