Programming Languages

P. Mironowicz

Politechnika Gdańska

25 października 2021

Programming paradigms

Programming paradigm is a way of looking at the execution process of a computer program *from the point of view of a programmer*.

Two basic programming paradigms: *imperative and declarative*.

Imperative and declarative programming

In **imperative** programming the the programmer looks at the program as a sequence of instructions executed sequentially and changing the state of the execution environment ($\approx$ as CPU does).

In **declarative** programming the programmer treats the code as a description of the conditions to be met by the solution of the chosen task ($\approx$ as most sensible non-IT professionals do, including humanists and mathematicians).

Functional programming

Functional programming is one of the paradigms of **declarative** programming.

Calculations are treated as a determination of the value of a mathematical function: one avoids the so-called **side effects** (skutków ubocznych) and **states of objects**.

One extensively use the so-called *lazy evaluation* (wartościowanie leniwe), that is, determining the value of function arguments only when needed.

Functional programming - side effects

The name can be a bit misleading ... A side effect is any effect of a function call that goes beyond returning a value.

Examples of side effects:

- object state change,
- change of global variable value,
- **input \output operations**.

Pure functions

They correspond to mathematical functions:

- Identical parameters $\Longrightarrow$ identical result.
- They do not cause side effects.

Basic advantages (we will come back to this):

- the function values can be determined in any order,
- easier analysis, avoiding errors.

Is the function `rand` pure?

Functional languages - examples

Examples of languages using this paradigm: Common Lisp, Wolfram Language, **Haskell**, R, XQuery \XSLT.

Python 3, C ++ 11 and C# 3.0 support elements of functional programming. In fact, most languages from the imperative paradigm *allow* for structures that are characteristic of the functional paradigm.

Functional programming - a bit of history

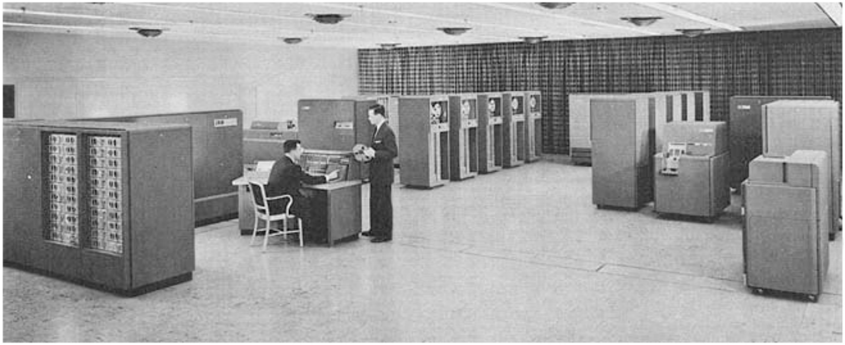
The beginnings. 1930s:

- lambda calculus (Alonzo Church)
- combinatorial calculus (Haskell Brooks Curry)

Mathematical concepts, not practical programming languages.

Functional programming - a bit of history

1958: LISP (John McCarthy), IBM 700/7000.



The second oldest high-level language (after Fortran from 1957). It has many dialects used even today (Common Lisp, Scheme, Clojure).

Functional programming - a bit of history

Haskell Brooks Curry (1900-1982) - logician, mathematician.

Curry's paradox: *If this sentence is true, then Santa Claus exists.*

Three languages were named in his honor: **Haskell**, Brook (GPGPU) and Curry ...

... and also a technique called currying.

1990: Haskell. Popular, purely functional language.

Formal system

The formal system consists of three elements:

1. set of symbols (= alphabet + rules of construction),
2. inference rules,
3. a set of axioms.

Simply anything where based on **assumptions** (3) we can draw **justified** (2) and **understandable** (1) conclusions.

Lambda calculus

Introduced by Alonzo Church in 1930. Used for research related to the fundamentals of mathematics, e.g. definability, computability and recursion.

It is equivalent to a single-tape Turing machine: the set of algorithms that can be written and performed in both calculation models is identical.

The λ account is more convenient, hence more popular.

Lambda calculus

Lambda expressions consist of:

- variables $x_1, x_2, \ldots$ (and some formulas built from them),
- symbols λ and dots or arrows,
- parentheses.

Using lambda expressions, one can write anonymous functions, i.e. functions without a name, e.g. $\lambda x.x^2$ (or, with an arrow, $\lambda x \rightarrow x^2$) defines the function that returns the square of its argument.

Lambda calculus

The set of all lambda expressions, denoted as Λ, is defined as follows:

- Variables belong to Λ.
- If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$ (so-called λ-abstraction).
- If $M, N \in \Lambda$, then $(MN) \in \Lambda$ (so-called λ-application).

The expression MN means passing N (treated as **argument**) to M (treated as a **function**).

Higher-order function = a function that accepts functions as arguments.

Lambda calculus

To simplify expressions:

- omit the outer brackets,
- shorten sequences, e.g. $\lambda x.\lambda y.M = \lambda xy.M$,
- use left-hand associative property (lewostronna łączność), i.e. $(MN)P = MNP$,
- extend the scope of the expression to the maximum right, i.e. $\lambda x.MN = \lambda x.(MN)$.

Currying and partial application

How to define a multi parameter function using a single parameter function?

Technique called **currying**: converting the calculation of the function of multiple arguments into a calculation sequence for single-parameter functions.

For instance, let:

$$F(x, y, z) = 3x + 2yz.$$

To calculate $F(1, 2, 3)$ we do the following:

$$\begin{aligned} F(1, y, z) &= 3 \cdot 1 + 2yz = 3 + 2yz =: G(y, z) \\ G(2, z) &= 3 + 2 \cdot 2z = 3 + 4z =: H(z) \\ H(3) &= 3 + 4 \cdot 3 = 15. \end{aligned}$$

$G(y, z)$ and $H(z)$ are the so-called **partial application**.

Currying and partial application

More generically:

Consider $F(x, y, z)$. Calling for each subsequent argument returns a function with one less argument (cf. $\mathrm{std::bind}$ in C++):

$$
\begin{aligned}
&F(x, y, z)(x_0) \Longrightarrow F_{x_0}(y, z)\\
&F_{x_0}(y, z)(y_0) \Longrightarrow F_{x_0 y_0}(z)\\
&F_{x_0 y_0}(z)(z_0) \Longrightarrow F_{x_0 y_0 z_0} = F(x_0, y_0, z_0).
\end{aligned}
$$

$F_{x_0}(y, z)$ and $F_{x_0 y_0}(z)$ are partial applications.

Currying and partial application

Recall: $\lambda xy.x + y = \lambda x.(\lambda y.x + y)$.

. Evaluation for $x = 3$ and $y = 4$ is the following:

$$\begin{aligned} &\lambda x.(\lambda y.x + y) \quad 3 \quad 4 \\ &\quad = (\lambda x.(\lambda y.x + y) \quad 3) \quad 4 \\ &\quad = (\lambda y.3 + y) \quad 4 \\ &\quad = 3 + 4 = 7 \end{aligned}$$

Expression $(\lambda y.3 + y)$ is a partial application.

Lambda calculus in C#

The lambda calculus in C# is implemented with the help of the right-hand associative operator =>.

The following examples of C# codes are based on examples from MSDN.

```
delegate int del(int i);
...
del myDelegate = x => x * x;
...
int j = myDelegate(5);
```

Lambda calculus in C#

If the compiler is unable to specify the data types, you can specify them explicitly:

```
(int x, string s) => s.Length > x
```

More extensive functions can be created using so-called lambda statements (*instrukcji lambda*):

```
delegate void TestDelegate(string s);
...
TestDelegate myDel = n => { string s = n + " " + "World"
    ; Console.WriteLine(s); };
myDel("Hello");
```

Lambda calculus in C#

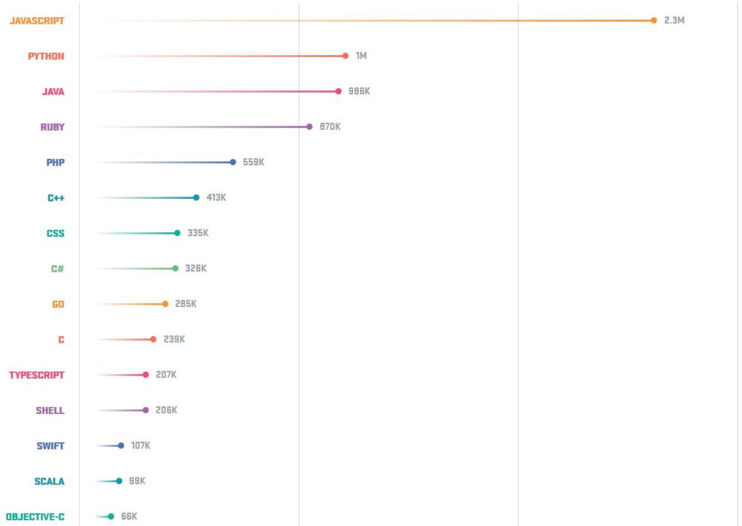
Lambda expressions in C# are especially useful for LINQ queries (Language INtegrated Query):

```
int[] scores = { 90, 71, 82, 93, 75, 82 };

// LINQ
int nHScore = scores.Where(n => n > 80).Count();

// nHScore is equal 4
```

Why functional programming?

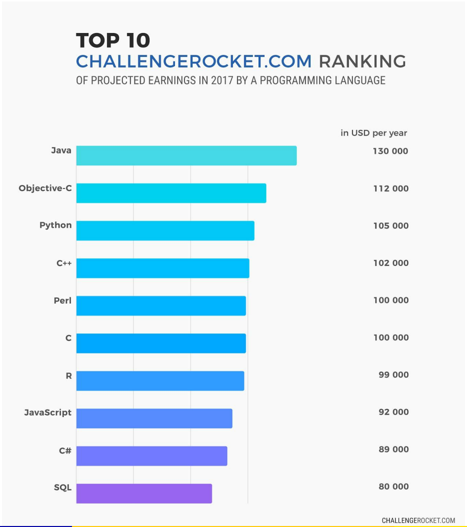
Why seemingly not?

Why the doubt? Number of projects at Github in 2017:



Wy certainly not?

What is most important...



But: a word from a **guru**

"A large fraction of the flaws in software development are due to programmers not fully **understanding all the possible states** their code may execute in. (...) **Programming in a functional style** makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible."

— John Carmack, *In-depth: Functional programming in C++*

One quote more

‘The problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.”

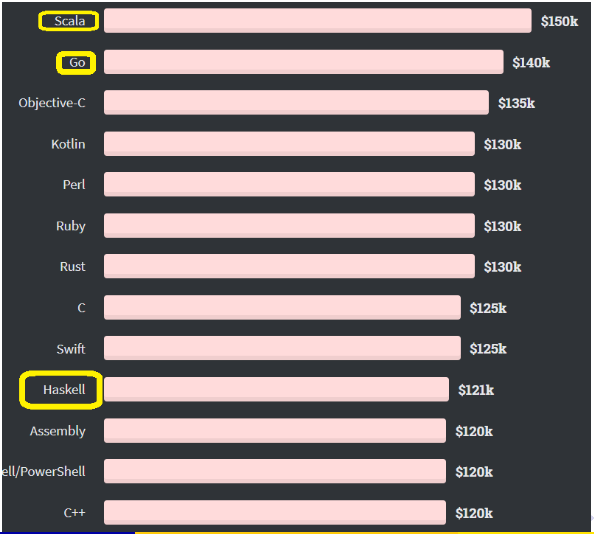
— Joe Armstrong, *Coders at Work: Reflections on the Craft of Programming*

Joe Armstrong is the creator of a functional language Erlang.

In OOP in order to use an object, you often need to import a huge hierarchy of other objects that you don’t need.

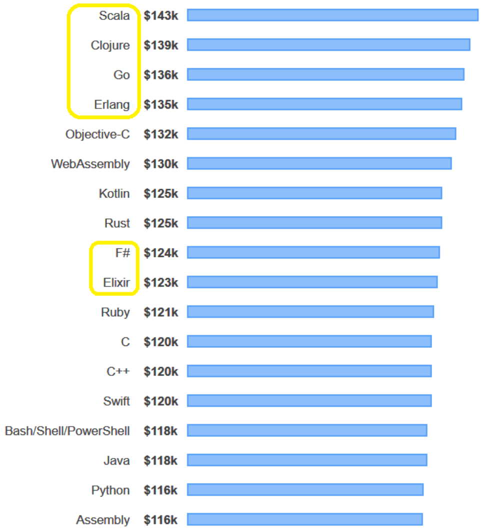
And!

Stack Overflow 2020:



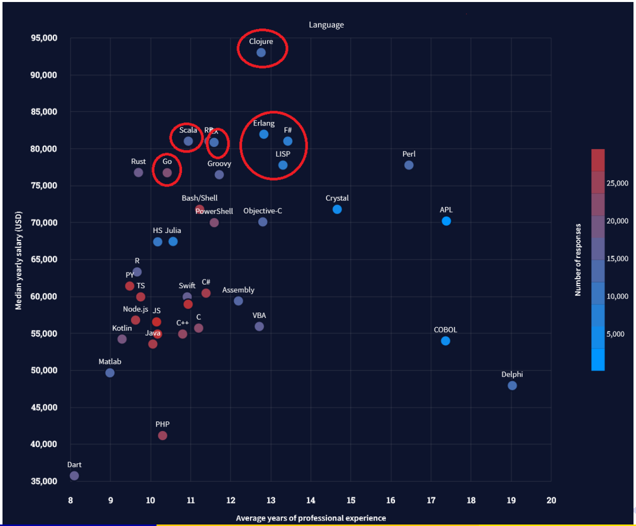
A year ago even more encouraging...

Stack Overflow 2019:



Anyway, it's getting better ......

Stack Overflow 2021 (this time world-wide):



Advantages of pure functions

Pure functions (in every language) have many advantages:

1. *Thread safety*,
2. easy code reuse,
3. easy to test,
4. easy to understand and document,
5. easy to analyze.

Why then? Because learning a functional programming **makes you a better programmer** regardless of the language you use.

What's next?

In the further part of the lecture we will discuss, among others:

- Haskell as an example of a purely functional language
- Programming in logic (Prolog)
- Functional programming in other languages (Python, C ++, C#, Java)
- How to define languages?

Haskell - a pure functional language

Haskell's features as a pure functional language:

- does not allow side effects of function calls,
- lazy evaluation (it even allows for infinitely long lists).

Strongly typed language. Types are automatically recognized, and usually there is no need to declare them.

No loops at all, recursion instead! **Variables do not change the value**!

GHC - Glasgow Haskell Compiler under BSD license. Prelude - basic library.

Defining functions

Simple functions:

```
Prelude> let pole r = pi * r^2
Prelude> pole 1
3.141592653589793
Prelude> pole 2
12.566370614359172
```

To avoid repetition we use a construction where, np. $a^2 + b^2$:

```
Prelude> let sos a b = sq(a) + sq(b) where sq x = x^2
Prelude> sos 3 4
25
```

Partial application in Haskell

An example of partial application. We calculate the expression $x + 2y + 3z$ and later assume $x = 0$:

```
Prelude> let sumaWaz3 x y z = x + 2 * y + 3 * z
Prelude> :t sumaWaz3
sumaWaz3 :: Num a => a -> a -> a -> a
...
Prelude> let sumaWaz3_x0 = sumaWaz3 0 -- part. appl.
    with x=0
Prelude> :t sumaWaz3_x0
sumaWaz3_x0 :: Integer -> Integer -> Integer
Prelude> sumaWaz3_x0 5 1 -- calculates 2 * 5 + 3 * 1
13
```

De facto all Haskell executions are performed using currying: `f x y` is always `(f x) y`.

Recursion

Factorial (pl. *silnia*):

```
silnia :: Integer -> Integer -- explicit type
    specification
silnia 0 = 1
silnia n = n * silnia(n - 1)
```

Note: the comment follows after -- (double dash).

Fibonacci sequence:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib(n - 1) + fib(n - 2)
```

Function composition

Function composition in maths ($\circ$): $(f \circ g)(x) \equiv f(g(x))$.

Function composition with operator . (dot). Note: In Polish *razy* means “times”.

```
Prelude> let plus1 x = x + 1
Prelude> let razy2 x = x * 2
Prelude> let razy2plus1 = razy2 . plus1
Prelude> let plus1razy2 = plus1 . razy2
Prelude> razy2plus1 3 -- (3 + 1) * 2
8
Prelude> plus1razy2 3 -- (3 * 2) + 1
7
```

η–conversion

η–conversion expresses some kind of tautology...

Note that the lambda expression $\lambda x.f(x)$ which is an anonymous function assigns to x the value of $f(x)$, where $f(\cdots)$ is no **not** an anonymous function.

Attention! We assume that x is **not** a free variable in $f(\cdot)$!

For instance `\x -> abs x` is equivalent to `abs` , and `(\x -> abs x) (-3)` works the same as `abs (-3)` .

η–conversion

η–conversion means switching between one form of writing and another, in other words it is **adding or rejecting the level of abstraction**.

For example we can write in Haskell: `\x -> sin x` . This is an η–conversion from the function `sin` to a lambda expression.

```
Prelude> map (\x -> sin x) [0, pi, pi/2, pi/4, pi/5] --
    so-called eta abstraction
[0.0,1.2246467991473532e
    -16,1.0,0.7071067811865475,0.5877852522924731]
```

η–conversion

η–conversion to the other direction is the η-reduction:

```
Prelude> let x2 = (\x -> x^2) -- eta reduction
Prelude> x2 4
16
Prelude> let razy2 = (2 *) -- eta reduction
Prelude> razy2 2
4
```

The coding style in which one frequently use the η–reduction is called *tacit programming* (see below).

Tacit programming

Tacit programming (in Polish “milczące programowanie”), or *pointfree programming*, is a coding style that avoids explicit specification of arguments of functions.

```
Prelude> let razy2 x = (2 * x)
Prelude> razy2 2
4
Prelude> let razy2' = (2 *) -- pointfree
Prelude> razy2' 2
4
Prelude> let sumuj = foldr (+) 0
Prelude> sumuj [1,2,3]
6
```

Tacit programming

```
Prelude> let sq = (^2)
Prelude> sq 5
25
Prelude> let pw = (2^)
Prelude> pw 5
32
```

A standard construction: if ... then ... else ... . In Haskell the else part **must** be defined.

```haskell
znak x =
    if x > 0
        then 1
        else if x == 0
            then 0
            else -1
```

Indentation (Wcięcia)

Indentations in Haskell matter for the parser. The use of indentation as a syntax element is sometimes called *offside rule* ("spalony").

When writing a multi-line definition, subsequent lines must have at least the same indentation as the first line.

Remark: For compatibility tabs should be avoided (in Windows, tabs have 4 spaces, in Linux 8).

Guards

Let's look at a typical mathematical notation:

$$|x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise.} \end{cases}$$

Construction in Haskell with the use of the so-called *guards*:

```
bezwzgledna x
      | x < 0 = -x
      | otherwise = x
```

Note: In Polish *bezwzględny* means "absolute".

Case

The expression *case* is similar to *guards*. Example: Replacing a number with the name of the month (here in Polish):

```
miesiac n =
    case n of
        1 ->  "Styczen"
        2 ->  "Luty"
        3 ->  "Marzec"
        4 ->  "Kwiecien"
        5 ->  "Maj"
...
        9 ->  "Wrzesien"
        10 ->  "Pazdziernik"
        11 ->  "Listopad"
        12 ->  "Grudzien"
        _ ->  "ERROR!"
```

Lists

Lists are used to store multiple items of the same type in a fixed order:

```
Prelude> let l1 = [2..9]
Prelude> l1
[2,3,4,5,6,7,8,9]
```

Concatenation of lists:

```
Prelude> concat [[1,2,3],[4,5],[6,7,8]]
[1,2,3,4,5,6,7,8]
```

One can also use the concatenation operator++.

Head and tail

We can divide the list into head and tail using... head and tail .

```
Prelude> let l1 = [1..7]
Prelude> head l1 -- head of the list
1
Prelude> tail l1 -- the rest of the list
[2,3,4,5,6,7]
```

Consing

Adding an item to the top of the list is called **consing** (from *constructor*):

```
Prelude> let liczby = [1..5]
Prelude> liczby
[1,2,3,4,5]
Prelude> 0:liczby -- this is consing
[0,1,2,3,4,5]
Prelude> 0:liczby ++ 6:7:[] -- consing and concatenation
[0,1,2,3,4,5,6,7]
```

All items on a list must be of the same type.

Consing

The consing operator can be used to divide the list into head and tail. Note: In Polish *długość* means “length”. Recursive definition of the list length:

```
dlugosc :: [a] -> Integer
dlugosc [] = 0
dlugosc (x:xs) = (dlugosc xs) + 1
```

All list items must be of the same type.

Creating and indexing lists

You can create lists in various ways. Strings are a sort of lists.

```
Prelude> [0,5..40] -- every 5th element
[0,5,10,15,20,25,30,35,40]
Prelude> concat ["ABC","DEF","GH"] - string
    concatenation
"ABCDEFGH"
Prelude> ['A','B','C'] == "ABC" -- LHS and RHS
    expressions are equivalent
True
```

0 based indexing using the operator `!!` :

```
Prelude> [0,5..40]!!3 -- 4th (0 based indexing) element
15
```

List truncation

List truncation:

```
Prelude> take 3 [1..10] -- take first 3 elements
[1,2,3]
Prelude> drop 3 [1..10] -- drop first 3 elements
[4,5,6,7,8,9,10]
```

Subsequence of list elements

Definition of selecting items from the range:

```
wez odtad dotad lista = take (dotad - odtad + 1) (drop
    odtad lista)
...
*Main> wez 3 6 [1..10]
[4,5,6,7]
```

Note: In Polish *wez* means ≈“take those”; *odtąd* means “from here”; *dotąd* means “til here”.

Mapping

Apply the given function to all elements of the list.
Execute in GHCi:

```
Prelude> let l1 = [2..9]
Prelude> let ff x = x^2
Prelude> map ff l1
[4,9,16,25,36,49,64,81]
Prelude> map (*2) l1
[4,6,8,10,12,14,16,18]
```

Using lambda expressions:

```
Prelude> mapuj (\ x -> x^2) [2..9] -- with lambda
    expression
[4,9,16,25,36,49,64,81]
```

Mapping and *pattern matching*

How to do it? Let us define:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = [] -- so-called pattern matching
map fun (x:xs) = (fun x) : (map fun xs)
```

Matching always happens with the first matching pattern (from up to down):

- `_` - matches everything but does not bind,
- `[]` - matches the empty list, does not bind,
- `fun` - matches everything, binds the match with the variable `fun`,
- `(x:xs)` - matches *non-empty* list formed by *cons*ing element (bound to `x`) with list (bound to `xs`).

Mapping and filtration

The mapping function may return values of a different type:

```
Prelude> let l1 = [2..9]
Prelude> let is57 x = (x == 5 || x == 7)
Prelude> map is57 l1
[False,False,False,True,False,True,False,False]
```

Filtration - selection of those elements for which a given predicate is true. Command filter .

```
Prelude> filter is57 l1
[5,7]
```

Mapping and filtration

Further examples:

```
*Main> map (*10) [1,2,3,4,5]
[10,20,30,40,50]
*Main> map (\x -> x^2 + 1) [1,2,3,4,5] -- with lambda
    expression
[2,5,10,17,26]
...
*Main> let is57 x = (x == 5 || x == 7)
*Main> filter is57 [1,5,2,5,3,5] -- filtration
[5,5,5]
```

Folding

Folding (or *reduction*, or *aggregation*) is a sort of „joining together" (or „summing up") all elements into one. One can fold from the left side (*foldl*) or from the right side (*foldr*).

Example in Haskell:

```
Prelude> foldr (+) 0 [3,4,5,1]
13
Prelude> foldl (*) 1 [1,2,3,4,5]
120
```

Indeed:
$suma([3,4,5,1]) = foldl(+,0,[3,4,5,1]) = \{[(0+3)+4]+5\}+1 = 13.$
Note: In Polish *suma* means "sum".

Folding

We must define: a function with two arguments or the relevant type, the initial value and the list to be reduce.
The direction of reduction is important when the function is not commutative.
For the initial value 0 and a list $[3, 4, 5, 1]$ and operator $-$ (minus), folding from the left side, we have:

$$\{[(0-3)-4]-5\}-1=-13,$$

from the right

$$3-\{4-[5-(1-0)]\}=3.$$

```
Prelude> foldl (-) 0 [3,4,5,1]
-13
Prelude> foldr (-) 0 [3,4,5,1]
3
```

Zip and unzip

Zip - combining two lists into a list of couples. Unzip - the inverse operation.

```
Prelude> :t zip  -- zapytanie o typ wyrazenia
zip :: [a] -> [b] -> [(a, b)]
Prelude> :t unzip
unzip :: [(a, b)] -> ([a], [b])
...
Prelude> zip [1,2,3] ['A','B','C']
[(1,'A'),(2,'B'),(3,'C')]
Prelude> unzip [(1,'A'),(2,'B'),(3,'C')]
([1,2,3],"ABC")
```

zipWith

zipWith performs a zip and an additional operation on each pair.

```
Prelude> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
Prelude> zipWith (*) [1,3,5] [2,2,1]
[2,6,5]
```

For two vectors $\vec{v} = (v_1, \ldots, v_N)$, $\vec{w} = (w_1, \ldots, w_N)$ scalar product is the sum of products of elements: $\sum_{i=1}^{N} v_i \cdot w_i$.

```
Prelude> foldl (+) 0 (zipWith (*) [1,3,5] [2,2,1]) --
    scalar product
13
```

Example: the sum of 3 vectors

One can zip on 3 lists.

```
Prelude> zipWith3 (\ x y z -> x + y + z) [1,2,3] [4,5,6]
      [7,8,9]
[12,15,18]
```

Tacit programming

A more drastic example of *tacit programming*:

```
Prelude> let mf warunek funkcja lista = filter warunek (map funkcja lista)
Prelude> let mf' = (. map) . (.) . filter -- tacit version
Prelude> mf (>10) (^2) [1,2,3,4,5]
[16,25]
Prelude> mf' (>10) (^2) [1,2,3,4,5]
```

The example above is by Udo Stenzel.
Note: In Polish *warunek* means “condition”; *funkcja* means “function”; *lista* means “list”.

Tacit programming

For those who are curious... derivation, by https://mail.haskell.org/pipermail/haskell-cafe/2006-September/017758.html:

```
func2 f g l = filter f (map g l)
func2 f g = (filter f) . (map g)  -- definition of (.)
func2 f g = ((.) (filter f)) (map g)  -- desugaring
func2 f = ((.) (filter f)) . map   -- definition of (.)
func2 f = flip (.) map ((.) (filter f)) -- desugaring,
    def. of flip
func2 = flip (.) map . (.) . filter   -- def. of (.),
    twice
func2 = (. map) . (.) . filter     -- add back some sugar
```

Infinite lists and *lazy evaluation*

In Haskell, the values of individual elements of tables are only calculated when they are needed (*lazy evaluation*). Thus, one can operate e.g. on infinite lists:

```
Prelude> let fib = 1 : 1 : zipWith (\x y -> x+y) fib (
    tail fib)
Prelude> fib!!0
1
Prelude> fib!!5
8
Prelude> fib!!7
21
Prelude> fib!!8
34
Prelude> fib!!9
55
```

Module Data.List

Additional operations on the lists are available after the module Data.List has been loaded:

```
Prelude> import Data.List
Prelude Data.List> intersperse ',' "12345" -- przeplatanie
"1,2,3,4,5"
Prelude Data.List> reverse [1,2,3,4,5] -- odwr. kolejnosci
[5,4,3,2,1]
Prelude Data.List> transpose [[11,12,13],[21,22,23],[31,32,33]] -- transpozycja
[[11,21,31],[12,22,32],[13,23,33]]
```

Module Data.List

Further operations from the module Data.List :

```
Prelude Data.List> subsequences "XYZ" -- wszystkie
    podciagi
["","X","Y","XY","Z","XZ","YZ","XYZ"]
Prelude Data.List> permutations [1,2,3] -- wszystkie
    permutacje
[[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
Prelude Data.List> replicate 5 "X" -- powtorzenie
    elementu
["X","X","X","X","X"]
Prelude Data.List> sort [1,5,3,2,7,8,6,4,7,3] --
    sortowanie
[1,2,3,3,4,5,6,7,7,8]
```

Module Data.List and infinite lists

```
Prelude Data.List> let potegi2 = iterate (\x -> 2*x) 1
    -- infinite composition of a given operation
Prelude Data.List> potegi2!!10
1024
Prelude Data.List> potegi2!!32
4294967296
Prelude Data.List> let czas = cycle "tik tak " --
    infinite cycle
...
*Main Data.List> wez 1000 1100 czas -- elements from a
    list within given scope
"tik tak tik tak tik tak tik tak tik tak tik tak tik tak
    tik tak tik tak tik tak tik tak tik tak tik t"
```

Note: In Polish *potęga* means “power”; *czas* means “time”; *tik tak* means “tic toc”.

Memoization

Two ways to define the Fibonacci sequence:

```
slow_fib :: Int -> Integer
slow_fib 0 = 0
slow_fib 1 = 1
slow_fib n = slow_fib (n-2) + slow_fib (n-1)

memoized_fib :: Int -> Integer
memoized_fib = (map fib [0 ..] !!) -- infinite list
    where fib 0 = 0
          fib 1 = 1
          fib n = memoized_fib (n-2) + memoized_fib (n-1)
```

Timings

The above example comes from `https://wiki.haskell.org/Memoization`.

Recall that one of the properties of Haskell is the *lazy evaluation*, i.e. the values of the expressions are determined only when they are needed.

The `System.CPUTime` module contains the `getCPUTime` function that returns the CPU time used by the current program, which can be displayed in a readable way using the `System.TimeIt` module.

The example call `slow_fib 30` takes 1.33s, while `memoized_fib 30,000` takes only 0.15s!

Z poziomu GHCi wyświetlanie czasu wykonania i użycia pamięci można włączyć komendą `:set +s`.

Fold... again

Let's define a function:

```haskell
f' :: (Eq a, Num a) => a -> Bool -> Bool
f' = (||).(==0)
```

or, equivalently:

```haskell
f :: (Eq a, Num a) => a -> Bool -> Bool
f x y = (x==0) || y
```

Let's look at the definition of the || operator:

```haskell
True  || _ = True
False || x = x
```

foldr

What is `foldr f False [3,0,1,5]` ? (f x y = (x == 0) || y)

```
f 3 (foldr f False [0,1,5])
(3==0) || (foldr f False [0,1,5])
False || (foldr f False [0,1,5])

foldr f False [0,1,5]
f 0 (foldr f False [1,5])
(0==0) || (foldr f False [1,5])
True || (foldr f False [1,5])

True
```

We didn't need to do anything with the list elements after a certain place.
It even works with infinite lists.

What about foldl ?

Suppose we would like to do something similar to foldl .

For some g the command foldl g False [3,0,1,5] expands to:

```
g (g (g (g False 3) 0) 1) 5
```

From here you can see that g should be of the form:

```
g :: (Eq a, Num a) => Bool -> a -> Bool
g x y = x || (y==0)
```

foldl must come to an end as it starts computing from the deepest nest.

foldr VS foldl

```
Prelude> lista = 3:0:[1..10000000]
(0.00 secs, 0 bytes)
Prelude> f x y = (x==0) || y
(0.01 secs, 0 bytes)
Prelude> g x y = x || (y==0)
(0.00 secs, 0 bytes)
Prelude> foldr f False lista
True
(0.01 secs, 83,800 bytes)
Prelude> foldl g False lista
True
(9.42 secs, 2,016,480,176 bytes)
```

Basic data types

Some basic data types:

- Integers: Integer - any size, Int - 32 or 64 bits.
- Real numbers: Double and Float .
- Logical values: Bool .

Rational numbers

Module Data.Ratio (we will tell more about modules later) has a type that stores rational numbers. They can be created, e.g. using the operator % :

```
Prelude> import Data.Ratio
Prelude Data.Ratio> let r = 3 % 4
Prelude Data.Ratio> :t r
r :: Integral a => Ratio a
Prelude Data.Ratio> let s = 4 % 9
Prelude Data.Ratio> let t = r * s
Prelude Data.Ratio> t
1 % 3
Prelude Data.Ratio> toRational(1.25) -- conversion from
    double to rational
5 % 4
```

Tuples (*krotki*)

Tuple does not have a consing operation, but can store values of different types.

```
Prelude> let tu1 = (1,'g',True,"gfgf")
Prelude> :t tu1
tu1 :: (Integer, Char, Bool, [Char])
...
Prelude> let tu2 = (1.1, tu1)
Prelude> tu2
(1.1,(1,'g',True,"gfgf"))
Prelude> :t tu2
tu2 :: (Double, (Integer, Char, Bool, [Char]))
```

Pairs

When a tuple is a pair of two elements, we can retrieve them using fst and snd .

```
Prelude> let pa1 = (2,3) -- pair
Prelude> fst pa1 -- 1st element
2
Prelude> snd pa1 -- 2nd element
3
```

where

As we mentioned earlier, the where construction avoids repetition and improves the readability of complex expressions:

```
liczbaPierwiastkow a b c -- wielomian a x^2 + b x +c
  | delta < 0 = 0
  | delta == 0 = 1
  | otherwise = 2
where delta = b^2 - 4 * a * c
...
*Main> liczbaPierwiastkow 1 2 3
0
*Main> liczbaPierwiastkow 1 20 3
2
*Main> liczbaPierwiastkow 1 4 4
1
```

Note: In Polish *liczba* means "number"; *pierwiastek* means "square

let … in …

The construction `let wyrazenie1 in wyrazenie2` allows you to define what the expression `wyrazenie1` means within the expression `wyrazenie2`.

Example: the Brahmagupta formula for the S area of a quadrilateral (czworokąt) with sides a, b, c, d inscribed (wpisany) in a circle:

$$S = \sqrt{(p-a)\cdot(p-b)\cdot(p-c)\cdot(p-d)}, \tag{1}$$

where $p = \frac{1}{2}(a+b+c+d)$.

let ... in ...

```
-- Brahmagupta formula
-- quadrilateral side lengths as parameters
poleCzworokataWpisanegoWOkrag a b c d =
let p = (a + b + c + d) / 2
in sqrt((p-a) * (p-b) * (p-c) * (p-d))
...
*Main> poleCzworokataWpisanegoWOkrag 2 2 2 2
4.0
*Main> poleCzworokataWpisanegoWOkrag 2 6 9 7
30.0
```

The content of where is visible for the entire definition, the content of let only inside the expression after in .
Note: In Polish *pole* means area; *czworokąt* means “quadrilateral”; *wpisany* means “inscribed in”; *okrąd* means “a cricle”.

Curly bracket

Instead using indentation one can disambiguate the syntax with curly brackets:

```
bezklamer = let a = 1
b = 2
c = 3
in a + b + c

zklamrami = let { a = 1;  b = 2;
c = 3 }
in a + b + c
```

Note: In Polish *bez klamer* means “without curly brackets”; *z klamrami* means “with curly brackets”.

Creating modules

Creation of a module `NazwaModulu` requires placing the definitions of expressions that we want to include in a file *NazwaModulu.hs*.
Note: In Polish *nazwa* means “name”.

To provide a name to a module, start the source file with an expression: `module NazwaModulu where` . The name should start with a capital letter.

One file may contain at most one module.

The module (as we have seen in the examples above) is imported using the command `import NazwaModulu` .

Creating modules

Let us create a file named *NazwaModulu.hs* with the following content:

```haskell
module NazwaModulu where

prostaInterakcja = do
    putStrLn "Enter one character:"
    c <- getChar
    putStrLn ("\nYou provided the character: " ++ [c])
```

The module can be compiled with a GHC command *ghc –make NazwaModulu.hs*.
Note: In Polish *prosta interakcja* means “simple interaction”.

Side effects... - again

As we mentioned, Haskell is a purely functional language, i.e. it doesn't allow side effects of a function. One of the side effects are I/O...

... I/O operations are allowed in Haskell, but are clearly separated. This is related to the concept of *monads*...

... which we won't discuss *yet* but...

Standalone programs in Haskell

... every standalone Haskell program is a function that returns a value of type IO .

The file with a stand-alone program is *de facto* a module called Main .

Creating a program

Let us create a file *prostyProgram.hs* with the content:

```
-- prostyProgram.hs
module Main where

import NazwaModulu

policzSinus = do
    putStrLn "Enter a number:"
    x <- readLn :: IO Double
    putStrLn ("\nsin(" ++ show x ++ ") = " ++ show (sin(
   x)))
...
```

Creating a program

```
...
main = do
    prostaInterakcja -- action from the imported module
    policzSinus
```

The program can be compiled with a GHC command *ghc –make -o prostyProgram prostyProgram.hs*.

Program execution

Sample execution from the GHC interpreter:

```
Prelude> :l prostyProgram.hs
Ok, modules loaded: Main, NazwaModulu.
Prelude Main> main
Enter one character:
q
You provided the character: q
Enter a number:
3.14

sin(3.14) = 1.5926529164868282e-3
```

Program execution

Sample execution from the OS shell:

```
Enter one character:
q
You provided the character: q
Enter a number:
3.14

sin(3.14) = 1.5926529164868282e-3
```

To confirm the character (in this example the letters **q**) we use CTRL + D.

Export of a module

Let's create a Haskell file with a definition of a function that we want to call from a C++ program:

```
module FunDlaCpp where

foreign export ccall fun :: Double -> Double

fun :: Double -> Double
fun = (\x -> x^2)
```

Let us save it under the name DoCpp.hs.

Compilation, e.g. with the command *ghc -c -XForeignFunctionInterface DoCpp.hs*, will create the object file *DoCpp.o* and the related header file *DoCpp_stub.h*.

Stub of a Haskell module in C++

The header file *DoCpp_stub.h* has the following content:

```
#include "HsFFI.h"
#ifdef __cplusplus
 extern "C" {
#endif
 extern HsDouble fun(HsDouble a1);
#ifdef __cplusplus
}
#endif
```

The keyword of the C language extern next to the declaration means that the compiler (not the linker) has no access to the definition of the expression.

Stub of a Haskell module in C++

Note that the stub file includes the header file `HsFFI.h` (FFI stands for *foreign function interface*).

This file specifies how to map variable types from Haskell to C or C++.

C++ code calling a Haskell module

An example file using the Haskell module may have the following form:

```
#include <iostream>
#include "DoCpp_stub.h"

int main(int argc, char *argv[])
{
hs_init(&argc, &argv);
std::cout << fun(3) << std::endl;
hs_exit();
}
```

C++ code calling a Haskell module

Let's save the above file under the name *uruchomHs.cpp* and compile using the command *g++ -c uruchomHs.cpp -I/usr/lib/ghc/include*.

We obtain an object file *uruchomHs.o*.

In the example call, we indicate to the compiler that additional header files may be searched in the directory */usr/lib/ghc/include*. It contains, among others, the mentioned *HsFFI.h* file. The path to this directory varies depending on the system and its configuration.

We link object files using the command *ghc -no-hs-main DoCpp.o uruchomHs.o -lstdc++ -o uruchomHs*. The *-no-hs-main* option will avoid double definition of main .

Synonyms

Using synonyms, we can give alternative names to built-in types (similar to typedef in C). Type names begin with a capital letter.

```
type Imie = String
type Nazwisko = String
type Telefon = Int
```

Note: In Polish *imię* means “name”; *nazwisko* means “surname”; *telefon* means “phone (number)”.

Defining data types

You declare your own data type using the keyword `data` :

```
data Data = Data Int Int Int -- dzien, miesiac, rok
```

There may be several different definitions under the same name (similar to `enum` in C):

```
data Znajomy =
 Bliski Imie Telefon Urodziny
 | Daleki Imie Nazwisko Telefon
```

`Znajomy` is a *type name*, `Bliski` and `Daleki` are *constructors*.
Note: In Polish *znajomy* means “friend”; *bliski znajomy* means “close friend”; *daleki znajomy* means “colleague”.

Defining data types

Functions for defined data types:

```
wyswietlDate :: Data -> String
wyswietlDate (Data d m r) =
show d ++ " " ++ miesiac m ++ " " ++ show r
...
*Main> let nowyRok = Data 1 1 2022
*Main> wyswietlDate nowyRok
"1 Styczen 2022"
```

(The expression `miesiac` was defined in the previous lecture.)

Defining data types

```haskell
wyswietlZnajomego :: Znajomy -> String
wyswietlZnajomego (Bliski imie telefon dataUrodzin) =
imie ++ " ma urodziny " ++ wyswietlDate dataUrodzin ++ ", jego telefon to " ++ show telefon ++ "."
wyswietlZnajomego (Daleki imie nazwisko telefon) =
"Pan(i) " ++ imie ++ " " ++ nazwisko ++ " ma numer telefonu " ++ show telefon ++ "."
```

Defining data types

```
paulina::Znajomy
paulina = Bliski "Paulina" 581000000 (Data 10 9 1992)
rektor::Znajomy
rektor = Daleki "Krzysztof" "Wilde" 583472747
...
*Main> wyswietlZnajomego paulina
"Paulina ma urodziny 10 Wrzesien 1992, i numer telefonu
    581000000."
*Main> wyswietlZnajomego rektor
"Pan Krzysztof Wilde ma numer telefonu 583472747."
```

Integers

Let's take a look at the following integer definitions:

```
Prelude> let a = 1
Prelude> let b = 1::Int
Prelude> let c = 1::Integer
Prelude> :t a
a :: Num p => p
Prelude> :t b
b :: Int
Prelude> :t c
c :: Integer
```

Operations `+`, `-` etc. are allowed between `a` and `b` (they return the type `Int`) and `a` and `c` (return type `Integer`), but **not** between `b` and `c`.

What is `Num` ? And what does `=>` mean?

Real numbers

Let us try now with real numbers:

```
Prelude> let x = 1.0
Prelude> let y = 1::Float
Prelude> let z = 1::Double
Prelude> :t x
x :: Fractional p => p
Prelude> :t y
y :: Float
Prelude> :t z
z :: Double
```

Arithmetic operations are similar to integer types: Fractional "matches" both Float and Double , but the latter two do not "match" each other.

What is this Fractional ?

Numeric data types

Let's try this:

```
Prelude> :t (a+x)
(a+x) :: Fractional a => a
Prelude> :t (a+y)
(a+y) :: Float
Prelude> :t (a+z)
(a+z) :: Double
```

We get the same types when we change the order. But the other combinations (eg Hinline Int plus Hinline Float) are not allowed.

Standard classes hierarchy

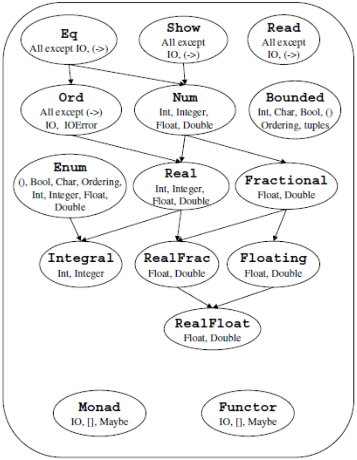


Figure 6.1: Standard Haskell Classes

Haskell 2010 Language Report, ed. Simon Marlow, strona 76.

Class Eq

The class (actually the so-called **type class**) is declared with the keyword class .

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

    -- Minimal complete definition:
    --       (==) or (/=)
  x /= y      = not (x == y)
  x == y      = not (x /= y)
```

Standard numerical type class Num

An example of Num type class declaration from Prelude module:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)       :: a -> a -> a
  negate              :: a -> a
  abs, signum         :: a -> a
  fromInteger         :: Integer -> a
    -- Minimal complete definition:
    --       All, except negate or (-)
  x - y              = x + negate y
  negate x           = 0 - x
```

In the expression (Eq a, Show a) => Num a , the operator => declares that the Num type inherits from Eq and Show . This is called **type constraint**.

Type class Fractional

```
class  (Num a) => Fractional a  where
  (/)                  :: a -> a -> a
  recip                :: a -> a
  fromRational         :: Rational -> a

    -- Minimal complete definition:
    --       fromRational and (recip or (/))
  recip x              = 1 / x
  x / y                = x * recip y
```

Inheritance

A simple example of the data structure inheriting from the class Show :

```
data Trojkat = Trojkat
     {a::Double ,
     b::Double ,
     c::Double
     } deriving (Show) -- inferitance from the class Show
```

The declaration that the type inherits from the class Show , **automatically** allows to print the type.

Haskell can **automatically** inherit from **some** classes, e.g. Eq , Ord , Enum , Show , Read . However, the "automatic" definition does not always meet the needs.

Inheritance

```
t1 = Trojkat{a = 3, c = 5, b = 4} -- one way to define
t2 = Trojkat 7 8 9 -- second way to define
t3 = Trojkat 4 4 4
...
*Main> show t1 -- automatically defined show
"Trojkat {a = 3.0, b = 4.0, c = 5.0}"
*Main> show t2
"Trojkat {a = 7.0, b = 8.0, c = 9.0}"
```

Inheritance from Eq

One of the basic classes in Haskell is the class Eq . It contains only the operator == and Hinline/= (one operator is a negation of the other, hence it is enough to define only one of them).
Declaration that a given type inherits from some class is made using the keyword instance .

```
obwod (Trojkat a b c) = a + b + c

instance Eq Trojkat where
    t1 == t2 = (obwod t1) == (obwod t2)
```

The mentioned keyword deriving would be automatically defined by the operator == , but in some *other* (default) way.
Note: In Polish *obwód* means "circumference"; *trójkąt* means "triangle".

Inheritance from Ord

Another of the basic classes in Haskell is the class Ord . It contains the operator <= , and inherits from the Eq class.

```
instance Ord Trojkat where
    (<=) t1 t2 = (obwod t1) <= (obwod t2)
...
*Main> t1 == t3
True
*Main> t1 == t2
False
*Main> t1 < t2
True
*Main> t1 < t3
False
```

Class declaration

Let's declare the class Pomnazalne (in Polish means “multiplicable”) containing the method pomnoz (in Polish means “multiply”). Let's declare that

```
class Pomnazalne a where
    pomnoz :: a -> Double -> a

instance Pomnazalne Trojkat where
    pomnoz (Trojkat a b c) x = Trojkat (a*x) (b*x) (c*x)
...
*Main> show t1
"Trojkat {a = 3.0, b = 4.0, c = 5.0}"
*Main> pomnoz t1 10
Trojkat {a = 30.0, b = 40.0, c = 50.0}
```

Note: In Polish *pomnażalne* means “multiplicable”; *pomnóż* means “multiply”

Lambda expressions

Simple lambda expression:

```
fun = lambda x, y : 2 * x + y
```

Lambda expression with parameters (in fact a partial application, often called in Python a closure):

```
def kwadratowa(a,b,c):
    return lambda x : a * (x ** 2) + b * x + c

fun2 = kwadratowa(2, 4, -6) # 2*(x-1)*(x+3) = 2*x*x + 4*
    x - 6

print((fun2(0), fun2(1), fun2(-3), fun2(2)))
```

The above will write a tuple $(-6,\ 0,\ 0,\ 10)$.

Partial application and nested functions

```
def potegaN(n):
  def potegaX(x): # funkcja zagniezdzona
    return x ** n
  return potegaX # zwraca closure

potega2 = potegaN(2)
potega3 = potegaN(3)

print(potega2(5))
print(potega3(5))
print(potega2.__closure__[0].cell_contents)
print(potega3.__closure__[0].cell_contents)
```

This will print: 25, 125, 2 i 3.

Partial application in functools

```python
from functools import partial

def potegaXN(x, n):
  return x ** n

potegaX2 = partial(potegaXN, n = 2)
print(potegaX2(7))
print(potegaX2.func)
print(potegaX2.keywords)
```

This prints:

```
49
<function potegaXN at 0x000000190B6858C8>
{'n': 2}
```

Iterable and iterator

Iterable in Python is an obiect containing a method `__iter__()` that returns an iterator.

Iterator is an object with a method `__next__()` .

```
prosta_krotka = ("raz", "dwa", "trzy")

it = iter(prosta_krotka) # pobierz iterator

# iteruj po elementach
print(next(it))
print(next(it))
print(next(it))
# print(next(it)) # spowoduje wyjatek StopIteration
```

Iterable and iterator

```python
for x in prosta_krotka:
    print(x)
```

Both will print:

```
raz
dwa
trzy
```

Note: In Polish *raz, dwa, trzy* means “one, two, three”.

map

Python has operations such as map , filter and reduce .

```
print(list(map((lambda x : x**2), range(5))))
```

This prints [0, 1, 4, 9, 16] . (Recall: range(5) gives [0, 1, 2, 3, 4].) Pay attention to the function list , which ensures that the contents of the list will actually be counted, because the function map only returns as the result a definition with a lazy evaluation, which is something like:

```
<map object at 0x2ad3a54e7748>
```

More *pythonic* way using *list comprehension*:

```
print([e**2 for e in range(5)])
```

filter

```
lista1 = [1, −7, 5, 3, −2]
print(list(filter((lambda x : x > 0), lista1)))
```

Prints [1, 5, 3] .

More *pythonic*:

```
print([e for e in lista1 if (e > 0)]) # [x for x in S if
    P(x)]
```

reduce

Reduction operation from functools (since 3.x, previously in standard library):

```
print(reduce((lambda x, y : x + y), [1, 2, 3, 4]))
print(reduce((lambda x, y : x * y), [1, 2, 3, 4]))
```

Prints 10 and 24 . Alternatively (and more efficiently) one can use a function sum and numpy.prod or math.prod (since 3.8).

What value do we start the summing, what the product? From the result of acting on the first two arguments.

You can also do in a more general way. Let's look at the header:

```
def reduce(function, sequence, initial=None)
```

accumulate

Function accumulate returns partial results:

```
from itertools import accumulate
print(list(accumulate([1, 2, 3, 4])))
print(list(accumulate([1, 2, 3, 4], (lambda x, y : x * y
    ))))
```

Prints [1, 3, 6, 10] and [1, 2, 6, 24] .

The **default** operation is summing.

What about Benevolent Dictator For Life?

“So now reduce(). This is actually the one I’ve always hated most, because, apart from a few examples involving + or *, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what’s actually being fed into that function before I understand what the reduce() is supposed to do. So in my mind, the applicability of reduce() is pretty much limited to associative operators, and in all other cases it’s better to write out the accumulation loop explicitly.”

— Guido van van Rossum, *The fate of reduce() in Python 3000*, 2005

“Readability is often enhanced by reducing unnecessary variability. When possible, there’s a single, obvious way to code a particular construct. This reduces the number of choices facing the programmer who is writing the code, and increases the chance that will appear familiar to a second programmer reading it.”

— Guido van van Rossum, *Foreword for "Programming Python"(1st ed.)*, 1996

Digression: PEP 20 – The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess
...
```

Digression: PEP 20 – The Zen of Python

```
There should be one-- and preferably only one --obvious
    way to do it.
Although that way may not be obvious at first unless you
    're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad
    idea.
If the implementation is easy to explain, it may be a
    good idea.
Namespaces are one honking great idea -- let's do more
    of those!
```

Memoization

```
from functools import lru_cache
# LRU = Least Recently Used

@lru_cache(maxsize = None)
def fib_lru(n):
  if n <= 2:
    return 1
  return fib_lru(n - 1) + fib_lru(n - 2)

print([fib_lru(n) for n in range(1,10)])
print(fib_lru.cache_info())
```

Memoization

The code from the previous slide would print:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34]
CacheInfo(hits=14, misses=9, maxsize=None, currsize=9)
```

Since 3.9 one can use a decorator `@cache`, equivalent to the decorator `@lru_cache` with the parameter `maxsize = None`.

Lambda expressions

```
#include <algorithm>
...
void wyswietlPlus1(int a)
{
  cout << a + 1 << ", ";
}
...
int tab[] = {0, 1, 2, 3};
...
for_each(tab, tab + 4, &wyswietlPlus1); \\ prints: 1, 2,
    3, 4,
```

Is it worth defining a specific function `void wyswietlPlus1(int a)` ?

Lambda expressions

```
for_each(tab, tab + 4,
  [](int a)
  {
    cout << a + 1 << ", ";
  }
);
```

Alternatively:

```
auto wyswietlPlus1_lambda = [](int a) {cout << a + 1 <<
    ", ";};
for_each(tab, tab + 4, wyswietlPlus1_lambda);
```

Class template std :: function

What is auto in lambda expressions?

Note: Using the -> operator you can explicitly specify the returned data type (otherwise it is deduced by the compiler).

```
#include <functional>
...
function<double(double)> x2_lambda = [](double x) ->
    double{return x * x;}; // jawne okreslenie typu
    zwracanego
function<int(double)> x2_lambda_int = x2_lambda; //
    rzutowanie wyniku na int
...
cout << x2_lambda(3.1) << endl; // 9.61
cout << x2_lambda_int(3.1) << endl; // 9
```

Lambda expressions and `this`

In short, an expression `[]() {};` is a full-fledged lambda expression in C ++.

Violating the principle that functions in functional programming do not change (and do not depend on) the state of the environment, we can allow the lambda expression to access other variables visible in the environment (given by the pointer `this`):

- using `[=]` (variables passed by value),
- or `[&]` (variables passed by reference),
- or explicitly listing which variables are to be available.

`[]` is the so-called *capture list*.

Map in C++

```
#include <algorithm>
...
vector<int> tab1 = {0, 1, 2, 3};
vector<int> tab_wynik;
...
transform(tab1.begin(), tab1.end(), back_inserter(
    tab_wynik), [](int a){return a * a;}); // tab_wynik =
     {0, 1, 4, 9}
```

zipWith in C++

```
#include <algorithm>
...
vector<int> tab1 = {0, 1, 2, 3};
vector<int> tab2 = {100, 200, 300, 400};
vector<int> tab_wynik;
...
transform(tab1.begin(), tab1.end(), tab2.begin(),
    back_inserter(tab_wynik), [](int a, int b){return a *
     a + b;}); // tab_wynik = {100, 201, 304, 409}
```

Filter in C++

```
#include <algorithm>
...
vector<int> tab1 = {0, 1, 2, 3};
...
copy_if(tab1.begin(), tab1.end(), back_inserter(
    tab_wynik), [](int a){return a % 2 == 0;}); //
    tab_wynik = {0, 2}
```

Fold in C++

```
#include <numeric>
...
vector<int> tab3 = {1 -10, 17, 1002, -5, 23, 555};
...
cout << accumulate(tab3.begin(), tab3.end(), 0) << endl;
     // suma
cout << accumulate(tab3.begin() + 1, tab3.end(), tab3
    [0], [](int a, int b){return a > b ? a : b;}) << endl
    ; // wartosc maksymalna
```

As a result it will print `1583` and `1002`.

Note that `accumulate` in C++ works differently than in Python.

Lambda calculus in C# - LINQ

Lambda expressions in C# are especially useful for LINQ queries (Language INtegrated Query):

```
int[] scores = { 90, 71, 82, 93, 75, 82 };

// LINQ
int nHScore = scores.Where(n => n > 80).Count();

// nHScore jest rowne 4
```

Complex expressions LINQ

```
Enumerable.Range(1, 100).
   Where(i => i % 20 == 0).
   OrderBy(i => -i).
   Select(i => $"{i}%")
   // => ["100%", "80%", "60%", "40%", "20%"]
```

E. Buonanno, *Functional Programming in C#*, 2017.

Monads - Hello World

The monads confuse most of Haskell's beginner.

There are a lot of tutorials, a few are created every year trying to explain the monads again: `https://wiki.haskell.org/Monad_tutorials_timeline`.

The main reason for the problems: monads are relatively abstract in relation to the moment when a beginner encounters them.

Typical first program: `main = putStrLn "Hello, World!"`

Monads - IO

As promised, all functions are clean (they do not change anything in the world, display nothing, do not load anything). To keep this word one had to cheat...

Where is the monad here? `main = putStrLn "Hello, World!"` ?

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

IO is a monad. But what does it mean?

Monads - IO

What does it mean? Just a moment...

Sample program with input and output operations:

```
module Main where

out1 = (putStrLn "Podaj liczbe:")
in1 = (readLn :: IO Double)
out2 = (\ x -> putStrLn ("\nsin(" ++ show x ++ ") = " ++
    show (sin(x))))

main = out1 >> in1 >>= out2
```

We discussed a similar program before, but using the `do` construct, which is a so-called *syntactic sugar*.

Monads - IO

Let's see types of the expressions:

```
out1  :: IO ()
in1  :: IO Double
out2  :: Double -> IO ()
-- previous slide reminder: out1 >> in1 >>= out2

(>>)  :: Monad m => m a -> m b -> m b -- monad sequence

(>>=)  :: Monad m => m a -> (a -> m b) -> m b -- sequence
     of monads with passing values
```

We see the scheme: operators >> and >>= combine expressions with the monads at the beginning and end.

Monads - IO - as a whole

```haskell
module Main where

out1 :: IO ()
out1 = (putStrLn "Enter a number:")

in1 :: IO Double
in1 = (readLn :: IO Double)

out2 :: Double -> IO ()
out2 = (\ x -> putStrLn ("\nsin(" ++ show x ++ ") = " ++
    show (sin(x))))

main = out1 >> in1 >>= out2
```

Monads - time for definition

Standard definition of the Haskell monad:

```haskell
class (Applicative m) => Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    return  :: a -> m a
```

Function `>>=` takes as an argument a **monad** (`m a`) and **a function that returns a monad** (`a -> m b`) and transforms them into a **monad** (`m b`)!

In other words `>>=` “takes” `a` from the monad `m a` and passes them to a function that will do something on it (get `b`) and turn it back into a monad, this time `m b` .

Monads - some side notes

The definition seems simple (or at least short), but the monad instance must inherit from the Applicative class, which inherits from the Functor class.

You do not need to use this (default) definition when defining your own monads. Custom types do not have to be instances of the Monad class to implement the monad concept.
`https://wiki.haskell.org/All_About_Monads`

Note: Only from version Haskell 7.10 (2014) the so-called Applicative => Monad proposal (AMP) - previously the monad did not inherit from the Applicative class, but only from Functor itself. `https://wiki.haskell.org/Functor-Applicative-Monad_Proposal`

What are functors?

Functor is a class that contains a definition of fmap :

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Compare: map :: (a –> b) –> [a] –> [b] .

Reasonable fmap should fulfil the laws of composition and identity (the **functor laws**):

```
fmap (f . g) = fmap f . fmap g -- Composition law
fmap id = id -- Identity law
```

Note: Not to be confused with functors from propositional calculus $(\wedge, \vee, \implies, \dots)$.

Functor laws

To be more precise...
If one composes two or more functions and pass that to fmap , then fmap 's output must be the same as the output of a composition of multiple calls of fmap separately for each function.

```
fmap (f . g) = fmap f . fmap g — Composition law
```

If one passes identity to fmap , fmap 's output must be equal to its input (*mapping identity does nothing*).

```
fmap id = id — Identity law
```

Monads - life examples

As it was mentioned, it is good to think about monads in an abstract way, without looking for any deeper comparisons with real terms. Comparisons are de facto examples of monads, e.g.

Functor is a “wrapping” of the value with the possibility of converting this value into another using the usual function.

Applicative can have “wrapped” not only values, but also functions and can work “wrapped” function on “wrapped” values.

A monad can put “wrapped” functions into sequences.

`http://adit.io/posts/2013-04-17-functors, _applicatives,_and_monads_in_pictures.html`

Monads - life examples

Other examples:

- fish eating fish and stones and cooperating with crabs http://maciejpirog.github.io/fishy/
- production line with semi-finished products in trays https://web.archive.org/web/20100910074354/http://www.haskell.org/all_about_monads/html/analogy.html
- programmable semicolons (; of imperative languages)
- utilizers of waste transported in containers https://en.wikibooks.org/w/index.php?title=Haskell/Understanding_monads&oldid=933545#The_nuclear_waste_metaphor

Monads - life examples

Other examples:

- slaughtered monsters with flowing guts `https://mail.haskell.org/pipermail/haskell-cafe/2006-November/019190.html`
- House of Lannisters from Game of Thrones `http://www.snoyman.com/blog/2016/09/monads-are-like-lannisters`

Creating examples of monads has been mocked e.g. in the poem of Michael Beker, *The Community Guide to Monads*, `https://8thlight.com/blog/michael-baker/2012/08/11/monads.html`

LISt Processor

The second oldest high-level programming language after Fortran (John McCarthy, 1958).

Has many dialects. Currently, the most popular are Common Lisp and Scheme.

Lists are the basic data structure. Even the LISP code is a list.

LISP is a multi-paradigm language. It allows to combine functional and imperative paradigm.

Symbols in LISP

Symbols in LISP are sequences of upper and lower case letters, numbers and dashes started with a letter.

Letters are not case-sensitive.

Special symbols are `NIL` (false) and `T` (true).

Expressions in LISP

All expressions are lists. The first element of the list defines the operation, and the remaining elements its arguments, e.g.

```
[1]> (+ 2 2)
4
[2]> (/ 1 3)
1/3
[3]> (/ 1 3.0)
0.33333334
[4]> (mod 7 3)
1
[5]> (mod -7 3)
2
[6]> (rem 7 3)
1
```

Expressions in LISP

Of course, expressions can be nested:

```
[1]> (* (+ 3 4) (/ 1 13))
7/13
```

Operators in LISP

Most operators can take more arguments than usual (compare with the fold operation in Haskell):

```
[13]> (+ 1 2 3)
6
[14]> (- 1 2 3)
-4
[15]> (/ 70 7 5)
2
```

Comment: $1 + 2 + 3 = 6$, $1 - 2 - 3 = -4$, $(70/7)/5 = 2$.

Logical operators

```
[15]> (> 1 2)
NIL
[16]> (<= 1 2)
T
[17]> (or 3 4)
3
[18]> (or 4 3)
4
[19]> (and nil t)
NIL
[20]> (= 2 2)
T
```

Numerical types in LISP

Integers size is limited only by available memory. Complex numbers are created using the operator #c :

```
[10]> (* #c(0 1) #c(0 -2))
2
[11]> -5
-5
[12]> -5.32
-5.32
```

Cons operator

Operator cons :

```
[3]> (cons 3 "dsd")
(3 . "dsd")
```

Value assignment

Using setq , one can assign a value to the symbol:

```
[4]> (setq x 5)
5
[5]> (setq y 2)
2
[6]> (* x y)
10
```

Local value assignment

Using let one can **locally** assign a value to the symbol:

```
[7]> (let ((a 3) (b 5) (c 7)) (+ a b c))
15
```

Logic programming

Logic programming (or programming in logic) is one of the paradigms of declarative programming.

The code is a set of expressions describing **facts** and **dependencies** between them.
Execution of the program is the derivation of **conclusions** corresponding to the user's query.

Logic programming

Languages Planner (1969), **Prolog** (1971), Datalog (1977), Answer set programming (1993), ECLiPSe(1992), Mercury (1995).

Prolog's extension with object-oriented mechanisms: Prolog++ (1989).

Applications: knowledge-based systems, artificial intelligence, natural language processing (Prolog in IBM Watson), automatic theorem proving, cloud computing (Datalog).

Implementations: e.g. SWI-Prolog, Visual Prolog.

Formal system - reminder

Reminder from the first lecture.

The formal system consists of three elements:

1. set of symbols (= alphabet + rules of construction),
2. inference rules,
3. a set of axioms.

Simply anything where based on **assumptions** (3) we can draw **justified** (2) and **understandable** (1) conclusions.

Algorithm = Logic + Control

In logical programming, the algorithm execution is **controlled logical deduction**. The result of the program is the conclusion drawn from the given premises.

The conclusion follows directly from the premises. From the *purely logical* point of view (i.e. relationships between facts), there is no point in talking about the *course* of the relationships.

Algorithm = Logic + Control

However, since **inference is a process** (performed by a machine or human), we can talk about the *process* of inference. This depends on the "implementation" (software, mind structure) that *controls* this process.

Algorithm = Logic + Control (R. Kowalski, 1979).

Example. Suppose the following statements are true:

1. *Marysia and Kuba have one umbrella.*
2. *If it rains in the morning, Kuba takes the umbrella.*
3. *If it rains in the afternoon, Marysia needs the umbrella.*
4. *If Marysia needs and doesn't have the umbrella, Kuba brings it to her.*
5. *It rains in the morning and afternoon.*

From the above, it follows: *Kuba brings Marysia an umbrella.*

The logical relationship between the above sentences *has been the same for centuries*. However, drawing the above conclusion is a process that occurs in time, especially when individual facts and dependencies are added sequentially (e.g. the fact *In the morning and afternoon it rains* appears as input to the program).

Propositional calculus - formulas and predicates

The number of parameters is called **arity**.

Functions expressing relationships between variables (not only logical) are so-called **predicates**.
$P_\phi(p, q, r)$ is a predicate of arity 3.

Automated theorem proving is possible within propositional calculus.

Atoms in Prolog

The simplest expressions in Prolog are called **atoms** (constants or strings) and do not mean anything in themselves.
In order to check if something is an atom one can use a predicate atom/1 ("/1" means the predicate is of arity 1).

```
?- atom(characters_and_UNDERSCORE_with_SMALL_letter).
true.

?- atom(A_with_CAPTIAL).
false.

?- atom(12).
false.

?- atomic(12).
true.
```

Atoms in Prolog

```
?- atom(q).
true.

?- atom(Q).
false.

?- atom('Q').
true.

?- atom('a sample text with spaces or not').
true.
```

Variables in Prolog

Variables (or more properly, *unknowns*) have names starting with a captial letter or underscore:

```
?- var(This_is_a_variable). % with CAPITAL letter

true.

?- var(_this_is_a_variable). % starts with underscore

true.
```

Variables in Prolog

```
?- var(_). % anonymous variable, one cannot refer to it

true.

?- var(and_this_is_atom). % this not a variable

false.
```

Numeric types in Prolog

In Prolog, we have two numeric types, integer and real:

```
?- integer(2).
true.

?- integer(2.1).
false.

?- integer(2.0).
false.

?- number(2.0).
true.

?- number(2.1).
true.

?- number(a).
false.
```

Predicates in Prolog

Predicates with substituted values express **facts** about the world that we want to include in the inference:

```
lubi(paulina, ola).
lubi(paulina, piotr).
lubi(piotr, rafal).
lubi(ola, paulina).
```

We decide ourselves what we mean by a given predicate, e.g. lubi(X,Y). may mean that X likes Y.
Note: In Polish *lubi* means “likes”.

Predicates in Prolog

Enumeration of relations:

```
?- lubi(X,Y).
X = paulina ,
Y = ola  ;
X = paulina ,
Y = piotr  ;
X = piotr ,
Y = rafal  ;
X = ola ,
Y = paulina .
```

Predicates in Prolog

Enumeration of relations:

```
?- lubi(paulina,X).
X = ola ;
X = piotr.

?- lubi(X,paulina).
X = ola.
```

It is usually convenient to list all the facts in an external file (typical extension: .pl). To load data from the file file.pl one use consult('file.pl').

Predicates in Prolog

Conjunction ($\wedge$) is expressed in the Prolog using a comma:

```
lubi(paulina, ola).
lubi(paulina, piotr).
lubi(piotr, rafal).
lubi(ola, paulina).
...
?- lubi(paulina, ola),lubi(paulina, piotr).
true.

?- lubi(X, ola),lubi(X, piotr).
X = paulina.

?- lubi(paulina,X),lubi(X,rafal).
X = piotr.
```

Predicates in Prolog

Alternative ($\vee$) is expressed in the Prolog using a semicolon:

```
lubi(paulina, ola).
lubi(paulina, piotr).
lubi(piotr, rafal).
lubi(ola, paulina).
 ...
?- lubi(X,ola);lubi(X,rafal).
X = paulina ;
X = piotr.
```

Resolution

The resolution is the **automatic theorem proving** method in which subsequent clauses are generated based on earlier ones.

Let $a_1, \ldots, a_m$, $b_1, \ldots, b_n$ and c be formulas or their negations (so-called *literals*).

If we know that it holds $a_1 \lor a_2 \ldots a_m \lor c$ **and** $b_1 \lor b_2 \ldots b_n \lor \neg c$, then we may infere that

$$a_1 \lor a_2 \ldots a_m \lor b_1 \lor b_2 \ldots b_n.$$

In particular, from the inference rule *modus ponens*: If $\neg p \lor q$ (i.e. $p \implies q$) and p, then q.

Horn Clauses

From the point of view of programming in logic, the so-called Horn clauses are particularly important.

Let $a_0, a_1, \dots a_m$ be literals. Horn Clauses are of the form:

$$\underbrace{a_0}_{\text{głowa}} \Longleftarrow \underbrace{a_1 \wedge a_2 \wedge \cdots \wedge a_m}_{\text{ciało}},$$

or, equivalenty

$$a_0 \vee \neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_m.$$

Note: $q \Longleftarrow p$ is equivalent $p \Longrightarrow q$.

Resolution on Horn clauses

Each Horn clause has at most one non-negated formula.

By properly choosing a pair of Horn clauses and carrying out a resolution, we can as a result obtain another Horn clause (and reject the matched pair). In this way we reduce the number of Horn clauses.

Facts, or Horn clauses with one non-negated formula without free variables, can reduce the length of the Horn clause by 1.

Inference in the Prolog

Let us express some more facts from which one can infer:

```
plec(ewa, k).
plec(ala, k).
plec(adam, m).
plec(marek, m).
rodzic(ewa, ala).
rodzic(ewa, marek).
rodzic(adam, marek).
matka(R, P) :- rodzic(R, P), plec(R, k). % Horn clause
```

(Example by dr K.M. Ocetkiewicz.)
Note: In Polish *płeć* means “gender”, *rodzic* means “a parent”, *matka* means “a mother”.

Inference in the Prolog

Sample inferences:

```
?- matka(ewa,marek).
true.

?- matka(adam,marek).
false.

?- matka(ala,ewa).
false.

?- matka(ewa,Y).
Y = ala ;
Y = marek.
```

Inference in the Prolog

How does inference work? We have a rule:

```
matka(R, P) :- rodzic(R, P), plec(R, k).
```

and a query matka(ewa,marek)..

We bind R with ewa and P with marek.

For matka(R, P) to be true, the following two has to hold: rodzic(R, P) and plec(R, k).

Subqueries are evaluated **left to right**. First, the subquery rodzic(ewa, marek). is evaluated, here it is "hit"; then the evaluated subquery plec(ewa, k).is also "hit". Both subqueries are "hit", so "hit"is also the conclusion that matka(ewa,marek). holds.

Keyword is

A prime number X is Mersenne if it is of the form $2^Y - 1$ for some natural Y. Example with keyword is:

```
p(2).
p(3).
p(5).
p(7).
...
p(23).
p(29).
p(31).
malaMersenne(X):-p(X),between(0,5,Y),pow(2,Y,Z),(X is Z
    - 1).
```

The query smallMersenne(X) returns 3, 7 and 31.
Note: In Polish *mała* means “small”.

Recursion in Prolog

An example of a recursively defined function: The power of two.

```
pow2(0,1).
pow2(Y,Z) :- Y1 is Y - 1, pow2(Y1,Z1), Z is Z1*2.
```

1 is zero power 2.

Z is equal 2^Y when $Z = 2 \times 2^{Y-1}$.

Recursion in Prolog - lists

Checking if an item is on the list of items:

```
naLiscie(X, [X|_]). % is when it is the first element
naLiscie(X, [_|T]):-naLiscie(X, T). % is when it is on
    the list without the first item
...
?- naLiscie(1, [1,2,3,4,5]).
true .

?- naLiscie(7, [1,2,3,4,5]).
false.
```

Note: In Polish *na liście* means “on a list”.

Recursion in Prolog - lists

Print list using the above predicate:

```
?- naLiscie(X, [1,2,3,4,5]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 ;
false.
```

Recursion in Prolog - lists

Length of the list:

```
dlugosc([], 0). % dlugosc listy pustej to 0
dlugosc([_|T], Lp):-dlugosc(T, L), Lp is L + 1. %
    dlugosc listy z jednym elementem wiecej
...
?- dlugosc([1,2,3,4,5],X).
X = 5.
```

Operator cut

The prologue makes inferences by examining various possible values of the variables and lists them all. The operator ! stops and breaks the consideration of further possibilities on the part of evaluation found so far (please reflect for a moment):

```
q(10).
q(20).
r(10,'a').
r(10,'b').
r(20,'c').
pierwszeQ(L):-q(X),!,r(X,L).
wszystkieQ(L):-q(X),r(X,L).
```

Note: In Polish *pierwsze* means “the first”, *wszystkie* means “all”.

Operator cut

The operator ! stops considering other possibilities of satisfying the part of the evaluation analysed so far:

```
?- pierwszeQ(L).
L = a ;
L = b.

?- wszystkieQ(L).
L = a ;
L = b ;
L = c.
```

Console interactions

Move to new line with *predicate* nl/0.

Printing *predicate* write/1 (write to current output).

Read input using *predicate* read/1.

```
czytaj(A):-nl,write('Podaj cos: '),read(A).
...
?- czytaj(A).

Podaj cos: asdfgh.
A = asdfgh.
```

Note: In Polish *czytaj* means “read”, *Podaj coś* means “Enter something”.

Write to file

Open the file with *predicate* tell /1.

Write with *predicate* write/1 (write to current output).

Close the file with *predicate* told/0.

```
?- tell('plik.out'),write('napis w pliku'),told.
true.
```

Note: In Polish *plik* means “file”.

Dziękuję za
uwagę!