UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 4.17
Haskell
Lazy Evaluation

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Part of these slides were adapted from the slides by

Adina Magda Florea
University Politehnica of Bucharest

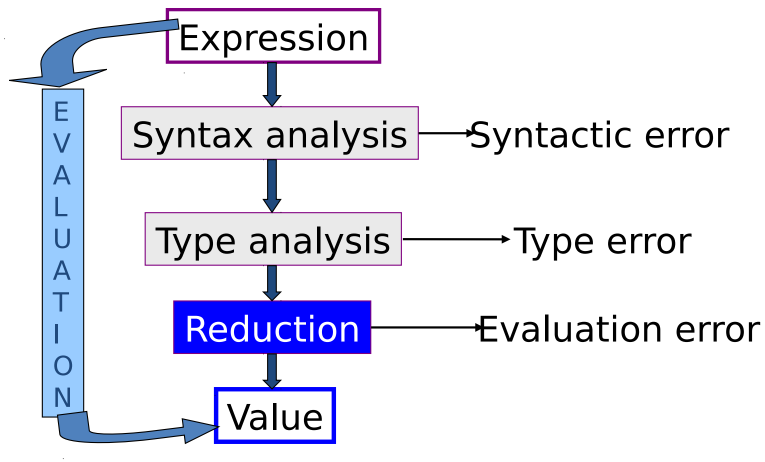
1. Reduction strategies and lazy evaluation

- Programming is not only about writing correct programs but also about writing fast ones that require little memory
- Aim: how Haskell programs are commonly executed on a real computer ⟹ a foundation for analyzing time and space usage
- Every implementation of Haskell more or less closely follows the execution model of *lazy evaluation*

1.1 Reduction

- Executing a functional program, i.e. evaluating an expression ⟹ means to repeatedly apply function definitions until all function applications have been expanded
- Implementations of modern **Functional Programming** languages are based on a simplification technique called **reduction**.

Evaluation in a strongly typed language



- The evaluation of an expression passes through three stages
- Only those expression which are syntactically correct and well typed are submitted for **reduction**

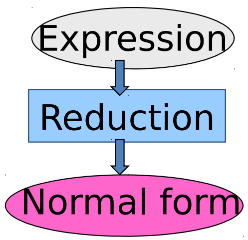
What is reduction?

- Given a set of rules $\mathbf{R_1, R_2, \ldots, R_n}$ (called reduction rules)
- and an expression **e**
- ***Reduction*** is the process of repeatedly simplifying **e** using the given reduction rules

$$
\begin{array}{lll}
e & \Rightarrow e_1 & R_{i1} \\
 & \Rightarrow e_2 & R_{i2} \\
\ldots & & \\
 & \Rightarrow \mathbf{e_k} & R_{ik}, \quad R_{ij} \in \{R_1, R_2, \ldots, R_n\}
\end{array}
$$

until no rule is applicable

- $\mathbf{e_k}$ is called the **normal form** of **e**
- It is the simplest form of **e**



2 types of reduction rules

- Built-in rules:

  addition, substraction, multiplication, division
- User supplied rules:

```
square x    = x * x
double x    = x + x
sum [ ]  = 0
sum (x:xs)  = x + sum xs
f x y       = square x + square y
```

1.2 Reduction at work

- Each reduction step replaces a subexpression by an equivalent expression by applying one of the 2 types of rules

```
f x y     = square x + square y
f 3 4  ⇒ (square 3) + (square 4)     (f)
       ⇒ (3*3) + (square 4)          (square)
       ⇒ 9 + (square 4)              (*)
       ⇒ 9 + (4*4)                   (square)
       ⇒ 9 + 16                      (*)
       ⇒ 25                          (+)
```

Reduction rules

- Every reduction replaces a subexpression, called **reducible expression** or **redex** for short, with an equivalent one, either by appealing to a function definition (like for square) or by using a built-in function like (+).
- *An expression without redexes is said to be in **normal form**.*
- The fewer reductions that have to be performed, the faster the program runs.
- We cannot expect each reduction step to take the same amount of time because its implementation on real hardware looks very different, but in terms of asymptotic complexity, this number of reductions is an accurate measure.

Alternate reductions

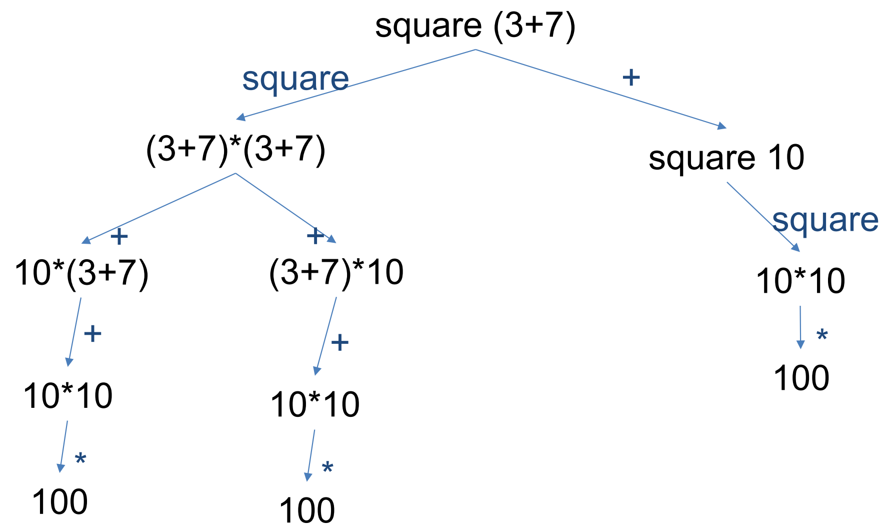
- square (3+7)
- square (10) (+)
- 10*10 (square)
- 100 (*)

normal form

- square (3+7)
- (3+7)*(3+7) (square)
- 10*(3+7) (+)
- 10*10 (+)
- 100 (*)

normal form

Possible ways of evaluating squares



Comments

- There are usually **several ways** for reducing a given expression to normal form
- Each way corresponds to a route in the evaluation tree: from the root (original expression) to a leaf (reduced expression)
- There are *three different ways* for reducing *square (3+7)*
- **Questions:**
  - Are all the answers obtained by following distinct routes identical?
  - Which is the best route?
  - Can we find an algorithm which always follows the best route?

Q&A

- **Q: Are all the values obtained by following distinct routes identical?**
- A: If two values are obtained by following two different routes, then these values must be identical

- **Q: Which is the best route?**
- Ideally, we are looking for the shortest route. Because this will take the least number of reduction steps and, therefore, is the most efficient.

Q&A

- **Q: Can we find an algorithm which always follows the best route?**
- In any tree of possible evaluations, there are usually two extremely interesting routes based on:
  - An Eager Evaluation Strategy
  - A Lazy Evaluation Strategy

1.3 Eager evaluation

- Given an expression such as:

  $f\ a$

  where f is a function and a is an argument.
- The Lazy Evaluation strategy reduces such an expression by attempting to **apply the definition of *f* first**.
- The Eager Evaluation Strategy reduces this expression by **attempting to simplify the argument *a* first.**

Example of Eager Evaluation

square (3 + 7)
= square (10)
= 10 * 10
= 100

By (+)
By square
By *

Reduction rules

Example of Lazy Evaluation

square (3 + 7)
= (3+7) * (3+7)
= 10 * (3+7)
= 10 * 10
= 100

By square
By +
By +
By *

Reduction rules

1.4 A&D of reduction strategies

- **LAZY EVALUATION** = Outer-most Reduction Strategy
  - Reduces **outermost redexes** = redexes that are not inside another redex.
- **EAGER EVALUATION** = Inner-most Reduction Strategy
  - Reduces **innermost redexes**
  - An **innermost redex** is a redex that has no other redex as subexpression inside.
- Advantages and drawbacks

UNIVERSITÀ DEGLI STUDI DI FERRARA

- EX LABORE FRUCTUS -

Capitolo 4.17

Haskell

Lazy evaluation: advantages & drawbacks

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Part of these slides were adapted from the slides by

Adina Magda Florea

University Politehnica of Bucharest

Repeated Reductions of Subexpressions

square (3+7)

Eager eval

= square 10

= 10 * 10

= 100

Lazy eval

= (3+7) * (3+7)

= 10 * (3+7)

= 10 * 10

= 100

- Eager Evaluation is better because it did not repeat the reduction of the subexpression *(3+7)!*

Repeated Reductions of Subexpressions

square (sum [1:100])

= square 5050

= 5050 * 5050

= 25502500

100 times +

square

*

= sum [1..100] * sum [1..100]

square

= 5050 * sum [1..100]

100 times +

= 5050 * 5050

100 times +

= 25502500

*

- Eager Evaluation requires 102 reductions
- Lazy Evaluation requires 202 reductions
- The Eager Strategy did not repeat the reduction of the subexpression *sum [1..100])!*

Performing Redundant Computations

`fst (2+2, square 15)`

- Lazy Evaluation
  - ⇒ 2 + 2 (fst)
  - ⇒ 4 (+)
- Eager Evaluation
  - ⇒ fst (4, square 15)(+)
  - ⇒ fst(4, 15*15) (square)
  - ⇒ fst(4, 225) (*)
  - ⇒ 4 (fst)

- Lazy evaluation is better as it avoids performing redundant computations

Termination

- For some expressions like `loop = 1 + loop` no reduction sequence may terminate; they do not have a normal form.
- But there are also expressions where some reduction sequences terminate and some do not
- `fst (5, loop)`
- Lazy Evaluation
  - ⇒ 5 (fst)
- Eager Evaluation
  - ⇒ fst(5, loop) (loop)

- Lazy evaluation is better as it avoids infinite loops in some cases

Run-time errors

- Same for errors
- `fst (5, 1 / 0)`
- Lazy Evaluation
  - ⇒ 5 (fst)
- Eager Evaluation
  - ⇒ fst(5, ERROR) (/)
  - ⇒ ERROR (Attempts to compute 1/0)

- Lazy evaluation is better as it avoids run-time errors in some cases

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 4.18
Haskell
Graph Reduction

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Part of these slides were adapted from the slides by

Adina Magda Florea
University Politehnica of Bucharest

Eager evaluation

- **Advantages:**
  - Repeated reductions of sub-expressions is avoided.
- **Drawbacks:**
  - Have to evaluate all the parameters in a function call, whether or not they are required to produce the final result.
  - It may not terminate.

Lazy evaluation

- **Advantages:**
  - A sub-expression is not reduced unless it is absolutely essential for producing the final result.
  - If there is any reduction order that terminates, then Lazy Evaluation will terminate.
- **Drawbacks:**
  - The reductions of some sub-expressions may be unnecessarily repeated.

Duplicated Reduction of Subexpressions

- The reduction of the expression (3+4) is duplicated when we attempt to use lazy evaluation to reduce

  square (3+4)

- This problem arises for any definition where a variable on the left-hand side appears more than once on the right-hand side.

  square x = x * x

  cube x = x * x * x

1.5 Graph Reduction

- Aim: Keep All good features of Lazy Evaluation and at the same time avoiding duplicated reductions of sub-expressions.
- Method: By representing expressions as graphs so that all occurrences of a variable are pointing to the same value.

Graph Reduction

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

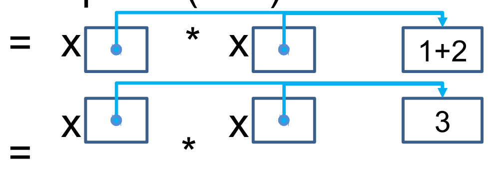
- Lazy evaluation

  square (1+2)
  = (1+2)*(1+2)

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

- Graph reduction

  square (1+2)



  = 9

Graph Reduction Strategy combines all the benefits of both Eager and Lazy evaluations with none of their drawbacks.

Graph Reduction

- The **outermost graph reduction** of

  square (3 + 4)

  now reduces every argument at most once.
- For this reason, it always takes fewer reduction steps than the innermost reduction
- Sharing of expressions is also introduced with **`let`** and **`where`** constructs.

Graph Reduction for let

Heron's formula for the area of a triangle with sides *a*, *b* and *c*

```
area a b c = let s = (a+b+c)/2 in
     sqrt (s*(s-a)*(s-b)*(s-c))
```

```
area 1 1 1
= sqrt ( *( -1)*( -1)*( -1))        s (1+1+1)/2
= sqrt ( *( -1)*( -1)*( -1))          1.5
…
= sqrt ( *0.5*0.5*0.5)                1.5
…
= sqrt 0.1875
= 0.4330127018922193
```

- Let-bindings simply give names to nodes in the graph

Graph Reduction

- Any implementation of Haskell is in some form based on **outermost graph reduction** which thus provides a good *model for reasoning about the asymptotic complexity of time and memory allocation*
- The number of reduction steps to reach normal form corresponds to the execution time and the size of the terms in the graph corresponds to the memory used.

Reduction of higher order functions and currying

```
(.) :: (b->c)->(a->b)->(a->c)
f . g = \x -> f (g x)
```

```
id x = x
a     = id (+1) 41
twice f = f . f
b         = twice (+1) (13*3)
```

where both `id` and `twice` are only defined with one argument.

- The solution is to see multiple arguments as subsequent applications to one argument - **currying**

Reduction of higher order functions and currying

```
id x   = x
a   = id (+1) 41
twice f   = f . f
b   = twice (+1) (13*3)
```

- **Currying**

```
a = (id (+1)) 41
b = (twice (+1)) (13*3)
```

- To reduce an arbitrary application

  expression$_1$ *expression*$_2$

  call-by-need first reduces *expression*$_1$ until this becomes a function whose definition can be unfolded with the argument *expression*$_2$.

Reduction of higher order functions and currying

```
a = (id (+1)) 41
```

```
id x   = x
```

a
$\Rightarrow$ (id (+1)) 41 (a)
$\Rightarrow$ (+1) 41 (id)
$\Rightarrow$ 42 (+)

Reduction of higher order functions and currying

```
twice f   = f . f
```

```
b = (twice (+1)) (13*3)
```

b
$\Rightarrow$ (twice (+1)) (13*3) (b)
$\Rightarrow$ ((+1).(+1)) (13*3) (twice)
$\Rightarrow$ (+1) ((+1) (13*3)) (.)
$\Rightarrow$ (+1) ((+1) 39) (*)
$\Rightarrow$ (+1) 40 (+)
$\Rightarrow$ 41 (+)

Reduction of higher order functions and currying

- Functions are useful as data structures.
- In fact, all data structures are represented as functions in the pure lambda calculus, the root of all functional programming languages.

Capitolo 4.19

Haskell

Lazy Evaluation: Memory Leaks

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI


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

sum1 = foldr (+) 0
```

```
lista i s
| i == s     = [i]
| otherwise =  i:lista i+1 s
```

```
sum1 [1..1e8]
foldr (+) 0 [1..1e8]
1+ (foldr (+) 0 [2..1e8])
1+ (2+ (foldr (+) 0 [3..1e8]))
1+ (2+ (3+ (foldr (+) 0 [4..1e8])))
1+ (2+ (3+ (4+ (foldr (+) 0 [5..1e8]))))
```

stack overflow

also called "memory leak"

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

sum2 = foldl (+) 0
```

```
sum2 [1..1e8]
foldl (+) 0 [1..1e8]
foldl (+) (0 + 1) [2..1e8]
foldl (+) ((0 + 1) + 2) [3..1e8]
foldl (+) (((0 + 1) + 2) + 3) [4..1e8]
foldl (+) ((((0 + 1) + 2) + 3) + 4)
  [5..1e8]
```

stack overflow

Forcing evaluation

```
seq :: a -> b -> b
```

- `seq` is a primitive system function
- `seq x y`
  will first evaluate `x` then return `y`.
- If `y` references `x`, when `y` is reduced `x` will not be an unreduced chain anymore.

```
foldl' f z []     = z
foldl' f z (x:xs) = let z' = f z x
                    in seq z' (foldl' f z' xs)

sum3 = foldl' (+) 0
```

```
sum3 [1..1e8]
foldl' (+) 0 [1..1e8]
foldl' (+) 1 [2..1e8]
foldl' (+) 3 [3..1e8]
foldl' (+) 6 [4..1e8]
foldl' (+) 10 [5..1e8]
…
```

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 4.20
Haskell
Programming with Lazy Evaluation

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Logical 'AND'

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
False && False = False
False && True  = False
True  && False = False
```

False && ((4*2 + 34) == 42)

Logical 'AND'

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

False && ((4*2 + 34) == 42)

- immediately reduces to False.
- Same as short-circuit evaluation in imperative languages, but that is hard-coded in the language!

'AND' of lists

```
and = foldr (&&) True
```

and [10==3,5>2,True,False, 10/2==5]

- immediately evaluates to False

Compare strings

```
prefix xs ys = and (zipWith (==) xs ys)
```

```
and [] = True
and (z:zs) = z && (and zs)
```

```
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) =
     (f x y):zipWith f xs ys
```

```
prefix "Haskell" "eager"
=>  and (zipWith (==) "Haskell "eager")
=>  and ('H' == 'e' : zipWith (==) "askell" "ager")
=>  'H' == 'e' && and (zipWith (==) "askell" "ager")
=>  False && and (zipWith (==) "askell" "ager")
=>  False
```

stops as soon as the first differing character is found

minimum of a list

- Shortest way to define the minimum

```
minimum = head . sort
```

- to use the predefined `sort` function:

```
import Data.List
```

- Otherwise, can be defined as

```
qsort []     = []
qsort (x:xs) =
  qsort [y|y<-xs,y<x] ++ [x] ++ qsort [z|z<-xs,z>=x]
```

```
head (qsort [6,9,4,3,5])
head ((qsort [y|y<-[9,4,3,5],y<6])++[6]++(qsort [y|y<-[9,4,3,5],y>=6]))
head ((qsort [y|y<-[4,3,5],y<6])++[6]++(qsort [y|y<-[9,4,3,5],y>=6]))
head ((qsort (4:[y|y<-[3,5],y<6]))++[6]++(qsort [y|y<-[9,4,3,5],y>=6]))
head ((qsort [y'|y'<-[y|y<-[3,5],y<6],y'<4]++[4]++qsort [y'|y'<-[y|y<-[3,5],y<6],y>=4
head ((qsort [y'|y'<-(3:[y|y<-[5],y<6]),y'<4]++[4]++qsort [y'|y'<-[y|y<-[3,5],y<6],y>=
head ((qsort (3:[y'|y'<-[y|y<-[5],y<6],y'<4]++[4]++qsort [y'|y'<-[y|y<-[3,5],y<6],y>=
head ((qsort [y"|y"<-[y'|y'<-[y|y<-[5],y<6],y'<4],y"<3]++[3]++(qsort [y"|y"<-[y'|y'<-[y
head ((qsort [y"|y"<-[y'|y'<-(5:[y|y<-[],y<6]),y<4],y"<3]++[3]++(qsort [y"|y"<-[y'|y'<-
head ((qsort [y"|y"<-[y'|y'<-[y|y<-[],y<6],y<4],y"<3]++[3]++(qsort [y"|y"<-[y'|y'<-[y|y
head ((qsort [y"|y"<-[y'|y'<-[],y<4],y"<3]++[3]++(qsort [y"|y"<-[y'|y'<-[y|y<-[5],y<6],y
head ((qsort [y"|y"<-[],y"<3]++[3]++(qsort [y"|y"<-[y'|y'<-[y|y<-[5],y<6],y<4],y">=3])
head ((qsort []++[3]++(qsort [y"|y"<-[y'|y'<-[y|y<-[5],y<6],y<4],y">=3])++[4]++qsort
head ([]++[3]++(qsort [y"|y"<-[y'|y'<-[y|y<-[5],y<6],y<4],y">=3])++[4]++qsort [y'|y'<
head ([3]++(qsort [y"|y"<-[y'|y'<-[y|y<-[5],y<6],y<4],y">=3])++[4]++qsort [y'|y'<-[y|y
3
```

10 smallest elements of a list

- Define a function **`tenSmallest`** that provides the 10 smallest elements of a list

Università degli Studi di Ferrara
- EX LABORE FRUCTUS -

Capitolo 4.20
Haskell
Lazy Evaluation and Infinite Lists

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Infinite lists

```
>[1..]
[1,2,3,4,5,6,…
```

- Given a list **`xs`**, create a list of pairs **`(p,x)`** where **`x`** is in **`xs`** and **`p`** is its position in **`xs`**

```
>pairs "Haskell"
 [(1,'H'),(2,'a'),(3,'s'),(4,'k'),
(5,'e'),(6,'l'),(7,'l')]
```

Modular programming

- Generare una lista contenente **n** volte un valore x

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

Modular programming

- Generare una lista contenente **n** volte un valore **x**

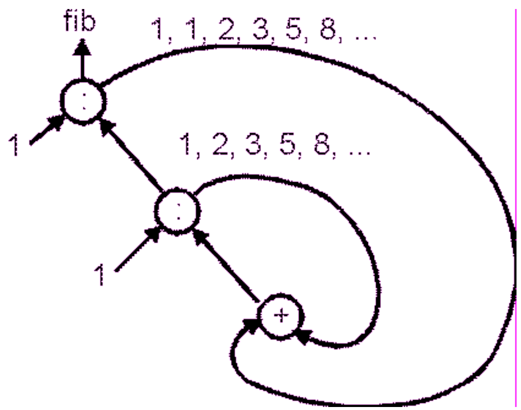
```
valore x = x:valore

take n (valore x)
```

La lazy evaluation permette di rendere i programmi più modulari, separando il controllo dai dati.

La parte di dati (`valore x`) viene valutata solo per quanto richiesto dalla parte di controllo (`take n`)

Fibonacci

```
fib = 1:1:[a+b| (a,b) <- zip fib (tail fib) ]
```



iterate

- The Prelude function **iterate** can be used to generate an infinite list of values

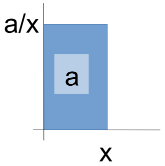
  [x, f x, f (f x), …]

  by repeatedly applying a function **f**:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Heron's method - square root

- Given an approximation **x** to the square root of the number **a**, a function `better` calculates a better approximation to the square root by taking an average




```
better a x = 1/2*(x + a/x)
```

- We stop if two adjacent values in the list are within some ε:

```
within eps (x:y:ys)
     | abs (x-y) <= eps = y
     | otherwise = within eps (y:ys)
```

```
squareRoot a = within 1e-9 (iterate (better a) 1)
```

User defined control structures

- Define a function

```
ifPzn x pos zero neg
```

  that evaluates to
  – pos if x>0
  – zero if x=0
  – neg if x<0
- then write a function `solve a b c` that provides a list of the solutions of the equation

$$ax^2+bx+c=0$$

Esercizio: merge

- Scrivere una funzione **`merge`** che, date due liste ordinate (anche infinite) fornisce la lista ordinata (eventualmente infinita) che contiene gli elementi di entrambe le liste
- Es:

```
> merge [2,4..] [5,10..]
[2,4,5,6,8,10,12,14,15,16,18,20,
 22,24,25,26,28,30,32,34,...
```

Exercise

A well-known problem, due to the mathematician W.R. Hamming, is to write a program that produces an infinite list of numbers with the following properties:

1. the list is in strictly increasing order;
2. the list begins with the number 1;
3. if the list contains the number *x*, then it also contains the numbers 2*x*, 3*x* and 5*x*;
4. the list contains no other numbers.

Thus, the required list begins with the numbers

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, . . .

Write a definition of **`hamming`** that produces this list.

Suggestion: Write a function merge that merges two sorted lists into a sorted list

```
""
"0"
"1"
"00"
"10"
"01"
"11"
"000"
"100"
"010"
"110"
"001"
"101"
"011"
"111"
...
```

Kleene closure

- The Kleene closure of a set S is the set of all strings with S as the alphabet. It's usually written as S*. For example, the Kleene closure of S={0,1} is given on the left.
- Write a function

  `kleene :: [a] -> [[a]]`

  that generates the Kleene closure of the set **`[a]`**
- Example

```
take 5 (kleene "01")
["","0","1","00","10"]
```

Crivello di Eratostene

- Il crivello di Eratostene serve a calcolare la sequenza dei numeri primi, partendo dalla sequenza di tutti i numeri >=2.
- Si parte da 2: riportiamo il 2 nella lista di uscita ed eliminiamo dalla lista di ingresso tutti i multipli di 2
- Riportiamo in uscita il primo numero rimasto (che a questo punto è il 3) e togliamo tutti i suoi multipli
- Riportiamo il primo numero rimasto (5) e togliamo tutti i suoi multipli
- …

```
*Main> crivello [2..]
[2,3,5,7,11,13,17,19,23,29,31,37,…
```

Infinite pairs

- You want to produce an infinite list of all distinct pairs (x, y) of natural numbers.
- It doesn't matter in which order the pairs are enumerated, as long as they all are there. Say whether or not the definition

  `allPairs= [(x,y)| x<-[0..], y<-[0..]]`

  does the job. If you think it doesn't, can you give a version that does?
- Suggerimento: in quale posizione della lista è l'elemento (2,1)?

Esercizio di esame

- http://www.unife.it/ing/lm.infoauto/linguaggi-e-traduttori/testi-di-esame/23-giugno-2016-laboratorio
- Per ottenere il massimo dei punti, si gestiscano le liste infinite