Lecture 5

Lazy evaluation.
Introduction to Haskell

Mircea Marin
West University of Timișoara
mircea.marin@e-uvt.ro

Computation (in FP) = evaluation
= sequence of reduction steps that replace a **redex** with the result of applying a **rule of reduction**.
It **stops** when we reach a **value**.

- The most common redexes are **function calls**, also known as β-**redexes**. They are reduced with the rule of β-reduction

$$\underbrace{\lambda(x_1\ \ldots\ x_n).block\ t_1\ \ldots\ t_n}_{\beta\text{-redex}} \rightarrow \underbrace{[t_1/x_1,\ldots,t_n/x_n]block}_{\text{capture-free substitution}}$$

- The redexes that are not function calls are called **special forms**. Every special form has its own rule of reduction, which must be learned separately, from the language specification.

- (if v t_1 t_2) $\rightarrow \begin{cases} t_1 & \text{if } v \text{ is a true value,} \\ t_2 & \text{if } v \text{ is value \#f.} \end{cases}$
- (or t_1 ... t_n) $\rightarrow \begin{cases} \#f & \text{if all } t_i\text{s have value \#f,} \\ v_i & \text{if } v_i \text{ is the first true value of a } t_i. \end{cases}$
- (and t_1 ... t_n) $\rightarrow$ ...
- (let ([x_1 t_1] ... [x_n t_n]) *block*) $\rightarrow$ ...
- (cond [$test_1$ $block_1$] ... [$test_n$ $block_n$]) $\rightarrow$ ...

Remarks

- Some special forms are syntactic sugar
  - The preprocessor of the language translates them (before compilation) into equivalent forms, that produce same result
  - Syntactic sugar is easier to write than the equivalent forms
- The other special forms should be as few as possible, to avoid learning too many rules of reduction.

Often, there are many redexes ⇒ many ways to compute the same value. Also, some choices can produce infinite computations.

Example

```
(define (f x) (cons x (f (+ x 1))))
```

Remark: `(f 3)` will run forever, trying to compute the infinite list

`'(3 4 5 6 7 8 ...)`

There are many ways to evaluate `(+ (+ 1 2) (car (f 3)))`, where `(+ 1 2)` is a redex and `(f 3)` is a redex:

```
(+ (+ 1 2) (car (f 3))) → (+ 3 (car (f 3)))
  → (+ 3 (car (cons 3 (f 4)))) → (+ 3 3) → 6
(+ (+ 1 2) (car (f 3))) → (+ (+ 1 2) (car (cons 3 (f 4))))
  → (+ 3 (car (cons 3 (f 4)))) → (+ 3 (car (cons 3 (cons 4 (f 5)))))
  → (+ 3 3) → 6
(+ (+ 1 2) (car (f 3)))→ (+ 3 (car (f 3))) → (+ 3 (car (cons 3 (f 4))))
  → (+ 3 (car (cons 3 (cons 4 (f 5))))) → ...        runs forever
```

Evaluation strategies

Programming languages implement only one way to compute a value, called evaluation strategy. The most popular evaluation strategies are:

- **Strict** (or call-by-value) evaluation: A function call is reduced only after the function arguments are reduced to values.
  - ⇒ the selected redex is the leftmost innermost (but not in the body of a function definition)

  Racket performs strict evaluation.
- **Lazy** (or call-by-name) evaluation: A function call is reduced as soon as the arguments contain enough information to perform β-reduction.
  Call-by-need evaluation is an optimized implementation of lazy evaluation, which reduces all duplicates of a redex only once (see also Lecture 2).
  - **Intuition:** expression are evaluated **on demand**, until they contain the information needed to compute the overall result.

  Haskell performs call-by-need evaluation.

We will practice lazy functional programming with Haskell.

- Download Haskell for your own platform (Windows, Linux or Mac OS X) from https://www.haskell.org/platform/

The platform includes GHCi, which allows to

- interactively evaluate Haskell expressions
- interpret Haskell programs
- load GHC-compiled modules

To start a GHCi session, type `ghci` at the command prompt:

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

- You will learn at labs 5 and 6 how to use GHCi to interact with Haskell.
- We will explain the important differences between Racket and Haskell

A function call $f(arg_1 \ \ldots \ arg_b)$ is written as

- $(f \ arg'_1 \ \ldots \ arg'_n)$ in Racket
- $f \ arg'_1 \ \ldots \ arg'_n$ in Haskell

A list with elements $e_1, \ldots, e_n$ is written as $[e_1, \ldots, e_n]$ in Haskell

- All elements $e_1, \ldots, e_n$ must have same type
- The empty list is []

Some binary functions, such as '+' are written in infix syntax. between their arguments (compare $x + y$ with $f \ x \ y$).

- Infix functions are called operators. Their names do not contain any numbers or letters of the alphabet.
- To avoid using many parentheses, most operators have predefined **precedence** and **associativity** rules. E.g., we write

  `x+y+z` instead of `(x+y)+z` because `+` is left associative
  `x+y*z` instead of `x+(y*z)` because `*` has higher precedence

Other rules of disambiguation

- Function application has higher priority than operator application. Example: $f\ x + g\ y$ is parsed as $(f\ x) + (g\ y)$
- Function application is left-associative: $f\ x\ y\ z$ is parsed as $(((f\ x)\ y)\ z)$.
- Operator application $x\ op\ y$ can be converted into function application, by writing $(op)\ x\ y$.

  Examples:

  `3 + 4` $\rightarrow$ 7

  `(+) 3 4` $\rightarrow$ 7
- Binary function application $f\ x\ y$ can be converted into operator application, by writing x ‘f‘ y.
  - Example. `mod` is a predefined binary function: `mod m n` returns the remainder of dividing integer `m` by integer `n`.

    `mod 8 3` $\rightarrow$ 2

    `8 ‘mod‘ 3` $\rightarrow$ 2
- A short list of useful predefined functions and operators can be found here.

Types

Every expression has an associated **type**. The following types are predefined:

- `Bool` – the type of Boolean values `True` and `False`
  `Int` – fixed precision integers between -2^{29} and 2^{29-1}
  `Integer` – arbitrary precision integers
  `Char` – characters, like `'a'`, `'A'`, `'!'`, `','`, `'z'`, `'Z'`
  `Float` – floating-point numbers with single-word precision
  `Double` – floating-point numbers with double-word precision

The most important **composite types** are lists and tuples:

- If T is a type, then `[`T`]` is the type of lists `[`$v_1, \ldots, v_n$`]` with elements $v_1, \ldots, v_n$ of type T. The empty list is `[]`.
  EXAMPLE: `[1,2,3,4]` is a list of type `[Int]`.
- If $T_1, \ldots, T_n$ are types, then `(`$T_1, \ldots, T_n$`)` is the type of tuples `(`$v_1, \ldots, v_n$`)` with v_1 of type T_1, ..., v_n of type T_n.
  EXAMPLE: `[(1,'A'),(2,'x')]` is a list of tuples; it has type `[(Int,Char)]`

Types and type constructors

- The values of simple types. like `Int`, `Char` and `Float`, are **literals**. Examples of literals: `1, 'A', 3.14`
- The values of composite types are built by applying **data constructors** to component values.
  - The constructors of lists are `[]`, _:_, and `[...]`
  - The constructor of tuples is `(...)`

REMARKS.

- `True` and `False` are nullary data constructors.
- Data constructors are a special kind of functions: they are used to build composite values.
- In Haskell, the names of data constructors can not start with a lowercase letter.
- Users can define their own composite types.

More about lists

The operator ':' is a right-associative **data constructor** for lists

- $x : xs$ is the list obtained by adding x in front of list xs

  REMARK: $[x_1, x_2, \ldots, x_n]$ is syntactic sugar for $x_1 : x_2 : \ldots : x_n : []$

The following operations on lists are **predefined**:

- $xs ++ ys$ appends lists xs and ys.
- `head xs` returns first element of `xs`, and `tail xs` returns the tail of list `xs`.
- `length` xs computes the length of list xs.
- `reverse` xs reverses list xs.
- `take n xs` returns the list of first `n` elements of list `xs`. If `xs` has less than `n` elements, it returns `xs`.

A string coincides with the list of its component characters. For example, we can write (and see) `"abc"` instead of `['a','b','c']`.

Strings have type `[Char]`

Definitions

A Haskell definition gives a name (or identifier) to an expression of a particular type.

```
name :: type          -- declare name of type type
name = expression     -- creates a binding of name to expression
```

Example:

```
x,y :: Int            -- declare x,y of type Int
x = 12 + 13
y = y + 1             -- example of a recursive binding
```

- comments start with '--' and are ignored by the compiler
- If we omit type declarations, Haskell tries to infer the type of *name* from the type of *expression*
  - there are very few cases when this is impossible.
- *expression* is **not** evaluated: the environment stores a binding of *name* to *expression*.

- If T_1, T_2 are types then T_1 -> T_2 is the type of functions which map inputs of type T_1 to results of type T_2.
  REMARK: T_1 -> T_2 -> ... -> T_n -> T is parsed as
  T_1 -> (T_2 -> (... -> (T_n -> T)...))
- We can define $f = \lambda x_1. \cdots .\lambda x_n.expr$ where every x_i has type type T_1 and the result has type T, by writing
  f :: T_1 -> ... -> T_n -> T
  f x_1 ... x_n = $expr$

Example (A function to compute the area of a rectangle)

```
rectArea :: Float -> Float -> Float
rectArea x y = x * y
```

binds rectArea to $\lambda\mathrm{x} :: \mathtt{Float}.\lambda\mathrm{y} :: \mathtt{Float}.(x * y)$

rectArea 3 4 = $\underline{\lambda x.\lambda y.(x * y)\ 3}\ 4 \to \underline{\lambda y.(3 * y)\ 4}$
$\to \underline{3{*}4} \to 12$

Both Racket and Haskell allow us to work with lambda expressions, but with different syntax. For example $\lambda x.\lambda y.(x * y)$ is written

```
(lambda (x) (lambda (y) (+ x y)))   in Racket
\x y -> x*y                         in Haskell
```

REMARK. In Haskell, `\x y z ->` *expr* is shorthand for

`\x -> \y -> \z ->` *expr*

Example

```
f :: Float -> Float -> Float
f = \x y -> x*y          -- same as f x y = x*y
f 3 4 =  (\x y -> x*y) 3 4
      →  (\y -> 3*y) 4
      →  3*4
      →  12
```

Patterns

Pattern = expression defined by the grammar

patt ::= _ | *variable* | *literal* | *C* $patt_1$ ... $patt_n$

where *C* is a data constructor with arity *n* and every variable occurs at most once.
'_' is called *anonymous variable*.

Examples of patterns

```
(x,y,z)      -- pattern for tuples with 3 components
True:xs      -- pattern for list of Booleans starting with True
[(1,'c'),_]  -- pattern for list of two tuples of type (Int,Char)
             -- starting with tuple (1,'c')
```

The following are not patterns:

```
(x,x)          -- variable x occurs twice
length x:xs    -- length x is not data constructor
```

Pattern matching

We can try to match pattern *patt* with a value *v*. The matching attempt can fail or succeed. If it succeeds, we get

> a substitution, called matcher, that binds the variables in *patt* to component values from *v*.
>
> - the anonymous variable '_' matches *any* value.

Pattern *patt*	Value *v*	match(*patt*, *v*)
`(x,y)`	`(1,True)`	`[1/x,True/y]`
`(x,_):(_,y):z`	`[(1,2),(3,4)]`	`[1/x,4/y,[]/z]`
`_:_`	`[]`	fail
`[_,x,_]`	`[1,2,3]`	[2/x]

Pattern matching

We can try to match pattern *patt* with a value *v*. The matching attempt can fail or succeed. If it succeeds, we get

> a substitution, called matcher, that binds the variables in *patt* to component values from *v*.
>
> - the anonymous variable '_' matches *any* value.

Pattern *patt*	Value *v*	match(*patt*, *v*)
`(x,y)`	`(1,True)`	`[1/x,True/y]`
`(x,_):(_,y):z`	`[(1,2),(3,4)]`	`[1/x,4/y,[]/z]`
`_:_`	`[]`	fail
`[_,x,_]`	`[1,2,3]`	[2/x]

Modern functional programming languages, including Haskell, allow us to define functions with pattern matching (see next.)

Function definitions by pattern matching

A function $f::T_1\texttt{->}\ldots\texttt{->}T_n\texttt{->}T$ can be defined by $k \geq 1$ equations

$f \;\; patt_{1,1} \;\; \ldots \;\; patt_{1,n} \;\; \texttt{=} \;\; expr_1$

$\ldots$

$f \;\; patt_{k,1} \;\; \ldots \;\; patt_{k,n} \;\; \texttt{=} \;\; expr_n$

which satisfy the condition that every

$(patt_{i,1}, \ldots, patt_{i,n})$ can match a value of type $(T_1, \ldots, T_n)$

How do we evaluate $(f \;\; expr_1 \;\; \ldots \;\; expr_m)$ for $m \leq n$?

for i from 1 to k
  reduce $expr_1 \to expr_1'$, ..., $expr_m \to expr_m'$ until
    $[\theta] = match((patt_{i,1}, \ldots, patt_{i,m}),(expr_1', \ldots, expr_m'))$ succeeds or fails
  if $[\theta] = \text{fail}$
    continue
  else
    reduce $(f \;\; expr_1 \;\; \ldots \;\; expr_m) \to [\theta]expr_i$
    break

This kind of computation is called call-by-need (or lazy) reduction.

1. A function to concatenate two lists (it does the same thing as the operator ++):

```
app []     ys = ys
app (x:xs) ys = x:(app xs ys)
```

2. A function to get the n-th element of a list:

```
nth 1 (x:_)  = x
nth n (_:xs) = nth (n-1) xs
```

3. A function that computes the infinite list $[n, n+1, n+2, \ldots]$ for an integer n:

```
intsFrom::Integer->[Integer]
intsFrom n = n:intsFrom (n+1)
```

4. The infinite list of natural numbers, starting from 1:

```
nats = intsFrom 1
```

Function definitions with guards

A definitional equation of the form

```
f patt1 ... pattn
  = if test1
    then expr1
    else if test2
         then expr2
         else if ...
```

can be rewritten in the more readable form

```
f patt1 ... pattn
  | test1      = expr1
  | test2      = expr2
  ...
  | otherwise  = exprn
```

The blue-colored parts are called **guards**.

REMARK. Indentation is important in Haskell: indent with the same amount!

Examples of lazy evaluation

- Computing the infinite list of natural numbers:

  `nats = intsFrom 1` $\to$ `1:intsFrom 2`

  $\to$ `1:2:intsFrom 3` $\to$ `...`

  - never ending computation
  - GHCi displays the list elements, as they are are generated progressively (on demand)
- Compute the second element of `nats`:

  `nth 2 nats = nth 2 intsFrom 1` -- reduction on demand

  $\to_{[1/\mathtt{n}]}$ `nth 2 (1:intsFrom 2)`

  $\to_{[2/\mathtt{n},(\mathtt{intsFrom\ 2})/\mathtt{xs}]}$ `nth 1 intsFrom 2` -- red. on demand

  $\to_{[2/\mathtt{n}]}$ `nth 1 (2:intsFrom 3)`

  $\to_{[2/\mathtt{x}]}$ 2

Remarks

- **Lazy languages** allow us to define and work with infinite data structures (e.g., `nats`), because reduction is on demand
- **Strict languages** (e.g., Racket) try to compute the complete values of function arguments $\Rightarrow$ nonterminating reductions.

More examples

In lazy languages, many special forms can be defined as functions that are evaluated on demand. For example:

1. `if` is a special form in Racket, but in Haskell we can define it as a function:

   ```
   if'::Bool->a->a->a
   if' True  x _ = x
   if' False _ y = y
   ```

   REMARK: `if'` has a polymorphic type: the branches and result of `if'` must have same type, which can be *any* type a.

2. A function definition of boolean operator && for conjunction:

   ```
   and False _ = False
   and True  x = x
   ```

3. The Boolean operator `||` for disjunction is a special form in Racket, but we can define it as a function in Haskell (how?).

Quiz: Use Haskell to define the infinite list `fib`=$[f_1, f_2, f_3, \ldots]$ of Fibonacci numbers, where $f_1 = f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$ if $n > 2$. Use the fact that, if we add componentwise `fib` with `tail fib` we obtain

$$
\begin{array}{rllllll}
\texttt{fib} = [& f_1, & f_2, & f_3, & f_4, & \ldots &] \quad + \\
\texttt{tail fib} = [& f_2, & f_3, & f_4, & f_5, & \ldots &] \\
\hline
[& f_3, & f_4, & f_5, & f_6, & \ldots &] \quad = \texttt{tail (tail fib)}
\end{array}
$$

Note that `tail` is a predefined function in Haskell.

Quiz: Use Haskell to define the infinite list `fib`=$[f_1, f_2, f_3, \ldots]$ of Fibonacci numbers, where $f_1 = f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$ if $n > 2$. Use the fact that, if we add componentwise `fib` with `tail fib` we obtain

$$
\begin{array}{rlllllll}
\texttt{fib} = [& f_1, & f_2, & f_3, & f_4, & \ldots &] & + \\
\texttt{tail fib} = [& f_2, & f_3, & f_4, & f_5, & \ldots &] & \\
\hline
[& f_3, & f_4, & f_5, & f_6, & \ldots &] & = \texttt{tail (tail fib)}
\end{array}
$$

Note that `tail` is a predefined function in Haskell.

Haskell solution:

```
-- this auxiliary function adds componentwise
-- two infinite lists of numbers
addLists :: [Integer] -> [Integer] -> [Integer]
addLists (x:xs) (y:ys) = (x+y):addLists xs ys
fib::[Integer]
fib = 1:1:addLists fib (tail fib)
```

Finding the *n*-th Fibonacci number

```
nthFib n = nth n fib
```

Example (Computation of the 3-rd Fibonacci number)

```
nthFib 3 →[3/n,1:1:(addLists fib (tail fib))/fib]
 nth 3 1:1:addLists fib (tail fib)
 →[3/n,1:addLists fib (tail fib)/xs] nth 2 1:addLists fib (tail fib)
 →[2/n,addLists fib (tail fib)/xs] nth 1 addLists fib (tail fib)
 = nth 1 addLists (1:1:addLists fib (tail fib))
                  tail (1:1:addLists fib (tail fib))
 → nth 1 addLists (1:1:addLists fib (tail fib))
                  (1:addLists fib (tail fib))
 → nth 1 2:addLists (1:addLists fib (tail fib))
                    addLists fib (tail fib)
 → 2
```

Higher-order functions on lists

1.
```
-- map has definition like in Racket
map::(a->b)->[a]->[b]
map _ []     = []
map f (x:xs) = (f x):(map f xs)
```

2.
```
filter::(a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if (p x)
                  then filter p xs
                  else x:filter p xs
```

3.
```
-- foldl f v lst behaves like
-- (foldl (lambda x y) (f y x) v lst) in Racket
foldl::(b -> a -> b) -> b -> [a] -> b
foldl _ v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
-- foldr behaves like in Racket
foldr::(a -> b -> b) -> b -> [a] -> b
foldr f v lst = foldl (\x y->f y x) v (reverse lst)
```

Higher-order functions on lists

1.
```
-- map has definition like in Racket
map::(a->b)->[a]->[b]
map _ []     = []
map f (x:xs) = (f x):(map f xs)
```
2.
```
filter::(a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if (p x)
                  then filter p xs
                  else x:filter p xs
```
3.
```
-- foldl f v lst behaves like
-- (foldl (lambda x y) (f y x) v lst) in Racket
foldl::(b -> a -> b) -> b -> [a] -> b
foldl _ v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
-- foldr behaves like in Racket
foldr::(a -> b -> b) -> b -> [a] -> b
foldr f v lst = foldl (\x y->f y x) v (reverse lst)
```

Remarks

In Haskell, all functions have a fixed arity ⇒ there is no function equivalent to `apply`.

Another example: Hamming numbers

A Hamming number is of the form $2^i 3^j 5^k$ where i, j, k are non-negative integers. The first five Hamming numbers are:

$$1 = 2^0 3^0 5^0 \quad 2 = 2^1 3^0 5^0 \quad 3 = 2^0 3^1 5^0 \quad 4 = 2^2 3^0 5^0 \quad 5 = 2^0 3^0 5^1$$

Quiz: Generate the list `ham` of all Hamming numbers in ascending order. Make use of the following observations:

1. The list starts with 1.
2. Every Hamming number $h > 1$ is of the form $a \cdot h'$ where $a \in \{2, 3, 5\}$ and h' is a Hamming number

Another example: Hamming numbers

A Hamming number is of the form $2^i 3^j 5^k$ where i, j, k are non-negative integers. The first five Hamming numbers are:

$$1 = 2^0 3^0 5^0 \quad 2 = 2^1 3^0 5^0 \quad 3 = 2^0 3^1 5^0 \quad 4 = 2^2 3^0 5^0 \quad 5 = 2^0 3^0 5^1$$

Quiz: Generate the list `ham` of all Hamming numbers in ascending order. Make use of the following observations:

1. The list starts with 1.
2. Every Hamming number $h > 1$ is of the form $a \cdot h'$ where $a \in \{2, 3, 5\}$ and h' is a Hamming number

⇒ the tail of `ham` is obtained by merging the following lists in increasing order

```
map (\x -> 2*x) ham   -- Hamming numbers multiple of 2
map (\x -> 3*x) ham   -- Hamming numbers multiple of 3
map (\x -> 5*x) ham   -- Hamming numbers multiple of 5
```

Another example: Hamming numbers

A Hamming number is of the form $2^i 3^j 5^k$ where i, j, k are non-negative integers. The first five Hamming numbers are:

$$1 = 2^0 3^0 5^0 \quad 2 = 2^1 3^0 5^0 \quad 3 = 2^0 3^1 5^0 \quad 4 = 2^2 3^0 5^0 \quad 5 = 2^0 3^0 5^1$$

Quiz: Generate the list `ham` of all Hamming numbers in ascending order. Make use of the following observations:

1. The list starts with 1.
2. Every Hamming number $h > 1$ is of the form $a \cdot h'$ where $a \in \{2, 3, 5\}$ and h' is a Hamming number

⇒ the tail of `ham` is obtained by merging the following lists in increasing order

```
map (\x -> 2*x) ham  -- Hamming numbers multiple of 2
map (\x -> 3*x) ham  -- Hamming numbers multiple of 3
map (\x -> 5*x) ham  -- Hamming numbers multiple of 5
```

⇒ Define an auxiliary function `merge xs ys` to merge two infinite lists of numbers which are in strict increasing order. The result should contain all numbers in strict increasing order.

If *op* is a binary operator and *v* some value, we can write

(*v* *op*) instead of `\x -> (`*v* *op* `x)`
(*op* *v*) instead of `\x -> (x` *op* *v*`)`

These abbreviations are called **sections**.

Example

```
> map (+3) [1,2,4] -- increment all list elements by 3
[4,5,7]
> filter (5<) [6,2,7,4,9] -- keep the numbers > 5
[6,7,9]
```

Hamming numbers (contd.)

```
merge::[Integer]->[Integer]->[Integer]
merge (x:xs) (y:ys)
  | (x<y)      = x:merge xs (y:ys)
  | (x==y)     = x:merge xs ys
  | otherwise = y:merge (x:xs) ys
ham::[Integer]
ham = 1:merge (merge (map (*2) ham)
                     (map (*3) ham))
              (map (*5) ham)
```

We can get the first *n* Hamming number with the predefined function `take`:

```
> take 20 ham -- get the first 20 Hamming numbers
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36]
```

Local definitions in Haskell

In Racket, we can work with blocks.
Haskell has no blocks but the following constructs:

```
let
    definition_1        -- can be function definitons, too
    ...
    definition_n
in  expr
```

or

```
expr  where
        definition_1
        ...
        definition_n
```

REMARK. All local definition should be indented with same non-empty amount.

If m, n are integers, then

- $[m..n]$ is the list of numbers from m to n
- $[m..]$ is the list of numbers starting from m, in increasing order
- Other list comprehensions, by example:

```
> [2*i | i<- [2..6]]
[4,6,8,10,12]
> [i | i<-[1..50],i `mod` 7==0]
[7,14,21,28,35,42,49]
> [(a,b,c) | a<-[1..10],b<-[1..10],c<-[1..10],a^2+b^2==c^2]
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
> lst = [(i,j) | i<-[1..],j<-[1..]]
> take 6 lst
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6)]
```

What is the *n*-th element of `lst`?

Quiz

Consider the following definitions:

```
sieve1,sieveAll::[Integer]->[Integer]
sieve1 (x:xs) = x:filter (\y->(mod y x) > 0) xs
sieveAll (x:xs)
  = x:sieveAll (filter (\y->(mod y x) > 0) xs)
```

- What does `sieve1 [n..]` compute for $n \in \mathbb{N}, n > 1$?

  Suggestion: check the results returned by `take 10 (sieve1 [n..])` for $n \in \{2, 3, 4\}$

- What does sieve1 [1..] compute?

  Does the computation terminate?

- What does `sieveAll [2..]` compute?

  Suggestion: check the result returned by `take 20 (sieveAll [2..])`

References

1. Simon Thompson. *Haskell: the craft of functional programming. Third Edition*. Pearson Education Limited. 2011.
2. Paul Hudak. *The Haskell School of Expression. Learning Functional Programming through Multimedia*. Cambridge University Press. 2007 (8th printing)